AequilibraE 2.0.0rc1

Pedro Camargo

Nov 19, 2026

CONTENTS

1 Installation 1

2 The AequilibraE project 5

3 Run module 75

4 Network Manipulation 79

5 Distribution Procedures 113

6 Path Computation 129

7 Static Traffic Assignment 141

8 Public Transport 213

9 Route Choice 239

10 Other Applications 259

11 API Reference 267

Python Module Index 321

Index 323

CHAPTER

ONE

INSTALLATION

In this section we describe how to install AequilibraE. The recommendations on this page are current as of September 2024.

Important

Although AequilibraE is under intense development, we try to avoid making breaking changes to the API. In any case, you should check for new features and possible API changes often.

1.1 Installation

1. Install Python 3.11, 3.12, 3.13 & 3.14. We recommend Python 3.12 or 3.13
2. Install AequilibraE

```
pip install aequilibrae
```

Python installations from the Windows store are NOT SUPPORTED

The Windows App Store ships a version of Python that contains an sqlite dll that does not support the loading of extensions. This means that Spatialite will not be loaded, and therefore AequilibraE will not work properly.

1.1.1 Installing the 2.0 release candidate

To try AequilibraE 2.0.0rc1 without replacing your 1.x installation, run this command in a separate virtual environment:

```
python -m pip install "aequilibrae==2.0.0rc1"
```

The 2.0 release candidate introduces breaking changes. Read the 2.0 migration guide before opening an existing project, and use a copy of the project for testing.

1.1.2 macOS

AequilibraE does not provide pre-built wheel files for macOS. When installing from PyPi, the source distribution will be used and the library will be compiled locally. AequilibraE can also be built from source. For both methods you will need to:

1. Install homebrew, a package manager for macOS, if you do not have it already.
2. Install LLVM or another C/C++ compiler with OpenMP support: `brew install llvm`

3. Install libspatialite: `brew install libspatialite` as per this answer on Stack Overflow
4. Set the C and C++ compilers: `export CXX=/opt/homebrew/opt/llvm/bin/clang++` and `export CC=/opt/homebrew/opt/llvm/bin/clang`
5. Set the `AEQ_SPATIALITE_DIR` environment variable to the directory containing `mod_spatialite`: `export AEQ_SPATIALITE_DIR="$(brew --prefix)/lib"`

Alternatively, run all steps at once:

```
brew install llvm
brew install libspatialite
export CXX=/opt/homebrew/opt/llvm/bin/clang++
export CC=/opt/homebrew/opt/llvm/bin/clang
export AEQ_SPATIALITE_DIR="/opt/homebrew/lib"
```

AequilibraE may also require raising the "open files" limit, this can be achieved with `ulimit -n 10240`. This should be placed in `.zshrc` or similar user shell configuration file.

1.2 Dependencies

All of AequilibraE's dependencies are readily available from PyPI for all currently supported Python versions and major platforms.

Although the presence of SpatiaLite is rather ubiquitous in the GIS ecosystem, it has to be installed separately from Python or AequilibraE in any platform.

This blog post has a more comprehensive explanation of what is the setup you need to get SpatiaLite working, but that is superfluous if all you want is to get it working.

1.2.1 Windows

Note

On Windows ONLY, AequilibraE automatically verifies if you have SpatiaLite installed in your system and downloads it to your temporary folder if you do not.

SpatiaLite does not have great support on Python for Windows. For this reason, it is necessary to download SpatiaLite for Windows and inform and load it to the Python SQLite driver every time you connect to the database.

One can download the appropriate version of the latest SpatiaLite release directly from its project page , or the cached versions on AequilibraE's website for 64-Bit Python

After unpacking the zip file into its own folder (say `D:/spatialite`), one can *temporarily* add the SpatiaLite folder to system path environment variable, as follows:

```
import os
os.environ['PATH'] = 'D:/spatialite' + ';' + os.environ['PATH']
```

For a permanent recording of the SpatiaLite location on your system, please refer to the blog post referenced above or Windows-specific documentation.

1.2.2 Ubuntu Linux

On Ubuntu it is possible to install SpatiaLite by simply using apt-get

```
sudo apt update -y
sudo apt install -y libsqlite3-mod-spatialite
sudo apt install -y libspatialite-dev
```

1.2.3 MacOS

On MacOS one can use brew as per this answer on Stack Overflow.

```
brew install libspatialite
```

1.3 Hardware requirements

How much hardware AequilibraE needs is driven mostly by the size of the model being used. The most important things to keep an eye on are:

- Number of zones on your model (size of the matrices you are dealing with)
- Number of matrices (vehicles classes (and user classes) you are dealing with)
- Number of links and nodes on your network (far less likely to create trouble)

Substantial testing has been done with large real-world models (up to 8,000 zones) and memory requirements did not exceed the traditional 32Gb found in most modeling computers these days. In most cases 16Gb of RAM is enough even for large models (5,000+ zones). Computationally intensive procedures such as skimming and traffic assignment have been parallelized, so AequilibraE can make use of as many CPUs as there are available in the system for such procedures.

CHAPTER

TWO

THE AEQUILIBRAE PROJECT

Similarly to commercial packages, any AequilibraE project must have a certain structure and follow a certain set of guidelines in order for software to work correctly.

One of these requirements is that AequilibraE currently only supports one projection system for all its layers, which is the **EPSG:4326** (WGS84). This limitation is planned to be lifted at some point, but it does not impact the result of any modeling procedure.

AequilibraE is built on the shoulder of much older and more established projects, such as SQLite, SpatiaLite and NumPy, as well as reasonably new industry standards such as the OpenMatrix format.

The performance, portability, self containment, and open-source character of these pieces of software, along with their large user base and wide industry support make them solid options to be AequilibraE's data backend.

Since working with SpatiaLite is not just a matter of a `pip install`, please refer to *Dependencies*. For QGIS users this is not a concern, while for Windows users this dependency is automatically handled under the hood, but the details are also discussed in the aforementioned dependencies section.

2.1 Package components: A conceptual view

As all the components of an AequilibraE model are based on open-source software and open-data standards, modeling with AequilibraE does not feel quite like modeling with commercial packages, as the user can read and manipulate model components outside the software modeling environments (Python and QGIS).

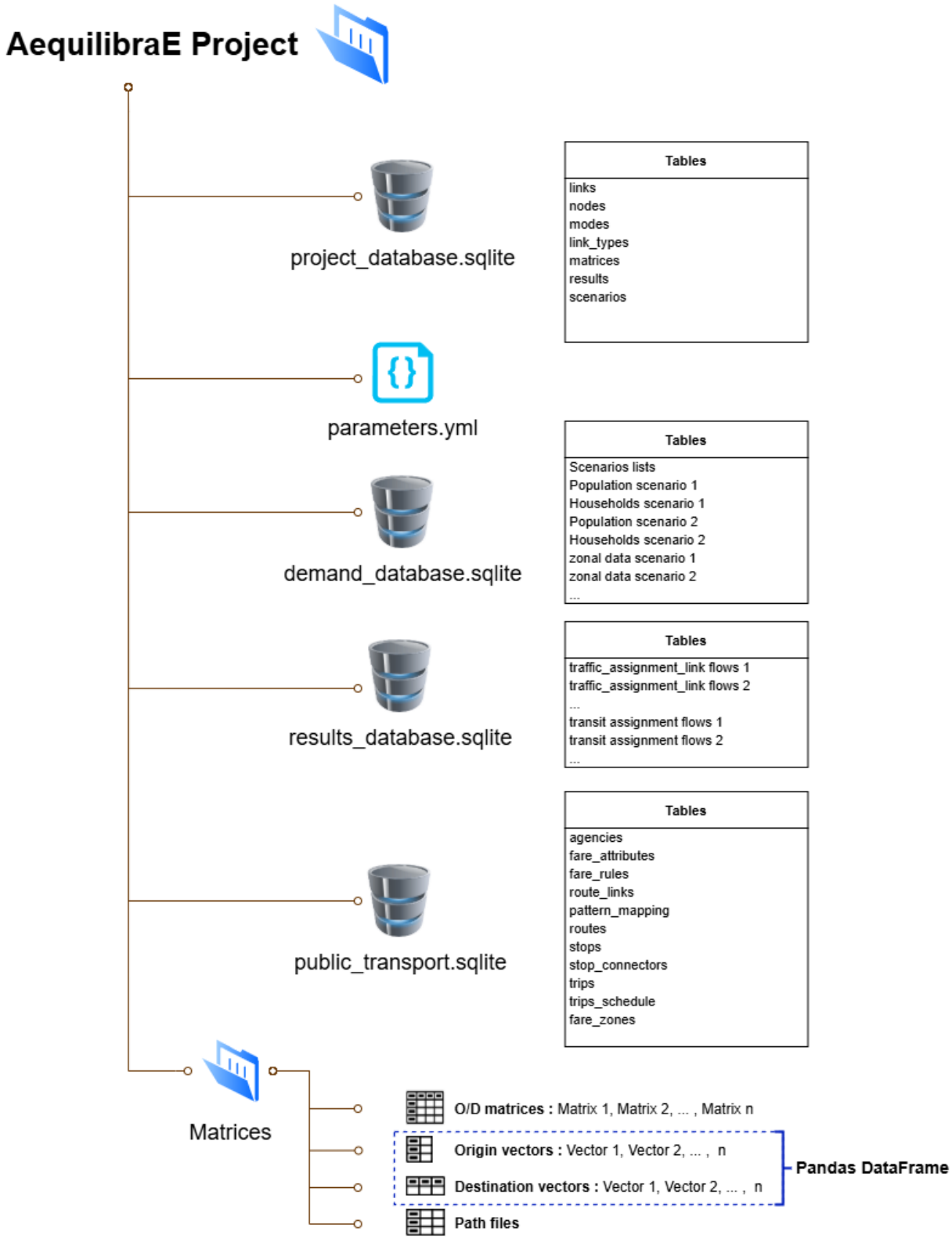
Thus, using/manipulating each one of an AequilibraE model components can be done in different ways depending on the tool you use for such.

It is then important to highlight that AequilibraE, as a software, is divided in three very distinctive layers. The first, which is responsible for tables consistent with each other (including links and nodes, modes and link_types), are embedded in the data layer in the form of geo-spatial database triggers. The second is the Python API, which provides all of AequilibraE's core algorithms and data manipulation facilities. The third is the GUI implemented in QGIS, which provides a user-friendly interface to access the model, visualize results and run procedures.

These software layers are *stacked* and depend on each other, which means that any network editing done in SQLite, Python or QGIS will go through the SpatiaLite triggers, while any procedure such as traffic assignment done in QGIS is nothing more than an API call to the corresponding Python method.

2.2 Project structure

Since version 0.7, the AequilibraE project consists of a main folder, where a series of files and sub folders exist, and the current project organization is as follows:


AequilibraE Project
project_database.sqlite
Tables
links
nodes
modes
link_types
matrices
results
scenarios
parameters.yml
demand_database.sqlite
Tables
Scenarios lists
Population scenario 1
Households scenario 1
Population scenario 2
Households scenario 2
zonal data scenario 1
zonal data scenario 2
...
results_database.sqlite
Tables
traffic_assignment_link flows 1
traffic_assignment_link flows 2
...
transit assignment flows 1
transit assignment flows 2
...
public_transport.sqlite
Tables
agencies
fare_attributes
fare_rules
route_links
pattern_mapping
routes
stops
stop_connectors
trips
trips_schedule
fare_zones
Matrices
O/D matrices : Matrix 1, Matrix 2, ... , Matrix n
Origin vectors : Vector 1, Vector 2, ... , n
Destination vectors : Vector 1, Vector 2, ... , n
Pandas DataFrame
Path files

The main component of an AequilibraE model is the **project_database.sqlite**, where the network and zoning system are stored and maintained. Matrix metadata is stored in the OMX files themselves, and procedure result metadata is stored alongside the result tables in **results_database.sqlite**.

The second key component of any model is the **parameters.yaml** file, which holds the default values for a number of procedures (e.g. assignment convergence), as well as the specification for networks imported from OpenStreetMap and other general import/export parameters.

The third and last required component of an AequilibraE model is the **Matrices folder**, where all the matrix files in OMX format should be placed. This folder can be empty, however, as no particular matrix is required to exist in an AequilibraE model.

The database that stores results in tabular format (e.g. link loads from traffic assignment), **results_database.sqlite**, also stores the **results** metadata table. It is created on-the-fly the first time `project.results` or `project.results_connection` is accessed, so the user does not need to create it manually.

The **demand_database.sqlite** is envisioned to hold all the demand-related information, and it is not yet structured within the AequilibraE code, as there is no pre-defined demand model shipped with AequilibraE. The database itself is not created with the model, but we recommend using this concept on your demand models.

The **public_transport.sqlite** database holds a transportation route system for a model, and has been introduced in AequilibraE version 0.9. This database is also created on-the-fly when the user imports a GTFS source into an AequilibraE model, but there is still no support for manually or programmatically adding routes to a route system as of yet.

Optionally a project can be configured with a **scenarios** folder. This allows the storage of many AequilbraE models within one directory, however it is intended to store modifications of some other project. The scenario system is built around a hierarchical structure where the "root" scenario represents the base model, and additional scenarios are created as branches from this foundation. Users can create entirely new scenarios with empty networks or clone existing scenarios to explore modifications. Scenarios are automatically isolated from each other, ensuring that changes made in one scenario do not affect others. The system integrates with all existing AequilibraE functionality, including traffic assignment, transit modelling, matrix operations, and results management.

In the following sections, we present the structure of each component of an AequilibraE project.

2.2.1 Project database

In this section we discuss on a nearly per-table basis the role of each table for an AequilibraE model. In the end, a more technical view of the *database structure*, including the SQL queries used to create each table and the indices used for each table are also available.

Network

The objectives of developing a network format for AequilibraE are to provide the users a seamless integration between network data and transportation modeling algorithms and to allow users to easily edit such networks in any GIS platform they'd like, while ensuring consistency between network components, namely links and nodes. As the network is composed by two tables, **links** and **nodes**, maintaining this consistency is not a trivial task.

As mentioned in other sections of this documentation, the links and a nodes layers are kept consistent with each other through the use of database triggers, and the network can therefore be edited in any GIS platform or programmatically in any fashion, as these triggers will ensure that the two layers are kept compatible with each other by either making other changes to the layers or preventing the changes.

We cannot stress enough how impactful this set of spatial triggers was to the transportation modeling practice, as this is the first time a transportation network can be edited without specialized software that requires the editing to be done inside such software.

Important

As of 2.0, AequilibraE supports turn penalties and bans for path computation, traffic assignment and route choice. See *Turn Restrictions*.

Consistency triggers

The **links** and **nodes** tables are kept consistent with each other, and with the geometries they carry, through the use of database triggers.

Link endpoints must match their geometry

The **a_node** and **b_node** fields on a link must always identify the nodes located at that link's start and end points, respectively. Any update that would leave one of these fields pointing at a node whose geometry does not match the corresponding endpoint is rejected, whether that update targets **a_node/b_node** directly or is an indirect consequence of some other edit. A no-op update, where the value does not actually change, is always allowed, since this is how AequilibraE's own maintenance triggers keep these fields up to date after a geometry or node identity change.

Editing a link's geometry

Moving one of a link's endpoints to a location with no existing node creates a new node there. Moving it on top of an existing node reattaches the link to that node instead. If the node the link used to be attached to is left with no other links connected to it, it is deleted, including when it is a centroid.

Moving and merging nodes

Moving a node's geometry on top of another node's location merges the two: every link attached to the node being replaced is reattached to the moving node, the replaced node is deleted, and the geometry of any link that was attached to either node is updated to match the moving node's new location.

Centroids cannot be merged

A centroid can never be the node that gets replaced by a merge, nor can it be merged into another node, regardless of whether the other node is itself a centroid. Attempting to move a centroid on top of another node, or another node on top of a centroid, fails the transaction.

An empty centroid, i.e. one with no links attached to it, must be demoted (`is_centroid` set to `0`) in a separate statement from any change to its geometry, since demoting an empty centroid deletes it.

Node identity changes

A node's **node_id** cannot be changed in the same statement as its **geometry**. Renumbering a node updates every link that referenced its old **node_id** to point at the new one instead.

See also

- *links table structure*
  Data model
- *nodes table structure*
  Data model

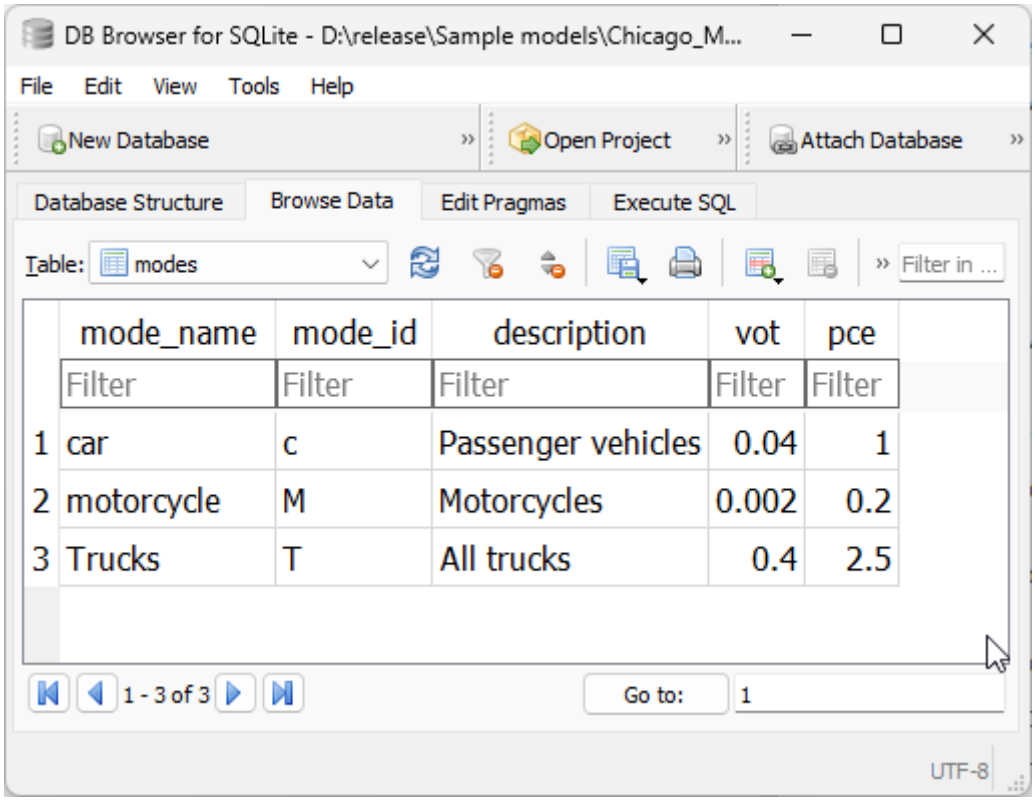
Modes table

The **modes** table exists to list all the modes available in the model's network, and its main role is to support the creation of graphs directly from the SQLite project.

Important

Modes must have a unique mode_id composed of a single letter, which is case-sensitive to a total of 52 possible modes in the model.

As described in the SQL data model, all AequilibraE models are created with 4 standard modes, which can be added to or removed by the user, and would look like the following.



Consistency triggers

As it happens with the links and nodes tables, the modes table is kept consistent with the links table through the use of database triggers.

Changing the modes allowed in a certain link

Whenever we change the modes allowed on a link, we need to check for two conditions:

- At least one mode is allowed on that link
- All modes allowed on that link exist in the modes table

For each condition, a specific trigger was built, and if any of the checks fails, the transaction will fail.

Having successfully changed the modes allowed in a link, we need to update the modes that are accessible to each of the nodes which are the extremities of this link. For this purpose, a further trigger is created to update the modes field in the

nodes table for both of the link's a_node and b_node.

Directly changing the modes field in the nodes table

A trigger guarantees that the value being inserted in the field is according to the values found in the associated links' modes field. If the user attempts to overwrite this value, it will automatically be set back to the appropriate value.

Adding a new link

The exact same behaviour as for *Changing the modes allowed in a certain link* applies in this case, but it requires specific new triggers on the **creation** of the link.

Editing a mode in the modes table

Whenever we want to edit a mode in the modes table, we need to check for two conditions:

- The new mode_id is exactly one character long
- The old mode_id is not still in use on the network

For each condition, a specific trigger was built, and if any of the checks fails, the transaction will fail.

The requirements for uniqueness and non-absent values are guaranteed during the construction of the modes table by using the keys **UNIQUE** and **NOT NULL**.

Adding a new mode to the modes table

In this case, only the first behaviour mentioned above on *Editing a mode in the modes table* applies, the verification that the mode_id is exactly one character long. Therefore only one new trigger is required.

Removing a mode from the modes table

In counterpoint, only the second behaviour mentioned above on *Editing a mode in the modes table* applies in this case, the verification that the old 'mode_id' is not still in use by the network. Therefore only one new trigger is required.

See also

- **`aequilibrae.project.network.modes.Modes()`**
  Class documentation
- ***modes table structure***
  Data model

Link types table

The **link_types** table exists to list all the link types available in the model's network, and its main role is to support processes such as adding centroids and centroid connectors, and to store reference data like default lane capacity for each link type.

Reserved values

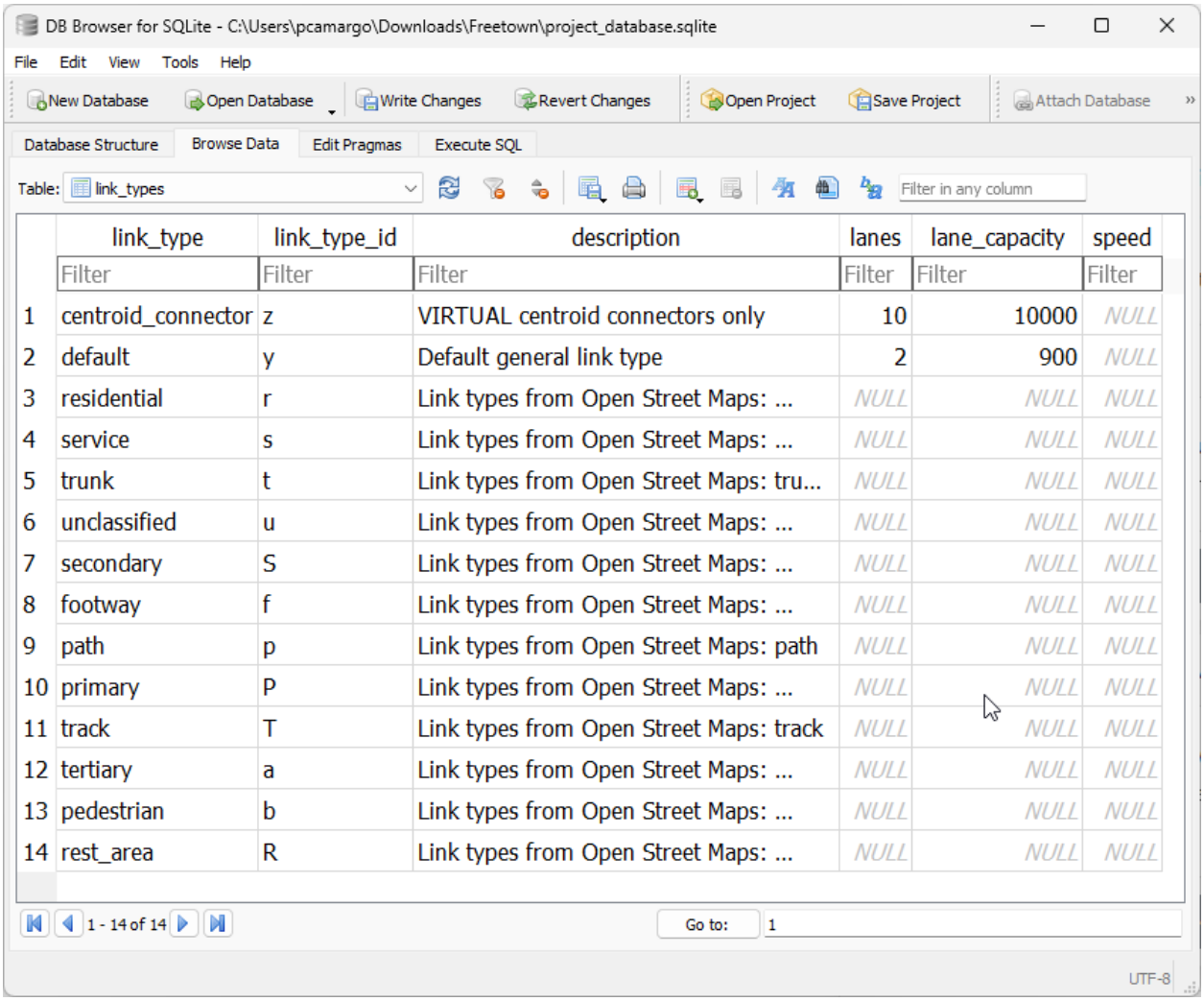
There are two default link types in the link_types table and that cannot be removed from the model without breaking it.

- **centroid_connector** - These are **VIRTUAL** links added to the network with the sole purpose of loading demand/traffic onto the network. The identifying letter for this mode is **z**.

- **default** - This link type exists to facilitate the creation of networks when link types are irrelevant. The identifying letter for this mode is **y**. That is right, you have from **a** to **x** to create your own link types, as well as all upper-case letters of the alphabet.

Adding new link types to a project

Adding link types to a project can be done through the Python API or directly into the 'link_types' table, which could look like the following.



	link_type	link_type_id	description	lanes	lane_capacity	speed
	Filter	Filter	Filter	Filter	Filter	Filter
1	centroid_connector	z	VIRTUAL centroid connectors only	10	10000	NULL
2	default	y	Default general link type	2	900	NULL
3	residential	r	Link types from Open Street Maps: ...	NULL	NULL	NULL
4	service	s	Link types from Open Street Maps: ...	NULL	NULL	NULL
5	trunk	t	Link types from Open Street Maps: tru...	NULL	NULL	NULL
6	unclassified	u	Link types from Open Street Maps: ...	NULL	NULL	NULL
7	secondary	S	Link types from Open Street Maps: ...	NULL	NULL	NULL
8	footway	f	Link types from Open Street Maps: ...	NULL	NULL	NULL
9	path	p	Link types from Open Street Maps: path	NULL	NULL	NULL
10	primary	P	Link types from Open Street Maps: ...	NULL	NULL	NULL
11	track	T	Link types from Open Street Maps: track	NULL	NULL	NULL
12	tertiary	a	Link types from Open Street Maps: ...	NULL	NULL	NULL
13	pedestrian	b	Link types from Open Street Maps: ...	NULL	NULL	NULL
14	rest_area	R	Link types from Open Street Maps: ...	NULL	NULL	NULL

Note

Both 'link_type' and 'link_type_id' MUST be unique

Consistency triggers

As it happens with the links and nodes tables, the 'link_types' table is kept consistent with the links table through the use of database triggers.

Changes to reserved link_types

For both link types mentioned about (**y** & **z**), changes to the 'link_type' and 'link_type_id' fields, as well as the removal of any of these records are blocked by database triggers, as to ensure that there is always one generic physical link type and one virtual link type present in the model.

Changing the link type for a certain link

Whenever we change the 'link_type' associated to a link, we need to check whether that link type exists in the links_table.

This condition is ensured by specific trigger checking whether the new 'link_type' exists in the link table. If if it does not, the transaction will fail.

We also need to update the 'link_types' field the nodes connected to the link with a new string of all the different 'link_type_id's connected to them.

Adding a new link

The exact same behaviour as for *Changing the link type for a certain link* applies in this case, but it requires an specific trigger on the **creation** of the link.

Editing a link type in the *link_types* table

Whenever we want to edit a 'link_type' in the 'link_types' table, we need to check for two conditions:

- The new 'link_type_id' is exactly one character long
- The old 'link_type' is not in use on the network

For each condition, a specific trigger was built, and if any of the checks fails, the transaction will fail.

The requirements for uniqueness and non-absent values are guaranteed during the construction of the 'link_types' table by using the keys **UNIQUE** and **NOT NULL**.

Adding a new link type to the *link_types* table

In this case, only the first behaviour mentioned above on *Editing a link type in the link_types table* applies, the verification that the 'link_type_id' is exactly one character long. Therefore only one new trigger is required.

Removing a link type from the *link_types* table

In counterpoint, only the second behaviour mentioned above on *Editing a link type in the link_types table* applies in this case, the verification that the old 'link_type' is not still in use by the network. Therefore only one new trigger is required.

See also

- `aequilibrae.project.network.link_types.LinkTypes()`
  Class documentation
- *link types table structure*
  Data model

Zones table

The default **zones** table has a **MultiPolygon** geometry type and a limited number of fields, as most of the data is expected to be in the **demand_database.sqlite**.

The API for manipulation of the zones table and each one of its records is consistent with what exists to manipulate the other fields in the database.

As it happens with links and nodes, zones also have geometries associated with them, and in this case they are of the type .

You can check *this example* to learn how to add zones to your project.

See also

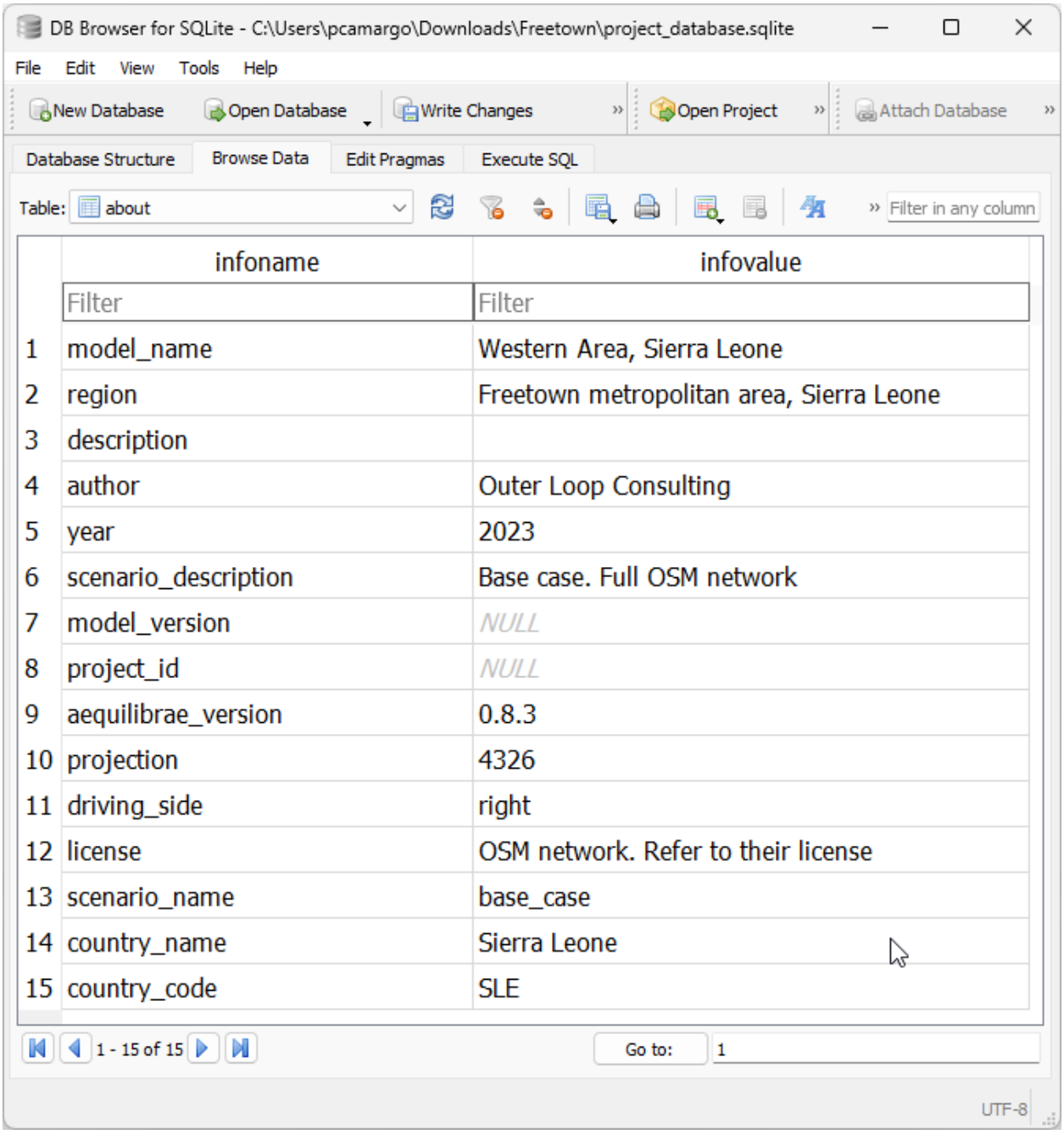
- `aequilibrae.project.zone.Zone()`
  Class documentation
- *zones table structure*
  Data model

About table

The **about** table is the simplest of all tables in the AequilibraE project, but it is the one table that contains the documentation about the project, and it is therefore crucial for project management and quality assurance during modeling projects.

It is possible to create new information fields programmatically. Once the new field is added, the underlying database is altered and the field will be present when the project is open during future use.

This table, which can look something like the example from image below, is required to exist in AequilibraE but it is not currently actively used by any process. We strongly recommend not to edit the information on **projection** and **aequilibrae_version**, as these are fields that might or might not be used by the software to produce valuable information to the user with regards to opportunities for version upgrades.


DB Browser for SQLite - C:\Users\pcamargo\Downloads\Freetown\project_database.sqlite
File Edit View Tools Help
New Database Open Database Write Changes Open Project Attach Database
Database Structure Browse Data Edit Pragmas Execute SQL
Table: about
Filter in any column
infoname infovalue
Filter Filter
1 model_name Western Area, Sierra Leone
2 region Freetown metropolitan area, Sierra Leone
3 description
4 author Outer Loop Consulting
5 year 2023
6 scenario_description Base case. Full OSM network
7 model_version NULL
8 project_id NULL
9 aequilibrae_version 0.8.3
10 projection 4326
11 driving_side right
12 license OSM network. Refer to their license
13 scenario_name base_case
14 country_name Sierra Leone
15 country_code SLE
1 - 15 of 15
Go to: 1
UTF-8

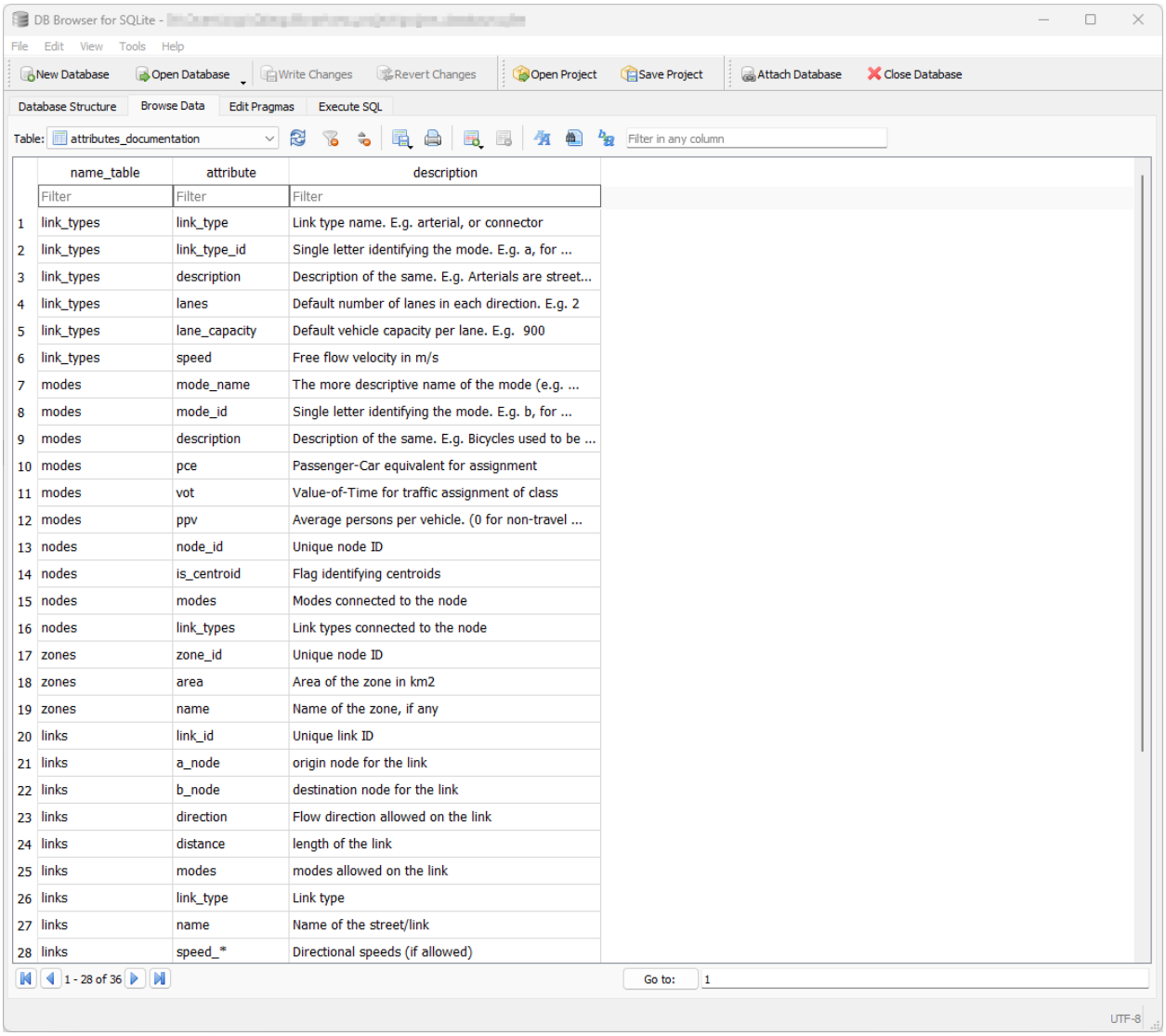
See also

- **`aequilibrae.project.about.About()`**
  Class documentation
- *about table structure*
  Data model

Project attributes

Documentation is paramount for any successful modeling project. For this reason, AequilibraE has a database table dedicated to the documentation of each field in each of the other tables in the project. This table, called **attributes_documentation** can be accessed directly through SQL, but it is envisaged that its editing and consultation would happen through the Python API itself.

As a simple table, it looks as follows:



	name_table	attribute	description
	Filter	Filter	Filter
1	link_types	link_type	Link type name. E.g. arterial, or connector
2	link_types	link_type_id	Single letter identifying the mode. E.g. a, for ...
3	link_types	description	Description of the same. E.g. Arterials are street...
4	link_types	lanes	Default number of lanes in each direction. E.g. 2
5	link_types	lane_capacity	Default vehicle capacity per lane. E.g. 900
6	link_types	speed	Free flow velocity in m/s
7	modes	mode_name	The more descriptive name of the mode (e.g. ...
8	modes	mode_id	Single letter identifying the mode. E.g. b, for ...
9	modes	description	Description of the same. E.g. Bicycles used to be ...
10	modes	pce	Passenger-Car equivalent for assignment
11	modes	vot	Value-of-Time for traffic assignment of class
12	modes	ppv	Average persons per vehicle. (0 for non-travel ...
13	nodes	node_id	Unique node ID
14	nodes	is_centroid	Flag identifying centroids
15	nodes	modes	Modes connected to the node
16	nodes	link_types	Link types connected to the node
17	zones	zone_id	Unique node ID
18	zones	area	Area of the zone in km2
19	zones	name	Name of the zone, if any
20	links	link_id	Unique link ID
21	links	a_node	origin node for the link
22	links	b_node	destination node for the link
23	links	direction	Flow direction allowed on the link
24	links	distance	length of the link
25	links	modes	modes allowed on the link
26	links	link_type	Link type
27	links	name	Name of the street/link
28	links	speed_*	Directional speeds (if allowed)

See also

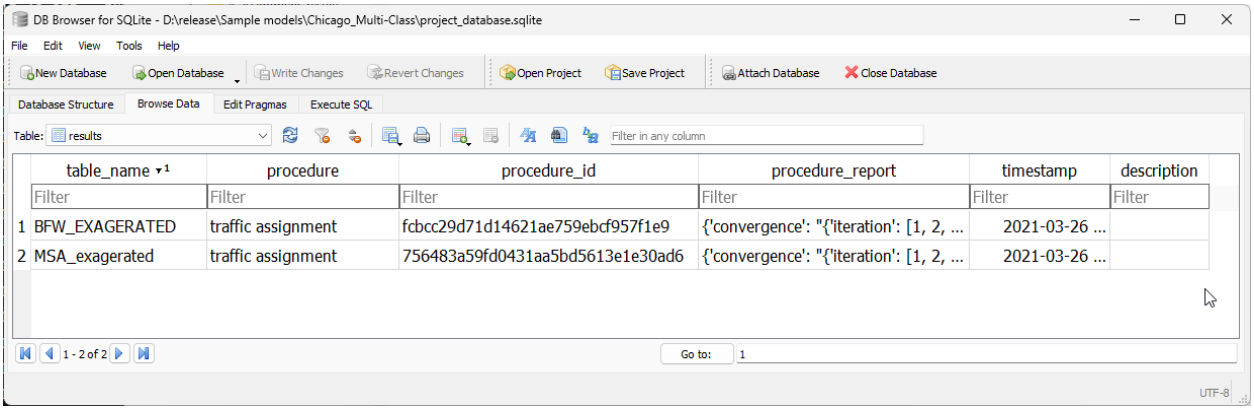
- *attributes documentation table structure*
  Data model

Results table

The **results** table is stored in **results_database.sqlite**, alongside the result data tables whose metadata it holds, in the same folder as the model database. In that, the 'table_name' field is unique and must match exactly the table name in the **results_database.sqlite**.

Although those results could as be stored in the model database, it is possible that the number of tables in the model file would grow too quickly and would essentially clutter the **project_database.sqlite**.

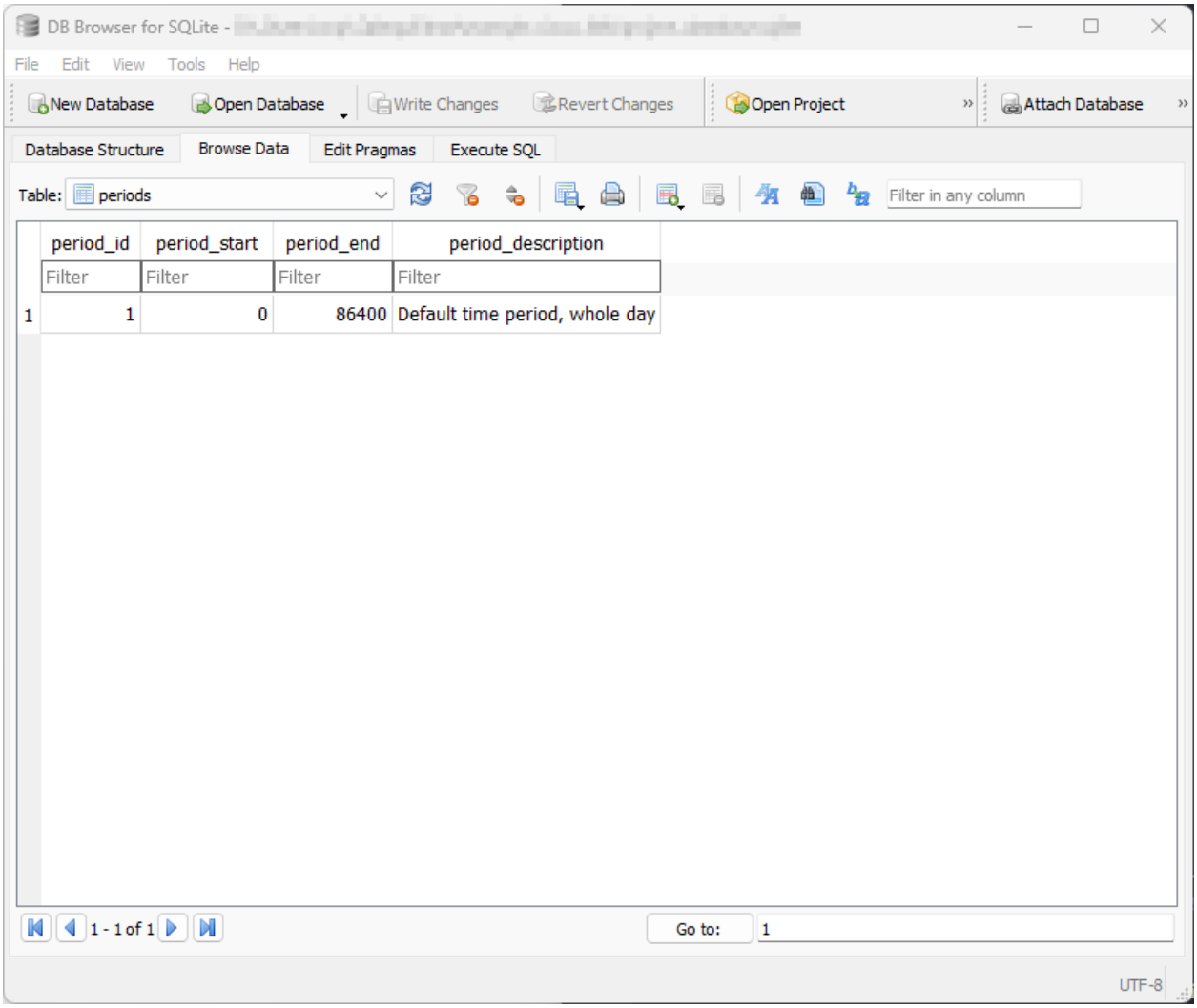
As a simple table, it looks as follows:



See also

- *results table structure*
  Data model

Periods table



See also

- **aequilibrae.project.network.periods.Periods()**
  Class documentation
- *periods table structure*
  Data model

2.2.2 Project database SQL data model

The data model presented in this section pertains only to the structure of AequilibraE's 'project_database' and general information about the usefulness of specific fields, especially on the interdependency between tables.

Conventions

A few conventions have been adopted in the definition of the data model and some are listed below:

- Geometry field is always called **geometry**
- Projection is 4326 (WGS84)
- Tables are all in all lower case

Project tables

about table structure

The *about* table holds information about the AequilibraE model currently developed.

The **infoname** field holds the name of information being added

The **infovalue** field holds the information to add

Field	Type	NULL allowed	Default Value
infoname	TEXT	NO	
infovalue	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists about (infoname  TEXT UNIQUE NOT NULL,
                                   infovalue TEXT
                                  );
INSERT INTO 'about' (infoname) VALUES('model_name');

INSERT INTO 'about' (infoname) VALUES('region');

INSERT INTO 'about' (infoname) VALUES('description');

INSERT INTO 'about' (infoname) VALUES('author');

INSERT INTO 'about' (infoname) VALUES('year');

INSERT INTO 'about' (infoname) VALUES('scenario_description');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'about' (infoname) VALUES('model_version');

INSERT INTO 'about' (infoname) VALUES('project_id');

INSERT INTO 'about' (infoname) VALUES('aequilibrae_version');

INSERT INTO 'about' (infoname) VALUES('projection');

INSERT INTO 'about' (infoname) VALUES('driving_side');

INSERT INTO 'about' (infoname) VALUES('license');

INSERT INTO 'about' (infoname) VALUES('scenario_name');

INSERT INTO 'about' (infoname, infovalue) VALUES('allow_uturns', '0');
```

attributes documentation table structure

The *attributes_documentation* table holds information about attributes in the tables links, link_types, modes, nodes, and zones.

By default, these attributes are all documented, but further attributes can be added into the table.

The **name_table** field holds the name of the table that has the attribute

The **attribute** field holds the name of the attribute

The **description** field holds the description of the attribute

It is possible to have one attribute with the same name in two different tables. However, one cannot have two attributes with the same name within the same table.

Field	Type	NULL allowed	Default Value
name_table	TEXT	NO	
attribute	TEXT	NO	
description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists attributes_documentation (name_table  TEXT NOT NULL,
                                                      attribute   TEXT NOT NULL,
                                                      description TEXT,
                                                      UNIQUE (name_table, attribute)
                                                      );

CREATE INDEX IF NOT EXISTS idx_attributes ON attributes_documentation (name_table,␣
↪attribute);
```

link types table structure

The *link_types* table holds information about the available link types in the network.

The **link_type** field corresponds to the link type, and it is the table's primary key

The **link_type_id** field presents the identification of the link type

The **description** field holds the description of the link type

The **lanes** field presents the number or lanes for the link type

The **lane_capacity** field presents the number of lanes for the link type

The **speed** field holds information about the speed in the link type Attributes follow

Field	Type	NULL allowed	Default Value
link_type*	VARCHAR	NO	
link_type_id	VARCHAR	NO	
description	VARCHAR	YES	
lanes	NUMERIC	YES	
lane_capacity	NUMERIC	YES	
speed	NUMERIC	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists link_types (link_type     VARCHAR UNIQUE NOT NULL PRIMARY␣
↪KEY,
                                        link_type_id  VARCHAR UNIQUE NOT NULL,
                                        description   VARCHAR,
                                        lanes         NUMERIC,
                                        lane_capacity NUMERIC,
                                        speed         NUMERIC
                                        CHECK(LENGTH(link_type_id) == 1));

INSERT INTO 'link_types' (link_type, link_type_id, description, lanes, lane_capacity)␣
↪VALUES('centroid_connector', 'z', 'VIRTUAL centroid connectors only', 10, 10000);

INSERT INTO 'link_types' (link_type, link_type_id, description, lanes, lane_capacity)␣
↪VALUES('default', 'y', 'Default general link type', 2, 900);

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','link_type', 'Link type name. E.g. arterial, or connector');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','link_type_id', 'Single letter identifying the mode. E.g. a, for␣
↪arterial');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','description', 'Description of the same. E.g. Arterials are streets␣
↪like AequilibraE Avenue');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','lanes', 'Default number of lanes in each direction. E.g. 2');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','lane_capacity', 'Default vehicle capacity per lane. E.g.  900');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','speed', 'Default uncongested speed for this link type, in m/s');
```

links table structure

The links table holds all the links available in the aequilibrae network model regardless of the modes allowed on it.

All information on the fields a_node and b_node correspond to a entries in the node_id field in the nodes table. They are automatically managed with triggers as the user edits the network. Updates that would make either field inconsistent with the corresponding geometry endpoint are rejected.

The **modes** field is a concatenation of all the ids (mode_id) of the models allowed on each link, and map directly to the mode_id field in the **Modes** table. A mode can only be added to a link if it exists in the **Modes** table.

The **link_type** corresponds to the *link_type* field from the *link_types* table. As it is the case for modes, a link_type can only be assigned to a link if it exists in the **link_types** table.

The fields **length**, **node_a** and **node_b** are automatically updated by triggers based in the links' geometries and node positions. Link length is always measured in **meters**.

The table is indexed on **link_id** (its primary key), **node_a** and **node_b**.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
link_id	INTEGER	NO	
a_node	INTEGER	YES	
b_node	INTEGER	YES	
direction	INTEGER	NO	0
distance	NUMERIC	YES	
modes	TEXT	NO	
link_type	TEXT	YES	
name	TEXT	YES	
fixed_cost_ab	NUMERIC	YES	
fixed_cost_ba	NUMERIC	YES	
speed_ab	NUMERIC	YES	
speed_ba	NUMERIC	YES	
travel_time_ab	NUMERIC	YES	
travel_time_ba	NUMERIC	YES	
capacity_ab	NUMERIC	YES	
capacity_ba	NUMERIC	YES	
other_attributes	TEXT	YES	
geometry	LINESTRING	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists links (ogc_fid           INTEGER PRIMARY KEY,
                                   link_id           INTEGER NOT NULL UNIQUE,
                                   a_node            INTEGER,
                                   b_node            INTEGER,
                                   direction         INTEGER NOT NULL DEFAULT 0,
                                   distance          NUMERIC,
```

(continues on next page)

(continued from previous page)

```
                                    modes             TEXT    NOT NULL,
                                    link_type         TEXT    REFERENCES link_
↪types(link_type) ON update RESTRICT ON delete RESTRICT,
                                    'name'            TEXT,
                                    fixed_cost_ab     NUMERIC,
                                    fixed_cost_ba     NUMERIC,
                                    speed_ab          NUMERIC,
                                    speed_ba          NUMERIC,
                                    travel_time_ab    NUMERIC,
                                    travel_time_ba    NUMERIC,
                                    capacity_ab       NUMERIC,
                                    capacity_ba       NUMERIC,
                                    other_attributes  TEXT
                                    CHECK(TYPEOF(link_id) == 'integer')
                                    CHECK(link_id > 0)
                                    CHECK(TYPEOF(a_node) IN ('integer', 'null'))
                                    CHECK(TYPEOF(b_node) IN ('integer', 'null'))
                                    CHECK(TYPEOF(direction) == 'integer')
                                    CHECK(LENGTH(modes)>0)
                                    CHECK(direction IN (-1, 0, 1)));

select AddGeometryColumn( 'links', 'geometry', 4326, 'LINESTRING', 'XY', 1);

CREATE UNIQUE INDEX idx_link ON links (link_id);

SELECT CreateSpatialIndex( 'links' , 'geometry' );

CREATE INDEX idx_link_anode ON links (a_node);

CREATE INDEX idx_link_bnode ON links (b_node);

CREATE INDEX idx_link_modes ON links (modes);

CREATE INDEX idx_link_link_type ON links (link_type);

CREATE INDEX idx_links_a_node_b_node ON links (a_node, b_node);

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','link_id', 'Unique link ID');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','a_node', 'origin node for the link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','b_node', 'destination node for the link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','direction', 'Flow direction allowed on the link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','distance', 'length of the link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','modes', 'modes allowed on the link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','link_type', 'Link type');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
```

(continues on next page)

(continued from previous page)

```
↪'links','name', 'Name of the street/link');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','fixed_cost_*', 'Directional fixed costs (if any). Tolls, for example');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','speed_*', 'Directional speeds (if allowed)');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','capacity_*', 'Directional link capacities (if allowed)');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','travel_time_*', 'Directional free-flow travel time (if allowed)');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'links','other_attributes', 'Other attributes of the link. Preferably in json format
↪');
```

modes table structure

The *modes* table holds the information on all the modes available in the model's network.

The **mode_name** field contains the descriptive name of the field.

The **mode_id** field contains a single letter that identifies the mode.

The **description** field holds the description of the mode.

The **pce** field holds information on Passenger-Car equivalent for assignment. Defaults to **1.0**.

The **vot** field holds information on Value-of-Time for traffic assignment. Defaults to **0.0**.

The **ppv** field holds information on average persons per vehicle. Defaults to **1.0**. **ppv** can assume value 0 for non-travel uses. Attributes follow

Field	Type	NULL allowed	Default Value
mode_name	VARCHAR	NO	
mode_id*	VARCHAR	NO	
description	VARCHAR	YES	
pce	NUMERIC	NO	1.0
vot	NUMERIC	NO	0
ppv	NUMERIC	NO	1.0

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists modes (mode_name   VARCHAR UNIQUE NOT NULL,
                                  mode_id     VARCHAR UNIQUE NOT NULL       PRIMARY␣
↪KEY,
                                  description VARCHAR,
                                  pce         NUMERIC        NOT NULL DEFAULT 1.0,
                                  vot         NUMERIC        NOT NULL DEFAULT 0,
                                  ppv         NUMERIC        NOT NULL DEFAULT 1.0
                                  CHECK(LENGTH(mode_id)==1));

INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('car', 'c', 'All␣
↪motorized vehicles');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('transit', 't', 'Public␣
```

(continues on next page)

(continued from previous page)

```
↪transport vehicles');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('walk', 'w', 'Walking␣
↪links');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('bicycle', 'b', 'Biking␣
↪links');

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','mode_name', 'The more descriptive name of the mode (e.g. Bicycle)');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','mode_id', 'Single letter identifying the mode. E.g. b, for Bicycle');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','description', 'Description of the same. E.g. Bicycles used to be human-
↪powered two-wheeled vehicles');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','pce', 'Passenger-Car equivalent for assignment');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','vot', 'Value-of-Time for traffic assignment of class');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','ppv', 'Average persons per vehicle. (0 for non-travel uses)');
```

nodes table structure

The *nodes* table holds all the network nodes available in AequilibraE model.

The **node_id** field is an identifier of the node.

The **is_centroid** field holds information if the node is a centroid of a network or not. Assumes values 0 or 1. Defaults to **0**.

The **modes** field identifies all modes connected to the node.

The **link_types** field identifies all link types connected to the node.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
node_id	INTEGER	NO	
is_centroid	INTEGER	NO	0
modes	TEXT	YES	
link_types	TEXT	YES	
other_attributes	TEXT	YES	
geometry	POINT	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists nodes (ogc_fid     INTEGER PRIMARY KEY,
                                  node_id     INTEGER UNIQUE NOT NULL,
                                  is_centroid INTEGER        NOT NULL DEFAULT 0,
                                  modes       TEXT,
                                  link_types  TEXT,
                                  other_attributes  TEXT
                                  CHECK(TYPEOF(node_id) == 'integer')
```

(continues on next page)

(continued from previous page)

```
                                    CHECK(TYPEOF(is_centroid) == 'integer')
                                    CHECK(is_centroid>=0)
                                    CHECK(is_centroid<=1));

SELECT AddGeometryColumn( 'nodes', 'geometry', 4326, 'POINT', 'XY', 1);

SELECT CreateSpatialIndex( 'nodes' , 'geometry' );

CREATE INDEX idx_node ON nodes (node_id);

CREATE INDEX idx_node_is_centroid ON nodes (is_centroid);

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'nodes','node_id', 'Unique node ID');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'nodes','is_centroid', 'Flag identifying centroids');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'nodes','modes', 'Modes connected to the node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'nodes','link_types', 'Link types connected to the node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'nodes','other_attributes', 'Other attributes of the node. Preferably in json format
↪');
```

periods table structure

The periods table holds the time periods and their period_id. Default entry with id 1 is the entire day. Attributes follow

Field	Type	NULL allowed	Default Value
period_id	INTEGER	NO	
period_start	INTEGER	NO	
period_end	INTEGER	NO	
period_description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists periods (period_id               INTEGER UNIQUE NOT NULL,
                                    period_start            INTEGER NOT NULL,
                                    period_end              INTEGER NOT NULL,
                                    period_description      TEXT
                                    CHECK(TYPEOF(period_id) == 'integer')
                                    CHECK(TYPEOF(period_start) == 'integer')
                                    CHECK(TYPEOF(period_end) == 'integer'));

INSERT INTO periods (period_id, period_start, period_end, period_description)␣
↪VALUES(1, 0, 86400, 'Default time period, whole day');

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'periods','period_id', 'ID of the time period');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'periods','period_start', 'Start of the time period');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'periods','period_end', 'End of the time period');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'periods','period_description', 'Optional description of the time period');
```

scenarios table structure

Attributes follow

Field	Type	NULL allowed	Default Value
scenario_name	TEXT	NO	
description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS scenarios (scenario_name TEXT UNIQUE NOT NULL, description␣
↪TEXT);

INSERT INTO 'scenarios' (scenario_name, description) VALUES('root', 'The default, and␣
↪root, scenario for an AequilbraE project. The name "root" is treated as a special␣
↪case.');

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'scenarios', 'scenario_name', 'The scenario folder name.');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'scenarios', 'description', 'Description of the scenario');
```

transit graph configs table structure

The *transit_graph_configs* table holds configuration parameters for a TransitGraph of a particular *period_id* Attributes follow

Field	Type	NULL allowed	Default Value
period_id*	INTEGER	NO	
config	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists transit_graph_configs (period_id INTEGER UNIQUE NOT NULL␣
↪PRIMARY KEY REFERENCES periods(period_id),
                                                  config    TEXT);

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
```

(continues on next page)

(continued from previous page)

```
↪'transit_graph_configs','period_id', 'The period this config is associated with.');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'transit_graph_configs','mode_id', 'JSON string containing the configuration␣
↪parameters.');
```

turn restrictions table structure

The *turn_restrictions* table holds information about turn restrictions and penalties for the network.

The **restriction_id** field is the unique identifier for the turn restriction

The **from_node** field is the node at the start of the incoming movement leg.

The **via_node** field is the node where the turn occurs.

The **to_node** field is the node at the end of the outgoing movement leg.

The **geometry** field is a LINESTRING built from three points representing the turn movement sequence (incoming leg, via node, outgoing leg).

The **penalty** field is the turn penalty in the same time unit as the graph cost field. NULL indicates a prohibited turn. Positive infinity is also accepted at the API layer and is normalised to NULL before storage. Negative values are not allowed.

The **modes** field is a concatenation of mode IDs for which this restriction applies.

Field	Type	NULL allowed	Default Value
restriction_id*	INTEGER	NO	
from_node	INTEGER	NO	
via_node	INTEGER	NO	
to_node	INTEGER	NO	
penalty	REAL	YES	
modes	TEXT	NO	
geometry	LINESTRING	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS turn_restrictions (
    restriction_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    from_node      INTEGER NOT NULL,
    via_node       INTEGER NOT NULL,
    to_node        INTEGER NOT NULL,
    penalty        REAL,
    modes          TEXT    NOT NULL,
    FOREIGN KEY(from_node) REFERENCES nodes(node_id) ON UPDATE CASCADE,
    FOREIGN KEY(via_node) REFERENCES nodes(node_id) ON UPDATE CASCADE,
    FOREIGN KEY(to_node) REFERENCES nodes(node_id) ON UPDATE CASCADE,
    CHECK(from_node != via_node),
    CHECK(via_node != to_node),
    CHECK(penalty IS NULL OR penalty >= 0),
    CHECK(LENGTH(modes) > 0),
    UNIQUE(from_node, via_node, to_node, modes)
```

(continues on next page)

(continued from previous page)

```
);

select AddGeometryColumn( 'turn_restrictions', 'geometry', 4326, 'LINESTRING', 'XY',␣
↪0);

SELECT CreateSpatialIndex( 'turn_restrictions' , 'geometry' );

CREATE INDEX IF NOT EXISTS idx_turn_restrictions_from_node ON turn_restrictions(from_
↪node);

CREATE INDEX IF NOT EXISTS idx_turn_restrictions_to_node ON turn_restrictions(to_
↪node);

CREATE INDEX IF NOT EXISTS idx_turn_restrictions_modes ON turn_restrictions(modes);

CREATE INDEX IF NOT EXISTS idx_turn_restrictions_via_node ON turn_restrictions(via_
↪node);
```

zones table structure

The *zones* table holds information on the Traffic Analysis Zones (TAZs) in AequilibraE's model.

The **zone_id** field identifies the zone.

The **area** field corresponds to the area of the zone in **km2**. TAZs' area is automatically updated by triggers.

The **name** fields allows one to identity the zone using a name or any other description.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
zone_id	INTEGER	NO	
area	NUMERIC	YES	
name	TEXT	YES	
geometry	MULTIPOLYGON	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE 'zones' (ogc_fid    INTEGER PRIMARY KEY,
                      zone_id    INTEGER UNIQUE NOT NULL,
                      area       NUMERIC,
                      "name"     TEXT);

SELECT AddGeometryColumn( 'zones', 'geometry', 4326, 'MULTIPOLYGON', 'XY', 1);
CREATE UNIQUE INDEX idx_zone ON zones (zone_id);
SELECT CreateSpatialIndex( 'zones' , 'geometry' );
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','zone_id', 'Unique node ID');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','area', 'Area of the zone in km2');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','name', 'Name of the zone, if any');
```

2.2.3 Results database SQL data model

The data model presented in this section describes the metadata table in `results_database.sqlite`. The same database stores the result data tables referenced by its `table_name` column.

Results tables

results table structure

The *results* table holds the metadata for results stored in *results_database.sqlite*.

The **scenario** field describes which scenario this record belongs to.

The **year** field describes which year of the scenario was used.

The **table_name** field presents the actual name of the result table in *results_database.sqlite*.

The **procedure** field holds the name the the procedure that generated the result (e.g.: Traffic Assignment).

The **procedure_id** field holds an unique UUID identifier for this procedure, which is created at runtime.

The **procedure_report** field holds the output of the complete procedure report.

The **timestamp** field holds the information when the procedure was executed.

The **description** field holds the user-provided description of the result.

Field	Type	NULL allowed	Default Value
scenario	TEXT	YES	
year	TEXT	YES	
table_name*	TEXT	NO	
reference_table	TEXT	YES	
procedure	TEXT	NO	
procedure_id	TEXT	NO	
procedure_report	TEXT	NO	
timestamp	DATETIME	YES	current_timestamp
description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
create TABLE if not exists results (scenario         TEXT,
                                    year             TEXT,
                                    table_name       TEXT     NOT NULL PRIMARY KEY,
                                    reference_table  TEXT,
                                    procedure        TEXT     NOT NULL,
                                    procedure_id     TEXT     NOT NULL,
                                    procedure_report TEXT     NOT NULL,
                                    timestamp        DATETIME DEFAULT current_
↪timestamp,
                                    description      TEXT);
```

2.2.4 Parameters YAML File

The parameter file holds the parameters information for a certain portion of the software.

Run

The run section of the parameter file defines the default keyword arguments for the callable objects in the *Run module*. Each subsection names a callable symbol within the `run/__init__.py` module, if the symbol does not exist a `RuntimeError` will be raised when `project.run` is accessed. The arguments are applied via `functools.partial` and replace the objects within the module.

```yaml
run:
  example_function_with_kwargs:
    arg1: "parameters.yml argument"
```

This can be used to define model entry points or functions that should be stored adjacent to the model itself.

Assignment

The assignment section of the parameter file is the smallest one, and it contains only the convergence criteria for assignment in terms of the maximum number of iterations and target Relative Gap.

```yaml
assignment:
  equilibrium:
    rgap: 1.0e-5
    maximum_iterations: 500
```

Although these parameters are required to exist in the parameters file, one can override them during the assignment, as detailed in *Convergence criteria*.

Distribution

The distribution section of the parameter file is also fairly short, as it contains only the parameters for number of maximum iterations, convergence level and maximum trip length to be applied in Iterative Proportional Fitting and synthetic gravity models, as shown below.

```yaml
distribution:
  gravity:
    max error: 0.0001
    max iterations: 100
    max trip length: -1
  ipf:
    balancing tolerance: 0.001
    convergence level: 0.0001
    max iterations: 5000
```

Network

There are four groups of parameters under the network section: *links*, *nodes*, *OSM*, and *GMNS*. The first are basically responsible for the design of the network to be created in case a new project/network is to bre created from scratch, and for now each one of these groups contains only a single group of parameters called *fields*.

Link Fields

The section for link fields are divided into *one-way* fields and *two-way* fields, where the two-way fields will be created by appending *_ab* and *_ba* to the end of each field's name.

There are 5 fields which cannot be changed, as they are mandatory fields for an AequilibraE network, and they are **link_id**, **a_node**, **b_node**, **direction**, **distance** and **modes**. The field **geometry** is also default, but it is not listed in the parameter file due to its distinct nature.

The list of fields required in the network are enumerated as an array under either *one-way* or *two-way* in the parameter file, and each field is a dictionary/hash that has the field's name as the only key and under which there is a field for *description* and a field for *data type*. The data types available are those that exist within the SQLite specification . We recommend limiting yourself to the use of **integer**, **numeric** and **varchar**.

```
network:
  links:
    fields:
      one-way:
        - link_id:
            description: Link ID. THIS FIELD CANNOT BE CHANGED
            type: integer
```

Note

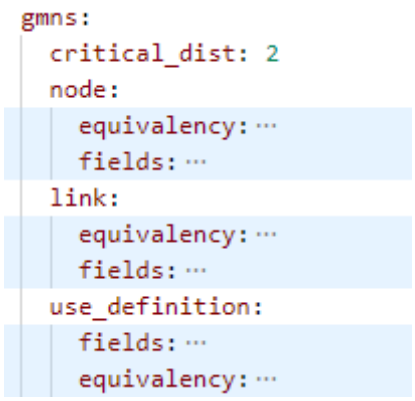
The OSM/Overture importers (`project.tables.importer.osm()` and `project.tables.importer.overture()`) do not use the link-field parameter specification to decide which tags to read. They parse OSM/Overture tags internally and store any attribute that does not map to a real table column in the `other_attributes` JSON column. Directional handling is also built in: `maxspeed` is applied to both directions, while a total `lanes` count for a two-way link is split across directions (the remainder of an odd count going to the AB direction). See *Importing from OpenStreetMap* for details.

Node fields

The specification for node fields is similar to the one for link fields, with the key difference that it does not make sense to have fields for one or two directions.

GMNS

The **GMNS** group of parameters has four specifications: **critical_dist**, **link**, **node**, and **use_definition**.



critical_dist is a numeric threshold for the distance.

Under the keys **links**, **nodes**, and **use_definition** there are the fields *equivalency* and *fields*. They represent the equivalency between GMNS and AequilibraE data fields and data types for each field.

System

The system section of the parameters file holds information on the number of threads used in multi-threaded processes, logging and temp folders and whether we should be saving information to a log file at all, as exemplified below.

```
system:
  cpus: 12
  default_directory: C:\Users\pedro\Research\sourcecode\drt
  driving side: right
  logging: true
  temp directory: /temp
  logging_directory: /temp
  spatialite_path: C:\Users\pedro\Documents\mod_spatialite-NG-win-amd64
```

The number of CPUs have a special behaviour defined, as follows:

- **cpus<0** : The system will use the total number logical processors **MINUS** the absolute value of **cpus**
- **cpus=0** : The system will use the total number logical processors available
- **cpus>0**
  [The system will use exactly **cpus** for computation, limited to] the total number logical processors available

A few of these parameters, however, are targeted at its QGIS plugin, which is the case of the *driving side* and *default_directory* parameters.

Open Street Maps

The OSM section of the parameter file configures the Overpass download used by `project.tables.importer.osm()`. It is relevant only when one plans to download a substantial amount of data from an Overpass API, in which case it is recommended to deploy a local Overpass server and point `overpass_endpoint` at it.

The available settings are:

- `max_query_area_size`: largest Overpass query part in square metres (default 100,000,000, or 100 km^2). Larger and disconnected boundaries are split without changing their requested coverage;
- `overpass_endpoint`: base URL of the Overpass API to query;
- `nominatim_endpoint`: base URL of the Nominatim server used to resolve `place_name` lookups;
- `accept_language`: language tag requested for tag values such as names;
- `timeout`: how long (in seconds) to wait for an Overpass response before giving up;
- `overpass_rate_limit`: whether osmnx should respect the server's rate-limit status (default `true`). Set to `false` for a self-hosted Overpass server: those report an unlimited rate limit in a format that makes osmnx wait forever for a slot that is already free.

Endpoint, language, timeout and rate-limit values are applied through `osmnx.settings` at import time. AequilibraE subdivides large query areas before passing them to `osmnx` and merges the resulting graphs before import.

See also

- ***aequilibrae.project.Parameters()***
  Class documentation

2.2.5 Public Transport Database

AequilibraE's transit module has been updated in version 0.9.0 and more details on the **public_transport.sqlite** are discussed on a nearly *per-table* basis below. We recommend understanding the role of each table before setting an AequilibraE model you intend to use.

Create the public transport database with `project.scenario.create_transit_database()` if it does not already exist, then access its gateway through `project.transit`. Creating the database can take a little while.

See also

- ***aequilibrae.Project.transit***
  Transit gateway; use `project.transit.create_graph()` to obtain a graph builder

In the following sections, we'll dive deep into the tables existing in the public transport database. Please notice that some tables are homonyms to the ones existing in the **project_database.sqlite**, but its contents are related to the public transport graph building and assignment processes.

2.2.6 Public Transport SQL Data model

The data model presented in this section pertains only to the structure of AequilibraE's 'public_transport' database and general information about the usefulness of specific fields, especially on the interdependency between tables.

Conventions

A few conventions have been adopted in the definition of the data model and some are listed below:

- Geometry field is always called **geometry**
- Projection is 4326 (WGS84)
- Tables are all in all lower case

Project tables

agencies table structure

The *agencies* table holds information about the Public Transport agencies within the GTFS data. This table information comes from GTFS file *agency.txt*. You can check out more information on agencies here.

agency_id identifies the agency for the specified route

agency contains the full name of the transit agency

feed_date indicates the date for which the GTFS feed is being imported

service_date indicates the date for the indicated route scheduling

description_field provides useful description of a transit agency

Field	Type	NULL allowed	Default Value
agency_id*	INTEGER	NO	
agency	TEXT	NO	
feed_date	TEXT	YES	
service_date	TEXT	YES	
description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
create TABLE IF NOT EXISTS agencies (
     agency_id     INTEGER  NOT NULL PRIMARY KEY AUTOINCREMENT,
     agency        TEXT     NOT NULL,
     feed_date     TEXT,
     service_date  TEXT,
     description   TEXT
);

create UNIQUE INDEX IF NOT EXISTS transit_operators_id ON agencies (agency_id);
```

attributes documentation table structure

The *attributes_documentation* table holds information about attributes in the tables links, link_types, modes, nodes, and zones.

By default, these attributes are all documented, but further attributes can be added into the table.

The **name_table** field holds the name of the table that has the attribute

The **attribute** field holds the name of the attribute

The **description** field holds the description of the attribute

It is possible to have one attribute with the same name in two different tables. However, one cannot have two attributes with the same name within the same table.

Field	Type	NULL allowed	Default Value
name_table	TEXT	NO	
attribute	TEXT	NO	
description	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists attributes_documentation (name_table  TEXT NOT NULL,
                                                      attribute   TEXT NOT NULL,
                                                      description TEXT,
                                                      UNIQUE (name_table, attribute)
                                                      );

CREATE INDEX IF NOT EXISTS idx_attributes ON attributes_documentation (name_table,␣
↪attribute);
```

fare attributes table structure

The *fare_attributes* table holds information about the fare values. This table information comes from the GTFS file *fare_attributes.txt*. Given that this file is optional in GTFS, it can be empty. You can check out more information on fare attributes here.

fare_id identifies a fare class

fare describes a fare class

agency_id identifies a relevant agency for a fare.

price specifies the fare price

currency_code specifies the currency used to pay the fare

payment_method indicates when the fare must be paid.

transfer indicates the number of transfers permitted on the fare

transfer_duration indicates the length of time in seconds before a transfer expires.

Field	Type	NULL allowed	Default Value
fare_id*	INTEGER	NO	
fare	TEXT	NO	
agency_id	INTEGER	NO	
price	REAL	YES	
currency	TEXT	YES	
payment_method	INTEGER	YES	
transfer	INTEGER	YES	
transfer_duration	REAL	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
create TABLE IF NOT EXISTS fare_attributes (
     fare_id           INTEGER  NOT NULL PRIMARY KEY AUTOINCREMENT,
     fare              TEXT     NOT NULL,
     agency_id         INTEGER  NOT NULL,
     price             REAL,
     currency          TEXT,
     payment_method    INTEGER,
     transfer          INTEGER,
     transfer_duration REAL,
     FOREIGN KEY(agency_id) REFERENCES agencies(agency_id) deferrable initially␣
↪deferred
);

CREATE UNIQUE INDEX IF NOT EXISTS fare_transfer_uniqueness ON fare_attributes (fare_
↪id, transfer);
```

fare rules table structure

The *fare_rules* table holds information about the fare values. This table information comes from the GTFS file *fare_rules.txt*. Given that this file is optional in GTFS, it can be empty.

The **fare_id** identifies a fare class

The **route_id** identifies a route associated with the fare class

The **origin** field identifies the transit fare zone for origin

The **destination** field identifies the transit fare zone for destination

The **contains** field identifies the zones that a rider will enter while using a given fare class.

Field	Type	NULL allowed	Default Value
fare_id	INTEGER	NO	
route_id	INTEGER	YES	
origin	TEXT	YES	
destination	TEXT	YES	
contains	TEXT	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
create TABLE IF NOT EXISTS fare_rules (
     fare_id     INTEGER  NOT NULL,
     route_id    INTEGER,
     origin      TEXT,
     destination TEXT,
     contains    TEXT,
     FOREIGN KEY(fare_id) REFERENCES fare_attributes(fare_id) deferrable initially␣
↪deferred
);
```

fare zones table structure

The *fare_zones* table hold information on the transit fare zones and the transit agencies that operate in it.

transit_fare_zone identifies the transit fare zones

agency_id identifies the agency/agencies for the specified fare zone

Field	Type	NULL allowed	Default Value
transit_fare_zone	TEXT	NO	
agency_id	INTEGER	NO	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS fare_zones (
     transit_fare_zone    TEXT    NOT NULL,
     agency_id            INTEGER NOT NULL,
     FOREIGN KEY(agency_id) REFERENCES agencies(agency_id) deferrable initially␣
↪deferred
);
```

link types table structure

The *link_types* table holds information about the available link types in the network.

The **link_type** field corresponds to the link type, and it is the table's primary key

The **link_type_id** field presents the identification of the link type

The **description** field holds the description of the link type

The **lanes** field presents the number or lanes for the link type

The **lane_capacity** field presents the number of lanes for the link type

The **speed** field holds information about the speed in the link type Attributes follow

Field	Type	NULL allowed	Default Value
link_type*	VARCHAR	NO	
link_type_id	VARCHAR	NO	
description	VARCHAR	YES	
lanes	NUMERIC	YES	
lane_capacity	NUMERIC	YES	
speed	NUMERIC	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists link_types (link_type     VARCHAR UNIQUE NOT NULL PRIMARY␣
↪KEY,
                                        link_type_id  VARCHAR UNIQUE NOT NULL,
                                        description   VARCHAR,
```

(continues on next page)

(continued from previous page)

```
                                    lanes         NUMERIC,
                                    lane_capacity NUMERIC,
                                    speed         NUMERIC
                                    CHECK(LENGTH(link_type_id) == 1));

INSERT INTO 'link_types' (link_type, link_type_id, description, lanes, lane_capacity)␣
↪VALUES('centroid_connector', 'z', 'VIRTUAL centroid connectors only', 10, 10000);

INSERT INTO 'link_types' (link_type, link_type_id, description, lanes, lane_capacity)␣
↪VALUES('default', 'y', 'Default general link type', 2, 900);

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','link_type', 'Link type name. E.g. arterial, or connector');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','link_type_id', 'Single letter identifying the mode. E.g. a, for␣
↪arterial');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','description', 'Description of the same. E.g. Arterials are streets␣
↪like AequilibraE Avenue');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','lanes', 'Default number of lanes in each direction. E.g. 2');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','lane_capacity', 'Default vehicle capacity per lane. E.g.  900');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'link_types','speed', 'Speed for the link to be used in assignment, in m/s');
```

links table structure

The links table holds all the links available in the aequilibrae network model regardless of the modes allowed on it.

All information on the fields a_node and b_node correspond to a entries in the node_id field in the nodes table. They are automatically managed with triggers as the user edits the network, but they are not protected by manual editing, which would break the network if it were to happen.

The **modes** field is a concatenation of all the ids (mode_id) of the models allowed on each link, and map directly to the mode_id field in the **Modes** table. A mode can only be added to a link if it exists in the **Modes** table.

The **link_type** corresponds to the *link_type* field from the *link_types* table. As it is the case for modes, a link_type can only be assigned to a link if it exists in the **link_types** table.

The fields **length**, **node_a** and **node_b** are automatically updated by triggers based in the links' geometries and node positions. Link length is always measured in **meters**.

The table is indexed on **link_id** (its primary key), **node_a** and **node_b**.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
link_id	INTEGER	NO	
period_id	INTEGER	NO	
a_node	INTEGER	YES	
b_node	INTEGER	YES	
direction	INTEGER	NO	0
distance	NUMERIC	YES	
modes	TEXT	NO	
link_type	TEXT	YES	
line_id	TEXT	YES	
stop_id	TEXT	YES	
line_seg_idx	INTEGER	YES	
trav_time	NUMERIC	NO	
freq	NUMERIC	NO	
o_line_id	TEXT	YES	
d_line_id	TEXT	YES	
transfer_id	TEXT	YES	
geometry	LINESTRING	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists links (ogc_fid          INTEGER PRIMARY KEY,
                                  link_id          INTEGER NOT NULL,
                                  period_id        INTEGER NOT NULL,
                                  a_node           INTEGER,
                                  b_node           INTEGER,
                                  direction        INTEGER NOT NULL DEFAULT 0,
                                  distance         NUMERIC,
                                  modes            TEXT    NOT NULL,
                                  link_type        TEXT    REFERENCES link_types(link_
↪type) ON update RESTRICT ON delete RESTRICT,
                                  line_id          TEXT,
                                  stop_id          TEXT    REFERENCES stops(stop_id)␣
↪ON update RESTRICT ON delete RESTRICT,
                                  line_seg_idx     INTEGER,
                                  trav_time        NUMERIC  NOT NULL,
                                  freq             NUMERIC  NOT NULL,
                                  o_line_id        TEXT,
                                  d_line_id        TEXT,
                                  transfer_id      TEXT,
                                  CHECK(TYPEOF(link_id) == 'integer')
                                  UNIQUE(link_id, period_id) ON CONFLICT ABORT
                                  CHECK(link_id > 0)
                                  CHECK(TYPEOF(a_node) == 'integer')
                                  CHECK(TYPEOF(b_node) == 'integer')
                                  CHECK(TYPEOF(direction) == 'integer')
                                  CHECK(LENGTH(modes)>0)
                                  CHECK(LENGTH(direction)==1));
```

(continues on next page)

(continued from previous page)

```
select AddGeometryColumn( 'links', 'geometry', 4326, 'LINESTRING', 'XY', 1);

CREATE UNIQUE INDEX idx_link ON links (link_id, period_id);

CREATE INDEX idx_period_links ON links (period_id);

SELECT CreateSpatialIndex( 'links' , 'geometry' );

CREATE INDEX idx_link_anode ON links (a_node);

CREATE INDEX idx_link_bnode ON links (b_node);

CREATE INDEX idx_link_modes ON links (modes);

CREATE INDEX idx_link_link_type ON links (link_type);

CREATE INDEX idx_links_a_node_b_node ON links (a_node, b_node);

INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','link_id', 'Unique link ID');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','a_node', 'origin node for the link');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','b_node', 'destination node for the link');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','direction', 'Flow direction allowed on the link');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','distance', 'length of the link');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','modes', 'modes allowed on the link');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','link_type', 'Link type');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','line_id', 'ID of the line the link belongs to');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','stop_id', 'ID of the stop the link belongs to ');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','line_seg_idx', 'Line segment index');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','trav_time', 'Travel time');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','freq', 'Frequency of link traversal');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','*_line_id', 'Origin/Destination line ID for transfer␣
↪links');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('links','transfer_id', 'Transfer link ID');
```

modes table structure

The *modes* table holds the information on all the modes available in the model's network.

The **mode_name** field contains the descriptive name of the field.

The **mode_id** field contains a single letter that identifies the mode.

The **description** field holds the description of the mode.

The **pce** field holds information on Passenger-Car equivalent for assignment. Defaults to **1.0**.

The **vot** field holds information on Value-of-Time for traffic assignment. Defaults to **0.0**.

The **ppv** field holds information on average persons per vehicle. Defaults to **1.0**. **ppv** can assume value 0 for non-travel uses. Attributes follow

Field	Type	NULL allowed	Default Value
mode_name	VARCHAR	NO	
mode_id*	VARCHAR	NO	
description	VARCHAR	YES	
pce	NUMERIC	NO	1.0
vot	NUMERIC	NO	0
ppv	NUMERIC	NO	1.0

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists modes (mode_name   VARCHAR UNIQUE NOT NULL,
                                  mode_id     VARCHAR UNIQUE NOT NULL       PRIMARY␣
↪KEY,
                                  description VARCHAR,
                                  pce         NUMERIC        NOT NULL DEFAULT 1.0,
                                  vot         NUMERIC        NOT NULL DEFAULT 0,
                                  ppv         NUMERIC        NOT NULL DEFAULT 1.0
                                  CHECK(LENGTH(mode_id)==1));

INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('car', 'c', 'All␣
↪motorized vehicles');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('transit', 't', 'Public␣
↪transport vehicles');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('walk', 'w', 'Walking␣
↪links');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('bicycle', 'b', 'Biking␣
↪links');

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','mode_name', 'The more descriptive name of the mode (e.g. Bicycle)');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','mode_id', 'Single letter identifying the mode. E.g. b, for Bicycle');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','description', 'Description of the same. E.g. Bicycles used to be human-
↪powered two-wheeled vehicles');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','pce', 'Passenger-Car equivalent for assignment');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','vot', 'Value-of-Time for traffic assignment of class');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'modes','ppv', 'Average persons per vehicle. (0 for non-travel uses)');
```

node types table structure

The *node_types* table holds information about the available node types in the network.

The **node_type** field corresponds to the node type, and it is the table's primary key

The **node_type_id** field presents the identification of the node type

The **description** field holds the description of the node type

Attributes follow

Field	Type	NULL allowed	Default Value
node_type*	VARCHAR	NO	
node_type_id	VARCHAR	NO	
description	VARCHAR	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE  if not exists node_types (node_type     VARCHAR UNIQUE NOT NULL PRIMARY␣
↪KEY,
                                        node_type_id  VARCHAR UNIQUE NOT NULL,
                                        description   VARCHAR);

INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('default', 'y',
↪ 'Default general node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('od', 'n',
↪'Origin/Destination node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('origin', 'o',
↪'Origin node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('destination',
↪'d', 'Destination node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('stop', 's',
↪'Stop node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('alighting', 'a
↪', 'Alighting node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('boarding', 'b
↪', 'Boarding node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('walking', 'w',
↪ 'Walking node type');

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'node_types','node_type', 'Node type name. E.g stop or boarding');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'node_types','node_type_id', 'Single letter identifying the mode. E.g. a, for␣
↪alighting');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'node_types','description', 'Description of the same. E.g. Stop nodes connect ODs␣
↪and walking nodes to boarding and alighting nodes via boarding and alighting links.
↪');
```

nodes table structure

The *nodes* table holds all the network nodes available in AequilibraE transit model.

The **node_id** field is an identifier of the node.

The **is_centroid** field holds information if the node is a centroid of a network or not. Assumes values 0 or 1. Defaults to **0**.

The **stop_id** field indicates which stop this node belongs too. This field is TEXT as it might encode a street name or such.

The **line_id** field indicates which line this node belongs too. This field is TEXT as it might encode a street name or such.

The **line_seg_idx** field indexes the segment of line **line_id**. Zero based.

The **modes** field identifies all modes connected to the node.

The **link_type** field identifies all link types connected to the node.

The **node_type** field identifies the types of this node.

The **taz_id** field is an identifier for the transit assignment zone this node belongs to.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
node_id	INTEGER	NO	
period_id	INTEGER	NO	
is_centroid	INTEGER	NO	0
stop_id	TEXT	YES	
line_id	TEXT	YES	
line_seg_idx	INTEGER	YES	
modes	TEXT	YES	
link_types	TEXT	YES	
node_type	TEXT	YES	
taz_id	INTEGER	YES	
geometry	POINT	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists nodes (ogc_fid      INTEGER PRIMARY KEY,
                                  node_id      INTEGER NOT NULL,
                                  period_id    INTEGER NOT NULL,
                                  is_centroid  INTEGER        NOT NULL DEFAULT 0,
                                  stop_id      TEXT,
                                  line_id      TEXT,
                                  line_seg_idx INTEGER,
                                  modes        TEXT,
                                  link_types   TEXT,
```

(continues on next page)

(continued from previous page)

```
                                    node_type    TEXT,
                                    taz_id       INTEGER,
                                    CHECK(TYPEOF(taz_id) == 'integer')
                                    CHECK(TYPEOF(node_id) == 'integer')
                                    CHECK(TYPEOF(is_centroid) == 'integer')
                                    CHECK(is_centroid>=0)
                                    CHECK(is_centroid<=1));

SELECT AddGeometryColumn( 'nodes', 'geometry', 4326, 'POINT', 'XY', 1);

SELECT CreateSpatialIndex( 'nodes' , 'geometry' );

CREATE INDEX idx_node ON nodes (node_id, period_id);

CREATE INDEX idx_period_nodes ON nodes (period_id);

CREATE INDEX idx_node_is_centroid ON nodes (is_centroid);

INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','node_id', 'Unique node ID');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','is_centroid', 'Flag identifying centroids');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','stop_id', 'ID of the Stop this node belongs to');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','line_id', 'ID of the Line this node belongs to');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','line_seg_idx', 'Index of the line segment this node␣
↪belongs to');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','modes', 'Modes connected to the node');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','link_types', 'Link types connected to the node');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','node_type', 'Node types of this node');
INSERT OR REPLACE INTO 'attributes_documentation' (name_table, attribute,␣
↪description) VALUES('nodes','taz_id', 'Transit assignment zone id');
```

pattern mapping table structure

The *pattern_mapping* table holds information on the stop pattern for each route.

pattern_id is an unique pattern for the route

seq identifies the sequence of the stops for a trip

link identifies the *link_id* in the links table that corresponds to the pattern matching

dir indicates the direction of travel for a trip

Field	Type	NULL allowed	Default Value
pattern_id*	INTEGER	NO	
seq	INTEGER	NO	
link	INTEGER	NO	
dir	INTEGER	NO	
geometry	LINESTRING	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS pattern_mapping (
     pattern_id  INTEGER    NOT NULL,
     seq         INTEGER    NOT NULL,
     link        INTEGER    NOT NULL,
     dir         INTEGER    NOT NULL,
     PRIMARY KEY(pattern_id, "seq"),
     FOREIGN KEY(pattern_id) REFERENCES routes (pattern_id) deferrable initially␣
↪deferred
);

SELECT AddGeometryColumn( 'pattern_mapping', 'geometry', 4326, 'LINESTRING', 'XY');

SELECT CreateSpatialIndex( 'pattern_mapping' , 'geometry' );
```

route links table structure

The *route_links* table holds information on the links of a route.

transit_link identifies the GTFS transit links for the route

pattern_id is an unique pattern for the route

seq identifies the sequence of the stops for a trip

from_stop identifies the stop the vehicle is departing

to_stop identifies the next stop the vehicle is going to arrive

distance identifies the distance (in meters) the vehicle travel between the stops

Field	Type	NULL allowed	Default Value
transit_link	INTEGER	NO	
pattern_id	INTEGER	NO	
seq	INTEGER	NO	
from_stop	INTEGER	NO	
to_stop	INTEGER	NO	
distance	INTEGER	NO	
geometry	LINESTRING	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS route_links (
     transit_link            INTEGER    NOT NULL,
     pattern_id              INTEGER    NOT NULL,
     seq                     INTEGER    NOT NULL,
     from_stop               INTEGER    NOT NULL,
     to_stop                 INTEGER    NOT NULL,
     distance                INTEGER    NOT NULL,
     FOREIGN KEY(pattern_id) REFERENCES "routes"(pattern_id) deferrable initially␣
↪deferred,
     FOREIGN KEY(from_stop)  REFERENCES "stops"(stop_id) deferrable initially deferred
     FOREIGN KEY(to_stop)    REFERENCES "stops"(stop_id) deferrable initially deferred
);

create UNIQUE INDEX IF NOT EXISTS route_links_stop_id ON route_links (pattern_id,␣
↪transit_link);

select AddGeometryColumn( 'route_links', 'geometry', 4326, 'LINESTRING', 'XY');

select CreateSpatialIndex( 'route_links' , 'geometry' );
```

routes table structure

The *routes* table holds information on the available transit routes for a specific day. This table information comes from the GTFS file *routes.txt*. You can find more information about the routes table here.

pattern_id is an unique pattern for the route

route_id identifies a route

route identifies the name of a route

agency_id identifies the agency for the specified route

shortname identifies the short name of a route

longname identifies the long name of a route

description provides useful description of a route

route_type indicates the type of transportation used on a route

pce indicates the passenger car equivalent for transportation used on a route

seated_capacity indicates the seated capacity of a route

total_capacity indicates the total capacity of a route

Field	Type	NULL allowed	Default Value
pattern_id*	INTEGER	NO	
route_id	INTEGER	NO	
route	TEXT	NO	
agency_id	INTEGER	NO	
shortname	TEXT	YES	
longname	TEXT	YES	
description	TEXT	YES	
route_type	INTEGER	NO	
pce	NUMERIC	NO	2.0
seated_capacity	INTEGER	YES	
total_capacity	INTEGER	YES	
geometry	MULTILINESTRING	YES	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS routes (
    pattern_id      INTEGER  NOT NULL PRIMARY KEY AUTOINCREMENT,
    route_id        INTEGER  NOT NULL,
    route           TEXT     NOT NULL,
    agency_id       INTEGER  NOT NULL,
    shortname       TEXT,
    longname        TEXT,
    description     TEXT,
    route_type      INTEGER  NOT NULL,
    pce             NUMERIC  NOT NULL DEFAULT 2.0,
    seated_capacity INTEGER,
    total_capacity  INTEGER,
    FOREIGN KEY(agency_id) REFERENCES agencies(agency_id) deferrable initially␣
↪deferred
);

select AddGeometryColumn( 'routes', 'geometry', 4326, 'MULTILINESTRING', 'XY');

select CreateSpatialIndex( 'routes' , 'geometry' );
```

stop connectors table structure

The *stops_connectors* table holds information on the connection of the GTFS network with the real network.

id_from identifies the network link the vehicle departs

id_to identifies the network link th vehicle is heading to

conn_type identifies the type of connection used to connect the links

traversal_time represents the time spent crossing the link

penalty_cost identifies the penalty in the connection

Field	Type	NULL allowed	Default Value
id_from	INTEGER	NO	
id_to	INTEGER	NO	
conn_type	INTEGER	NO	
traversal_time	INTEGER	NO	
penalty_cost	INTEGER	NO	
geometry	LINESTRING	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS stop_connectors (
     id_from        INTEGER  NOT NULL,
     id_to          INTEGER  NOT NULL,
     traversal_time INTEGER  NOT NULL,
     penalty_cost   INTEGER  NOT NULL);

SELECT AddGeometryColumn('stop_connectors', 'geometry', 4326, 'LINESTRING', 'XY', 1);

SELECT CreateSpatialIndex('stop_connectors' , 'geometry');

CREATE INDEX IF NOT EXISTS stop_connectors_id_from ON stop_connectors (id_from);

CREATE INDEX IF NOT EXISTS stop_connectors_id_to ON stop_connectors (id_to);
```

stops table structure

The *stops* table holds information on the stops where vehicles pick up or drop off riders. This table information comes from the GTFS file *stops.txt*. You can find more information about the stops table here.

stop_id is an unique identifier for a stop

stop identifies a stop, station, or station entrance

agency_id identifies the agency for the specified route

link identifies the *link_id* in the links table that corresponds to the pattern matching

dir indicates the direction of travel for a trip

name identifies the name of a stop

parent_station defines hierarchy between different locations defined in *stops.txt*.

description provides useful description of the stop location

street identifies the address of a stop

zone_id identifies the model zone for a stop

transit_fare_zone identifies the transit fare zone for a stop

route_type indicates the type of transportation used on a route

Field	Type	NULL allowed	Default Value
stop_id*	TEXT	YES	
stop	TEXT	NO	
agency_id	INTEGER	NO	
link	INTEGER	YES	
dir	INTEGER	YES	
name	TEXT	YES	
parent_station	TEXT	YES	
description	TEXT	YES	
street	TEXT	YES	
zone_id	INTEGER	YES	
transit_fare_zone	TEXT	YES	
route_type	INTEGER	NO	-1
geometry	POINT	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS stops (
     stop_id           TEXT    PRIMARY KEY,
     stop              TEXT    NOT NULL,
     agency_id         INTEGER NOT NULL,
     link              INTEGER,
     dir               INTEGER,
     name              TEXT,
     parent_station    TEXT,
     description       TEXT,
     street            TEXT,
     zone_id           INTEGER,
     transit_fare_zone TEXT,
     route_type        INTEGER NOT NULL DEFAULT -1,
     FOREIGN KEY(agency_id) REFERENCES agencies(agency_id)
);

create INDEX IF NOT EXISTS stops_stop_id ON stops (stop_id);

select AddGeometryColumn( 'stops', 'geometry', 4326, 'POINT', 'XY', 1);

select CreateSpatialIndex( 'stops' , 'geometry' );
```

trigger settings table structure

This table intends to allow the enabled and disabling of certain triggers

Field	Type	NULL allowed	Default Value
name*	TEXT	YES	
enabled	INTEGER	NO	TRUE

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists trigger_settings (name TEXT PRIMARY KEY, enabled INTEGER␣
↪NOT NULL DEFAULT TRUE);
INSERT INTO trigger_settings (name, enabled) VALUES('new_link_a_or_b_node', TRUE);
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'trigger_settings', 'name', 'name for trigger to query against');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'trigger_settings', 'enabled', 'boolean value');
```

trips table structure

The *trips* table holds information on trips for each route. This table comes from the GTFS file *trips.txt*. You can find more information about the trips table here.

trip_id identifies a trip

trip identifies the trip to a rider

dir indicates the direction of travel for a trip

pattern_id is an unique pattern for the route

Field	Type	NULL allowed	Default Value
trip_id*	INTEGER	NO	
trip	TEXT	YES	
dir	INTEGER	NO	
pattern_id	INTEGER	NO	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS trips (
    trip_id         INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    trip            TEXT,
    dir             INTEGER NOT NULL,
    pattern_id      INTEGER NOT NULL,
    FOREIGN KEY(pattern_id) REFERENCES routes(pattern_id) deferrable initially␣
↪deferred
);
```

trips schedule table structure

The *trips_schedule* table holds information on the sequence of stops of a trip.

trip_id is an unique identifier of a trip

seq identifies the sequence of the stops for a trip

arrival identifies the arrival time at the stop

departure identifies the departure time at the stop

Field	Type	NULL allowed	Default Value
trip_id*	INTEGER	NO	
seq	INTEGER	NO	
arrival	INTEGER	NO	
departure	INTEGER	NO	

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE IF NOT EXISTS trips_schedule (
    trip_id   INTEGER  NOT NULL,
    seq       INTEGER  NOT NULL,
    arrival   INTEGER  NOT NULL,
    departure INTEGER  NOT NULL,
    PRIMARY KEY(trip_id,"seq"),
    FOREIGN KEY(trip_id) REFERENCES trips(trip_id) deferrable initially deferred
);
```

zones table structure

The *zones* table holds information on the Traffic Analysis Zones (TAZs) in AequilibraE's model.

The **zone_id** field identifies the zone.

The **area** field corresponds to the area of the zone in **km2**. TAZs' area is automatically updated by triggers.

The **name** fields allows one to identity the zone using a name or any other description.

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
zone_id	INTEGER	NO	
area	NUMERIC	YES	
name	TEXT	YES	
geometry	MULTIPOLYGON	NO	‘’

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE 'zones' (ogc_fid    INTEGER PRIMARY KEY,
                      zone_id    INTEGER UNIQUE NOT NULL,
                      area       NUMERIC,
                      "name"     TEXT);

SELECT AddGeometryColumn( 'zones', 'geometry', 4326, 'MULTIPOLYGON', 'XY', 1);
CREATE UNIQUE INDEX idx_zone ON zones (zone_id);
SELECT CreateSpatialIndex( 'zones' , 'geometry' );
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','zone_id', 'Unique node ID');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','area', 'Area of the zone in km2');
```

(continues on next page)

(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','name', 'Name of the zone, if any');
```

2.2.7 Project Data Components

In the *Project structure* section, we present the structure of an AequilibraE project: databases, folders, and parameters. In this section, we present the data components of the project, that is, the data that is presented in the databases.

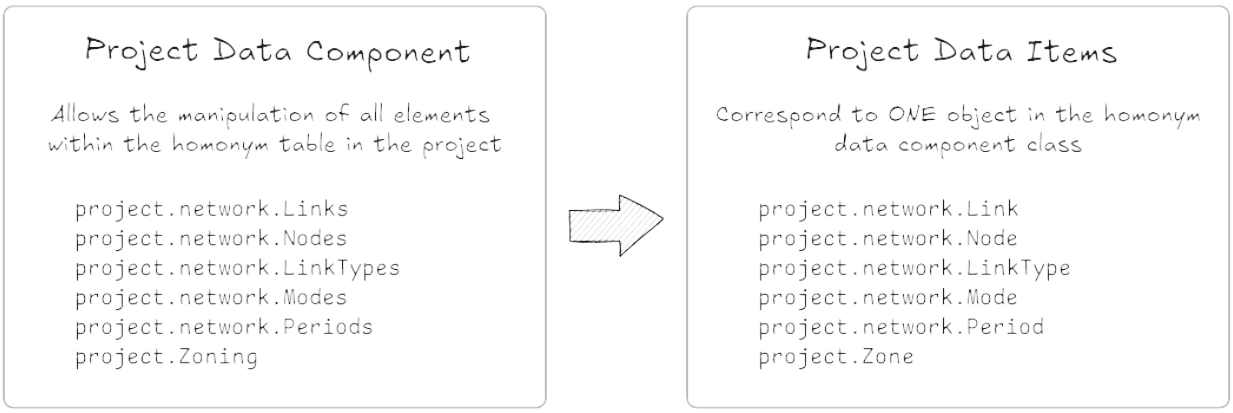
The components of an AequilibraE project are:

- `project.about`
- `project.tables`
- `project.road`
- `project.matrices`
- `project.log()`

The Tables component contains the geo-spatial information of the project, such as links, nodes, and zones, which can also be manipulated. In the Tables component, there are also non-geometric classes related to the project network, such as Modes, LinkTypes, and Periods.

Use `project.road.build_network()` to copy the tables into an independent routing network, or `project.road.build_graph(mode, cost)` to build a graph directly. The road gateway follows the project's current scenario; table editing remains on `project.tables`.

One important thing to observe is that related to each table in Tables, there is an object with similar name that corresponds to one object in the class. Thus `project.tables.links` enables the access to manipulate the 'links' table, and each item in the items table is a `Link` object.



Components

An AequilibraE project holds geometric information that can be accessed by the user in three different classes: `Links`, `Nodes`, and `Zones`. We'll first cover these classes, and then we'll go over the project components without geo-spatial information.

`project.tables.links`

This method allows you to access the API resources to manipulate the 'links' table. Each item in the 'links' table is an immutable `LinkRecord`.

```
>>> from shapely.geometry import LineString

>>> project = create_example(project_path, "coquimbo")

>>> project_links = project.tables.links

# Let's add a new field to our 'links' table
>>> project_links.fields.add("my_field", "This is an example", "TEXT")

# Let's add a new link to our project
>>> new_link_id = project_links.insert(modes="bctw", geometry=LineString([(-71.304754,
↪ -29.955233), (-71.304863, -29.954049)]))

# The 'links' table has three fields which cannot be empty (i.e. with `NULL` values):
# `link_id`, `direction`, and `modes`. `insert()` generates a `link_id`, defaults
# direction to 0, and derives endpoint nodes from the geometry. Thus, the modes
# information must be provided.

# To delete one link from the project, you can use the following
>>> project_links.delete(21332)

# The `copy` function creates a copy of a specified link and returns its ID.
# It is very helpful when you want to split a link.
# You can check this in one of the usage examples.
>>> link_copy_id = project_links.copy(10972)
```

References

- *Link layer changes and expected behavior*

See also

- **aequilibrae.project.network.links.Links()**
  Class documentation
- ***Create project from a link layer***
  Usage example
- ***Editing network geometry: Splitting link***
  Usage example

project.tables.nodes

This method allows you to access the API resources to manipulate the 'nodes' table. Each item in the 'nodes' table is an immutable `NodeRecord`.

```
>>> from shapely.geometry import Point

>>> project_nodes = project.tables.nodes

# To get one node record
```

(continues on next page)

(continued from previous page)

```
>>> node = project_nodes.get(10070)

# We can check the existing fields for each node in the 'nodes' table
>>> project_nodes.columns
('node_id', 'is_centroid', 'modes', 'link_types', 'geometry', 'osm_id', 'other_
↪attributes')

# Let's renumber this node
>>> project_nodes.renumber(node_id=10070, new_id=1000)

# A node can also be used to add a special generator
# `new_centroid` inserts a centroid and returns its ID.
# A geometry is required for a new centroid
# This centroid corresponds to the Port of Coquimbo!
>>> centroid_id = project_nodes.new_centroid(2000, geometry=Point(-71.32, -29.94))
>>> centroid = project_nodes.get(centroid_id)

# As this centroid is not associated with a zone, we must tell AequilibraE the␣
↪initial area around
# the centroid to look for candidate nodes to which the centroid can connect.
>>> project_nodes.connect_mode(node_id=centroid_id, area=centroid.geometry.buffer(0.
↪01), mode_id="c")

# Last but not less important, you can check your project nodes
# `project_nodes.data` returns a geopandas GeoDataFrame.
>>> nodes_data = project_nodes.data

>>> # or if you want to check the coordinate of each node in the shape of
>>> # a Pandas DataFrame
>>> coords = project_nodes.lonlat
>>> coords.head(3)
  node_id        lon        lat
0   10037 -71.315117 -29.996804
1   10064 -71.336604 -29.949050
2   10065 -71.336517 -29.949062
```

References

- *Node layer changes and expected behavior*

See also

- **aequilibrae.project.network.nodes.Nodes()**
  Class documentation
- *Editing network geometry: Nodes*
  Usage example

project.tables.zones

This method allows you to access the API resources to manipulate the 'zones' table. Each item in the 'zones' table is an immutable `ZoneRecord`.

```
>>> from shapely.geometry import Point, Polygon

>>> project_zones = project.tables.zones

# Let's start this example by adding a new field to the 'zones' table
>>> project_zones.fields.add("parking_spots", "Number of public parking spots",
↪"INTEGER")

# We can check if the new field was indeed created
>>> sorted(project_zones.fields.all_fields())
['area', 'employment', 'geometry', 'name', 'parking_spots', 'population', 'zone_id']

# Now let's get a zone and modify it
>>> zone = project_zones.get(40)

# By disconnecting the transit mode
>>> project_zones.disconnect_mode("t")

# Connecting the bicycle mode
>>> project_zones.connect_mode("b")

# And adding the number of public parking spots in the field we just created
>>> project_zones.update(zone.zone_id, parking_spots=30)

# The changes connecting / disconnecting modes reflect in the zone centroids
# and can be seen in the 'nodes' table.

# To return a dictionary with all zone records in the model
>>> zones_by_id = {zone.zone_id: zone for zone in project_zones}

# If you want to delete a zone
>>> project_zones.delete(38)

# Or to add a new one
>>> zone_extent = Polygon([(-71.3325, -29.9473), (-71.3283, -29.9473), (-71.3283, -29.
↪9539), (-71.3325, -29.9539)])

>>> new_zone_id = project_zones.insert(zone_id=38, geometry=zone_extent)

# We can add a centroid to the zone we just created by specifying its location or
```

(continues on next page)

(continued from previous page)

```
# pass `None` to use the geometric center of the zone
>>> project_zones.add_centroid(new_zone_id, Point(-71.33, -29.95))

# Finally, to return a geopandas GeoDataFrame with the project zones
>>> zones = project_zones.data

# To get a Shapely Polygon or Multipolygon with the entire zoning coverage
>>> boundaries = project_zones.coverage()

# And to get the nearest zone to a given geometry
>>> project_zones.get_closest_zone(Point(-71.3336, -29.9490))
57

>>> project.close()
```

See also

- **aequilibrae.project.network.zones.Zones()**
  Class documentation
- *Create a zone system based on Hex Bins*
  Usage example

project.about

This table object provides access to project metadata. Use the standard table API to add metadata records or update existing values.

```
>>> project = create_example(my_folder_path / "about")

>>> project.about.insert(infoname="my_new_field")
'my_new_field'
>>> project.about.update("my_new_field", infovalue="add some useful information about␣
↪the field")

# We can update data in an existing field
>>> project.about.update("author", infovalue="Your Name")

# Metadata is available as table data
>>> "my_new_field" in project.about.data.infoname.values
True

>>> project.close()
```

See also

- **aequilibrae.project.about.About()**
  Class documentation
- *About table*
  Table documentation

project.FieldEditor

The `FieldEditor` allows the user to edit the project data tables, and it has two different purposes:

- Managing data tables by adding fields
- Editing the tables' metadata (aka the description of each field)

This class is directly accessed from within the corresponding module one wants to edit.

```
>>> project = create_example(my_folder_path / "field_editor", "nauru")

# We'll edit the fields in the 'nodes' table
>>> node_fields = project.tables.nodes.fields

# To add a new field to the 'nodes' table
>>> node_fields.add("my_new_field", "this is an example of AequilibraE's␣
↪functionalities", "TEXT")

# To edit the description of a field
>>> node_fields.set_description("osm_id", "number of the osm node_id")

# Or just to access the description of a field
>>> node_fields.get_description("modes")
'Modes connected to the node'

# One can also check all the fields in the 'nodes' table.
>>> node_fields.all_fields()
[..., 'my_new_field']

# Add two directional columns that share one description
>>> link_fields = project.tables.links.fields
>>> link_fields.add("sample_cost", "Cost of traversing the link", "REAL",␣
↪directional=True)
>>> link_fields.get_description("sample_cost_ab")
'Cost of traversing the link'
>>> link_fields.get_description("sample_cost_ba")
'Cost of traversing the link'

# Add several fields and edit several descriptions
>>> link_fields.add_many([
...     ("survey_note", "Survey note", "TEXT"),
...     ("survey_year", "Year of survey", "INTEGER"),
... ])
>>> link_fields.set_descriptions([
...     ("sample_cost_*", "Cost per vehicle"),
...     ("survey_note", "Note from the field survey"),
... ])

>>> project.close()
```

Field additions and description edits take effect immediately. All field descriptions are kept in the table 'attributes_documentation'. To refer to a shared directional description, use a directional column name or `name_*`. The base name also works when there is no unsuffixed column of that name. Each `add_many()` or `set_descriptions()` call applies the entire batch in one transaction.

See also

- **aequilibrae.project.field_editor.FieldEditor()**
  Class documentation

project.log

Every AequilibraE project contains a log file that holds information on all the project procedures. It is possible to access the log file contents, as presented in the next code block.

```
>>> project = create_example(my_folder_path / "log", "nauru")

>>> project_log = project.log()

# Returns a list with all entries in the log file.
>>> print(project_log.contents())
['2021-01-01 15:52:03,945;aequilibrae;INFO ; Created project on D:/release/Sample␣
↪models/nauru', ...]

# If your project's log is getting cluttered, it is possible to clear it.
# Use this option wisely once the deletion of data in the log file can't be undone.
>>> project_log.clear()

>>> project.close()
```

See also

- *aequilibrae.Project.log()*
  Project log access
- *Checking AequilibraE's log*
  Usage example

project.matrices

This manager provides access to all the matrix files available in the model. Matrices are identified by their filenames, and their metadata is stored in the OMX files.

`get_matrix()` returns a lazy `MatrixStore` by default, so OMX data can be inspected without loading every matrix into memory. Pass `subset` to load the selected matrices as an in-memory `AequilibraeMatrix` for computation:

```
project = create_example(my_folder_path / "matrices")
matrices = project.matrices
with matrices.get_matrix("demand_mc.omx") as store:
    matrix_names = list(store)
demand = matrices.get_matrix("demand_mc.omx", subset=["car", "trucks"])
```

If `index_name` is omitted, the only OMX index is selected automatically. When an OMX file has multiple indices, `main_index` is preferred if present, otherwise, provide `index_name` explicitly.

```
>>> project = create_example(my_folder_path / "matrices")
```

(continues on next page)

(continued from previous page)

```
>>> matrices = project.matrices

# One can also check all the project matrices as a Dictionary
>>> matrices.get_available_matrices_and_metadata()

# We can add a new matrix
>>> demand = matrices.get_matrix("demand_mc.omx", subset=["car", "trucks"])
>>> matrices.save_matrix("demand_copy.omx", demand)

# If you're unsure if you have a matrix in your project, you can check if it exists.
# This function will return `True` or `False`.
>>> matrices.file_exists("demand_copy.omx")
True

# To delete a matrix from the project, we can delete the file directly
>>> (matrices.folder / "demand_copy.omx").unlink()

# Or open a matrix file to edit its metadata
>>> with matrices.get_matrix("demand_mc.omx", mode="r+") as matrix:
...     matrix.metadata["description"] = "Multi-class demand"

>>> project.close()
```

See also

- ***aequilibrae.Project.matrices***
  Project matrix manager

project.tables.link_types

This method allows you to access the API resources to manipulate the 'link_types' table. Each item in the 'link_types' table is an immutable `LinkTypeRecord`.

```
>>> project = create_example(my_folder_path / "link_types", "coquimbo")

>>> link_types = project.tables.link_types

# Create a new LinkType with ID 'A'
>>> link_types.insert(link_type_id="A", link_type="Arterial")
'A'

# We can update information for the LinkType we just created
>>> link_types.update("A", description="This is a description", speed=35)

# To create a new field in the 'link_types' table, you can call the function `fields`
# to return a FieldEditor instance, which can be edited
>>> link_types.fields.add("my_new_field", "this is an example of AequilibraE's␣
↪functionalities", "TEXT")

# You can also remove a LinkType from a project using its `link_type_id`
```

(continues on next page)

(continued from previous page)

```
>>> link_types.delete("A")

# To collect all LinkType records by their IDs
>>> link_types_by_id = {link_type.link_type_id: link_type for link_type in link_types}

# There are two ways to get a LinkType from the 'link_types' table
# using the `link_type_id`
>>> get_link = link_types.get("p")

# or using the `link_type`
>>> get_link = link_types.get_by_name("primary")

>>> project.close()
```

See also

- **aequilibrae.project.network.link_types.LinkTypes()**
  Class documentation
- *Link types table*
  Table documentation

project.tables.modes

This method allows you to access the API resources to manipulate the 'modes' table. Each item in 'modes' table is an immutable `ModeRecord`.

```
>>> project = create_example(my_folder_path / "modes", "coquimbo")

>>> modes = project.tables.modes

# We create a new mode
>>> modes.insert(mode_id="k", mode_name="flying_car")
'k'

# We can continue editing the mode after adding it
>>> modes.update("k", description="Like the one in the cartoons")

# You can also remove a Mode from a project using its ``mode_id``
>>> modes.delete("k")

# To collect all Mode records by their IDs
>>> modes_by_id = {mode.mode_id: mode for mode in modes}

# There are two ways to get a Mode from the 'modes' table
# using the ``mode_id``
>>> get_mode = modes.get("c")

# or using the ``mode_name``
>>> get_mode = modes.get_by_name("car")
```

(continues on next page)

(continued from previous page)

```
>>> project.close()
```

See also

- **aequilibrae.project.network.modes.Modes()**
  Class documentation
- ***Modes table***
  Table documentation

project.tables.periods

This method allows you to access the API resources to manipulate the 'periods' table. Each item in the 'periods' table is a `Period` object.

```
>>> project = create_example(my_folder_path / "periods", "coquimbo")

>>> periods = project.tables.periods

# Let's add a new field to our 'periods' table
>>> periods.fields.add("my_field", "This is field description", "TEXT")

# To see all periods data as a Pandas' DataFrame
>>> all_periods = periods.data

# To add a new period
>>> new_period_id = periods.new_period(2, 21600, 43200, "6AM to noon")

# We can update the new period with a value for the new field
>>> periods.update(new_period_id, my_field="hello world")

# It is also possible to renumber a period
>>> periods.renumber(new_period_id, 9)

# And check the existing data fields for each period
>>> periods.columns
('period_id', 'period_start', 'period_end', 'period_description', 'my_field')

>>> project.close()
```

See also

- **aequilibrae.project.network.periods.Periods()**
  Class documentation
- *Periods table*
  Table documentation

AequilibraE Matrix

AequilibraE provides `AequilibraeMatrix` for in-memory matrix work and `MatrixStore` for reading and updating OMX files on disk. For the complete API, see ../../matrix.

AequilibraeMatrix

Create an in-memory matrix by providing its zone index, then assign named matrix arrays directly:

```
>>> import numpy as np
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> from aequilibrae.matrix import AequilibraeMatrix, from_file

>>> index = np.arange(1, 6, dtype=np.uint64)
>>> values = np.ones((5, 5), dtype=np.float32)
>>> mat = AequilibraeMatrix(index=index, index_name="zones")
>>> mat["only_ones"] = values

# Inspect its names and zone index.
>>> list(mat)
['only_ones']
>>> mat.index
array([1, 2, 3, 4, 5], dtype=uint64)
>>> mat["only_ones"]
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]], dtype=float32)
```

Use the named array directly in computations. `subset()` creates an independent in-memory copy containing only the requested matrices:

```
>>> working = mat.subset(["only_ones"])
>>> working["only_ones"].shape
(5, 5)
```

Save an in-memory matrix as OMX with `save_as_omx()` and load it with `from_file()`. A single OMX index is inferred automatically; pass `index_name` when the file has multiple indices and no `main_index`.

```
>>> matrix_tmpdir = TemporaryDirectory()
>>> omx_path = Path(matrix_tmpdir.name) / "matrix_example.omx"
>>> mat.save_as_omx(omx_path)
>>> loaded = from_file(omx_path, subset=["only_ones"])
```

(continues on next page)

(continued from previous page)

```
>>> loaded["only_ones"].shape
(5, 5)
```

A `MatrixStore` keeps data on disk and is useful when loading every matrix into memory is undesirable. Select a working subset directly with `from_file(..., subset=...)` or use a store to inspect the file before selecting matrices:

```
>>> with from_file(omx_path, store=True) as store:
...     list(store)
['only_ones']
```

OpenMatrix (OMX)

AequilibraE uses OMX files as a standard format for storing its matrices. If you're wondering what OMX is and what it stands for, this section is for you. The text in this section is borrowed from OpenMatrix Wiki page.

The OpenMatrix file format (or simply OMX) is a standard matrix format for storing and transferring matrix data across different models and software packages, intended to make the model development easier. It is a file capable of storing more than one matrices at a time, including multiple indexes/lookups, and attributes (key/value pairs) for matrices and indexes.

There are APIs in different programming languages that allow you to use OMX. In Python, we use the `omx-python` library. See its project page for a brief tutorial.

2.2.8 Logging System Internals

AequilibraE implements a custom logging infrastructure designed to handle three specific constraints:

1. **Progress Bar Compatibility:** Preventing log messages from breaking the visual structure of `tqdm` bars in the terminal.
2. **Thread Safety (GIL-free):** Allowing logging from heavy computational loops in Cython (`nogil`) and C++ without acquiring the GIL.
3. **QGIS Integration:** Routing logs correctly when running inside the QGIS Python environment.

For an example of the logging to the terminal see *Logging to terminal*.

Python Logging

All Python code should utilise the standard Python `logging` module.

We should aim to module-level loggers using `__name__`.

```
import logging

# Define at the top of the module
logger = logging.getLogger(__name__)

def complex_function():
    logger.info("Starting process...")
    try:
        # logic
        logger.debug("Detailed step info")
    except Exception:
        logger.error("Something went wrong", exc_info=True)
```

Logging configuration should be opt-in with easy-to-apply defaults (`aequilibrae.utils.logging_utils.basic_config()`).

If a more complicated logging configuration is required the `aequilibrae.utils.logging_utils.AequilibraEStreamHandler` class should be utilised to preserve progress bars on stream-based handlers.

Cython & C++ Logging (The Bridge object)

When operating within performance-critical loops (Cython `nogil` blocks or pure C++), standard Python logging is inaccessible because it requires the GIL. AequilibraE solves this using a `Bridge` object.

The `Bridge` accumulates log messages in a thread-safe C++ queue and consumes them in another Python thread, forwarding them to the standard Python logging system.

Using the Bridge in Cython

To log from `nogil` blocks, you must pass the `Bridge` object into your function/method.

1. **Import the logging utilities:**

```
from aequilibrae.utils.cython.bridge cimport Bridge, log, f, DEBUG, INFO, WARN,␣
↪ERROR
```

2. **Use the log macro:**

   The syntax is `log(bridge_instance, level, message_expression)`.

```
# Within a nogil block
cdef void heavy_computation(Bridge bridge, int iteration) nogil:
    # String formatting must use the 'f' helper for delayed evaluation
    log(bridge, INFO, f("Starting iteration: ", iteration))

    if iteration < 0:
        log(bridge, ERROR, f("Invalid iteration: ", iteration))
```

The third argument to `log` is an expression. In this example the `f()` helper (which wraps C++ stream formatting) is **only evaluated** if the log level threshold is met. This prevents expensive string construction for `DEBUG` logs when the level is set to `INFO`.

Using the Bridge in C++

When calling pure C++ code from Cython, pass the underlying pointer to the Bridge struct.

1. **Include the header:**

```
#include "aeq_log.h"
// This ensures you have access to the generated bridge structure definition and␣
↪all other required headers
```

2. **Pass the pointer:**

   Your C++ function signature should accept a `struct Bridge *bridge`. This struct is declared in the `bridge.h` header which is generated from `bridge.pxd` at compile time. However, it's preferred to include the `aeq_log.h` header as it also includes the other required headers for logging.

3. **Use the macro:**

   The macro syntax mirrors the Cython implementation.

```
#include "aeq_log.h"

void run_algo(struct Bridge *bridge) {
    int nodes = 100;
    // AEQ_LOG is defined in _aeq_log.h
    // aeq_format_string handles variadic arguments
    AEQ_LOG(bridge, 20, aeq_format_string("Graph has ", nodes, " nodes"));
}
```

Log Levels

The bridge module also maps standard Python log levels to integers which are accessible without the GIL. They are available as constants in `aequilibrae.utils.cython.bridge`:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

2.2.9 Examples

AequilibraE Project

Upgrade project database

In this example, we show how to upgrade a project database to the latest version. This is useful when you need to use the latest AequilibraE's database schemas or formats.

References

- database_migration

See also

Functions used in this example:

- *aequilibrae.Project.upgrade()*

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join

from aequilibrae.project import MigrationManager
from aequilibrae.examples import create_example
from aequilibrae.utils.spatialite_utils import connect_spatialite
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)

# Let's use Sioux Falls project
project = create_example(fldr)
```

To upgrade all database migrations in a single transaction, we can use:

```
# project.upgrade()
```

However, it is possible to upgrade only the project database.

```
project.upgrade(ignore_transit=True, ignore_results=True)
```

Finally, we close the project

```
project.close()
```

Logging to terminal

This example demonstrates how to configure and use the logging system in AequilibraE.

For details of the logging setup used with AequilibraE see *Logging System Internals*.

AequilibraE uses the standard Python *logging* module, but with specific configurations to ensure compatibility with:

1. Progress bars (tqdm), so logs do not break the visual bars in the terminal/notebook.
2. QGIS integration.
3. Multithreaded C++/Cython extensions.

Basic Configuration

By default, AequilibraE does not configure the root logger to avoid interfering with other libraries or your own script's settings. To see AequilibraE logs in your terminal or notebook, you should use the provided helper.

```
import logging
from aequilibrae.utils import basic_config
```

This helper installs a custom StreamHandler that writes to `stdout` via `tqdm.write`, ensuring that log messages print cleanly above active progress bars.

```
# Configure logging to INFO level
handler = basic_config(level=logging.INFO)

# Now we can log
logger = logging.getLogger("aequilibrae")
logger.info("This is a standard info message.")
```

How it works

When `aequilibrae.utils.logging_utils.basic_config()` is called:

1. It sets the level for the `"aequilibrae"` logger.

2. It sets `logger.propagate = False`. This prevents AequilibraE logs from bubbling up to the root logger. This is crucial because standard root handlers might print directly to streams that conflict with the terminal-based progress bars.
3. It attaches an `aequilibrae.utils.logging_utils.AequilibraEStreamHandler` which preserves the terminal-based progress bars.

You can customise the format and stream:

```
logger.removeHandler(handler)
basic_config(level=logging.DEBUG, format="%(levelname)s: %(message)s")
logger.debug("This is a debug message with a custom format.")
```

Note that we removed the first handler that `basic_config` added otherwise the message will appear twice. If the `"aequilibrae"` has any handlers writing to standard out or standard error then `basic_config` will do not do anything.

Configuring Specific Modules

All AequilibraE logging happens under the `"aequilibrae"` namespace. You can fine-tune logging for specific modules if you want to debug only one part of the system.

```
# Example: Turn on DEBUG logging only for the paths module
paths_logger = logging.getLogger("aequilibrae.routing")
paths_logger.setLevel(logging.DEBUG)
```

The parent `"aequilibrae"` logger might be set to WARNING, but this specific module will now output DEBUG messages.

Scenario-Based Logging

AequilibraE Projects handle logging automatically when switching scenarios.

When you open a project athe Project attaches a specific *FileHandler* to the logger.

- **One Log per Scenario:** If you run multiple scenarios, AequilibraE automatically switches the log file handler to a new file located in the scenario directory.
- **Auto-Switching:** You do not need to manually close or remove handlers; the logging system handles the rotation to ensure logs from scenario A don't end up in scenario B's file.

Avoiding the automatic download of SpatiaLite binaries on Windows

In this example, we show how to prevent Windows from downloading the SpatiaLite binaries automatically.

This may be relevant to users in corporate environments where the download and use of binaries to the Windows temporary is restricted.

Spatialite Logo by Massimo Zedda, image from https://www.gaia-gis.it/

```
# Imports
import os
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

from aequilibrae.examples import create_example
```

```
from aequilibrae.utils import set_known_spatialite_folder, ensure_spatialite_binaries

# First we prevent Windows from downloading spatialite binaries during this session
# THIS VALUE MUST BE UPPER CASE TO BE EFFECTIVE
os.environ["AEQ_SPATIALITE_DIR"] = r"C:\path\to\existing\download"
```

Now we can go about our business as usual

```
project = create_example(join(gettempdir(), uuid4().hex))
project.close()
```

Project Connections

In this example, we show how to use AequilibraE's database connections within a project.

References

- *The AequilibraE project*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.db_connection()*
- *aequilibrae.Project.results_connection()*
- *aequilibrae.Project.transit_connection()*

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from pathlib import Path

import geopandas as gpd
import pandas as pd

from aequilibrae.examples import create_example
```

```
# We create the example project inside our temp folder.
fldr = Path(gettempdir()) / uuid4().hex
project = create_example(fldr, "sioux_falls")
```

All AequilibraE projects present three types of connections in the form of properties:

- Spatial connection to the project database
- General connection to the results database
- Spatial connection to the transit database

Each connection can be easily accessed as follows:

```
with project.db_connection as conn:
    modes = pd.read_sql("SELECT * FROM modes", conn)
```

```
modes
```

As the project database connection is spatial, it can also be used to query spatial data directly.

```
with project.db_connection as conn:
    nodes = gpd.read_postgis("SELECT zone_id, ST_AsBinary(geometry) geom FROM zones;",
↪ con=conn, geom_col="geom", crs=4326)
```

```
nodes.head()
```

For accessing both results and transit databases, the procedure is the same.

When you're done, don't forget to close the project.

```
project.close()
```

Checking AequilibraE's log

AequilibraE's log is a very useful tool to get more information about what the software is doing under the hood.

Information such as Traffic Class and Traffic Assignment stats, and Traffic Assignment outputs. If you have created your project's network from OSM, you will also find information on the number of nodes, links, and the query performed to obtain the data.

In this example, we'll use Sioux Falls data to check the logs, but we strongly encourage you to go ahead and download a place of your choice and perform a traffic assignment!

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.examples import create_example
from aequilibrae.traffic import TrafficAssignment, TrafficClass
from aequilibrae.traffic.vdf import bpr
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr)
```

```
network = project.road.build_network()
```

We get our demand matrix from the project and create a computational view

```
proj_matrices = project.matrices
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])
```

Now let's perform our traffic assignment

```
assigclass = TrafficClass(
    name="car",
```

(continues on next page)

(continued from previous page)

```
    mode="c",
    cost="free_flow_time",
    demand=demand,
    skimming=["free_flow_time", "distance"],
    graph_options={"block_centroid_flows": False},
)


assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=bpr,
    name_mapping={"alpha": 0.15, "beta": 4.0},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="bfw",
)
assig.execute(max_iterations=50, rgap=0.001)
```

```
with open(join(fldr, "aequilibrae.log")) as file:
    for idx, line in enumerate(file):
        print(idx + 1, "-", line)
```

The log records the Traffic Class specifications, including the network mode, cost field, PCE, value of time, and matrix totals.

The log then shows the Traffic Assignment specifications. We can see that the VDF parameters, VDF function, capacity and time fields, algorithm, maximum number of iterations, and target gap are just like the ones we set previously. The only information that might be new to you is the number of cores used for computation. If you haven't set any, AequilibraE is going to use the largest number of CPU threads available.

Each iteration records its relative gap and step size. `assig.report()` returns the same convergence data as a DataFrame.

In case you execute a new traffic assignment using different classes or changing the parameters values, these new specification values would be stored in the log file as well so you can always keep a record of what you have been doing. One last reminder is that if we had created our project from OSM, the lines on top of the log would have been different to display information on the queries done to the server to obtain the data.

Log image by OSRS Wiki

Project Scenarios

In this example, we show how to use AequilibraE's scenario system to manage multiple model variants within a single project, using different example networks to demonstrate scenario isolation and management.

References

- *The AequilibraE project*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.list_scenarios()*
- *aequilibrae.Project.use_scenario()*
- *aequilibrae.Project.create_empty_scenario()*
- *aequilibrae.Project.clone_scenario()*

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from pathlib import Path

from aequilibrae.examples import create_example
from aequilibrae.traffic import TrafficAssignment, TrafficClass
from aequilibrae.traffic.vdf import bpr
```

```
# We create the example project inside our temp folder.
fldr = Path(gettempdir()) / uuid4().hex
project = create_example(fldr, "sioux_falls")
```

Working with scenarios

Let's first see what scenarios exist in our project

```
project.list_scenarios()
```

The root scenario is always present and represents the base model. Let's examine the current scenario's network

```
print(f"Current scenario network has {len(project.tables.links.data)} links")
print(f"Current scenario network has {len(project.tables.nodes.data)} nodes")
```

Creating new scenarios

We can create empty scenarios or clone existing ones

```
# Create an empty scenario to manually populate with a future/different network
project.create_empty_scenario("test_modifications", "Scenario for testing network␣
↪modifications")

# Clone the root scenario to preserve the original network
project.clone_scenario("limited_capacity", "Testing different assignment parameters")
```

Let's see our updated scenario list

```
project.list_scenarios()
```

Switching between scenarios

Each scenario operates independently with its own data

```
# Switch to the cloned scenario
project.use_scenario("limited_capacity")
print(f"This scenario has {len(project.tables.links.data)} links")

# Modify the network
with project.db_connection as conn:
    conn.execute("UPDATE links SET capacity_ab=capacity_ab/2, capacity_ba=capacity_ba/
↪2 WHERE link_id > 20 AND link_id < 50")
```

Let's perform a traffic assignment in this scenario with lowered capacity

```
# Load the network snapshot
network = project.road.build_network()

# Get the demand matrix
mat = project.matrices.get_matrix("demand.omx", subset=["matrix"])

# Create traffic assignment with alternative parameters
traffic = TrafficClass(
    name="car", mode="c", cost="distance", demand=mat, graph_options={"block_centroid_
↪flows": False}
)
assignment = TrafficAssignment(
    network=network, classes=[traffic], time_field="distance", capacity_field=
↪"capacity",
    vdf=bpr, name_mapping={"alpha": 0.15, "beta": 4.0}, algorithm="msa",␣
↪project=project,
)
assignment.execute(max_iterations=10)

# Save results specific to this scenario
assignment.save_results("alternative_assignment")

print(f"Assignment completed. Total flow: {assignment.class_results['car'].total_link_
↪loads.sum():.2f}")
```

Switch to empty scenario for modifications

```
project.use_scenario("test_modifications")
print(f"Empty scenario has {len(project.tables.links.data)} links")

# This scenario starts with an empty network, suitable for building from scratch
# or testing specific network configurations
```

Scenario isolation demonstration

Let's switch back to root and show that scenarios are isolated

```
project.use_scenario("root")
print(f"Back to root scenario with {len(project.tables.links.data)} links")
```

(continues on next page)

(continued from previous page)

```
# Check results - only root scenario results should be visible
root_results = project.results.list()
print(f"Root scenario has {len(root_results)} result tables")

# Switch to alternative scenario and check its results
project.use_scenario("limited_capacity")
alt_results = project.results.list()
print(f"Alternative scenario has {len(alt_results)} result tables")

# Each scenario maintains its own results database
alternative_assignment_exists = "alternative_assignment" in alt_results["table_name"].
↪values
print(f"Alternative assignment result exists in this scenario: {alternative_
↪assignment_exists}")
```

Best practices for scenario management

```
# Always return to root when doing project-wide operations
project.use_scenario("root")

# List scenarios for reference
final_scenarios = project.list_scenarios()
print("\nFinal scenario summary:")
for _, scenario in final_scenarios.iterrows():
    project.use_scenario(scenario['scenario_name'])
    link_count = len(project.tables.links.data)
    result_count = len(project.results.list())
    print(f"  {scenario['scenario_name']}: {link_count} links, {result_count} results
↪")
    print(f"    Description: {scenario['description']}")
```

Clean up

```
project.use_scenario("root")  # Always end on root scenario
project.close()
```

CHAPTER

THREE

RUN MODULE

AequilibraE provides a convenient method for defining model entry points and their default arguments via `run/__init__.py` and `parameters.yml` respectively. These can be used to couple model parameters and methods to run models to the model itself.

3.1 `run/__init__.py`

AequilibraE imports the run module when you access `project.run`. The property returns a dictionary keyed by the function names under `run` in `parameters.yml`. AequilibraE applies the configured arguments with `functools.partial`. For example, call `project.run["results_summary"]()` to run the configured results summary.

You do not need to list every function in `parameters.yml`, but each name you list must refer to a callable in the module.

By default an AequilibraE project comes with four example functions: `matrix_summary`, `graph_summary`, `results_summary`, and `example_function_with_kwargs`. The default `parameters.yml` lists the summary functions without extra arguments.

Functions must accept an explicit `project` argument, supplied by `Project.run`.

State within the module should be avoided as the file may be run multiple times.

3.2 Command line interface

Functions in the run module can also be invoked from the command line via the `aeq` entry point. `aeq run --help` lists the available functions, and each function accepts arguments derived from its signature, with the defaults from `parameters.yml` pre-applied (`--no-defaults` disables this).

```
aeq -p /path/to/project run --help
aeq -p /path/to/project run example_function_with_kwargs --help
aeq -p /path/to/project run example_function_with_kwargs --arg1 hello iterations=50
```

Required parameters become positional arguments and parameters with defaults become `--options`. Parameters whose default value is a boolean become `--flag`/`--no-flag` pairs, and functions accepting `**kwargs` take trailing `key=value` pairs. All values are parsed as Python literals where possible and kept as strings otherwise, mirroring how `parameters.yml` values are typed; quote a value (e.g. `'"101"'`) to force a string. Type annotations are shown in the help text but not enforced. If `-p`/`--project` is omitted the current directory is used.

3.2.1 Examples

Run module

Run module

In this example we demonstrate how to use AequilibraE's run module using Sioux Falls example.

References

- *Run module*
- *Run*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.run*

```
# Imports
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

from aequilibrae.examples import create_example
```

```
# Let's create the Sioux Falls example in an arbitrary folder.
folder = join(gettempdir(), uuid4().hex)

project = create_example(folder)
```

First, let's check the matrix information using `matrix_summary()`. This method provides us useful information such as the matrix total, minimum and maximum values in the array, and the number of non-empty pairs in the matrix.

Notice that the matrix summary is presented for each matrix core.

```
project.run["matrix_summary"]()
```

If our matrices folder is empty, instead of a nested dictionary of data, AequilibraE run would return an empty dictionary.

Let's create a graph for mode *car*.

```
mode = "c"
```

```
network = project.road.build_network()
graph = network.build_graph(mode=mode, cost="distance", skimming=["distance"], block_
↪centroid_flows=False)
```

The network mapping describes the source topology. Building with compact=True can reduce the routing counts without changing the source mapping.

```
{
    "network_num_links": network.mapping.link_count,
    "network_num_nodes": network.mapping.node_count,
    "num_zones": graph.num_zones,
    "graph_num_links": graph.num_links,
```

(continues on next page)

(continued from previous page)

```
    "graph_num_nodes": graph.num_nodes,
}
```

Let's add a `create_delaunay` function to our `run/__init__.py` file.

This function replicates the example in which we *create Delaunay lines*.

```
func_string = """
def create_delaunay(project, source: str, name: str, matrix_core: str, result_name:␣
↪str, overwrite: bool=False):\n
\tfrom aequilibrae.utils.create_delaunay_network import DelaunayAnalysis\n
\tmatrix = project.matrices\n
\tmat = matrix.get_matrix(name, subset=[matrix_core])\n
\tda = DelaunayAnalysis(project)\n
\tda.create_network(source, overwrite)\n
\tda.assign_matrix(mat, result_name)\n
"""
```

```
with open(join(folder, "run", "__init__.py"), "a") as file:
    file.write("\n")
    file.write(func_string)
```

Now we add new parameters to our model

```
p = project.project_parameters
p.parameters["run"]["create_delaunay"] = {}
p.parameters["run"]["create_delaunay"]["source"] = "zones"
p.parameters["run"]["create_delaunay"]["name"] = "demand.omx"
p.parameters["run"]["create_delaunay"]["matrix_core"] = "matrix"
p.parameters["run"]["create_delaunay"]["result_name"] = "my_run_module_example"
p.write_back()
```

And we run the function

```
project.run["create_delaunay"]()
```

Note

To run the `create_delaunay` function we created above without argument values, we must insert the values as a project parameter. Adding an unused parameter to the `parameters.yml` file will raise an execution error.

Creating Delaunay lines also creates a `results_database.sqlite` that contains the result of the all-or-nothing algorithm that generated the output. We can check the existing results in the results_database using the `results_summary` method.

```
project.run["results_summary"]()
```

Let's check what our Delaunay lines look like!

```
import geopandas as gpd
```

Let's retrieve the results

```
results = project.results.get_results("my_run_module_example").set_index("link_id")
```

```
with project.db_connection as conn:
    links = gpd.read_postgis(
        "SELECT link_id, st_asBinary(geometry) geometry FROM delaunay_network", conn,␣
↪geom_col="geometry", crs=4326
    )
links.set_index("link_id", inplace=True)
```

```
df = links.join(results)
max_vol = df.matrix_tot.max()
```

And finally plot the data

```
df.plot(linewidth=5 * df["matrix_tot"] / max_vol, color="blue")
```

```
project.close()
```

Pipeline image credits to Data-pipeline icons created by Vectors Tank - Flaticon

CHAPTER

FOUR

NETWORK MANIPULATION

In this section, we discuss how can we import and export data to/from an AequilibraE project. Besides, important concepts on geometry manipulation are presented. Finally, some examples that involve project creation, edition of links and nodes, and identification of disconnected links for network clean up are presented.

4.1 Importing and exporting the network

AequilibraE can populate a project network from OpenStreetMap, Overture Maps, GMNS files, and link layers. It can also export an existing project network to GMNS. This page documents the public import APIs and the behavior of the new OSM/Overture import pipeline.

4.1.1 Optional dependencies

The OSM and Overture importers use optional packages. Install them with the `create` extra before using these importers:

```
pip install aequilibrae[create]
```

The relevant optional packages are:

- `osmnx` for OSM Overpass downloads and OSMnx simplification;
- `pyrosm` for local `.osm.pbf` imports;
- `overturemaps` for Overture Maps cloud downloads;
- `neatnet` for neatnet simplification.

4.1.2 Importing from OpenStreetMap

Use `project.tables.importer.osm()` to create an AequilibraE network from OpenStreetMap. Exactly one source selector must be provided:

- `place_name` for an OSM place lookup through Overpass;
- `model_area` for an EPSG:4326 polygon downloaded through Overpass;
- `pbf_path` for a local `.osm.pbf` extract read with `pyrosm`.

Examples:

```
project.tables.importer.osm(place_name="Nauru")

from shapely.geometry import box

project.tables.importer.osm(
```

(continues on next page)

(continued from previous page)

```
    model_area=box(-112.185, 36.59, -112.179, 36.60),
    modes=("car", "walk"),
    simplify=False,
)

project.tables.importer.osm(
    pbf_path="/path/to/extract.osm.pbf",
    modes=("car", "transit", "bicycle", "walk"),
    simplify="osmnx",
)
```

`.osm`, `.osm.bz2` and XML inputs are not supported directly. Convert them to `.osm.pbf` first, for example with `osmium`:

```
osmium cat input.osm -o output.osm.pbf
```

The importer preserves the OSM `highway` value as `link_type`. It does not apply a link-type allow-list; filtering is controlled by the `modes` argument. OSM tags that do not map to real project table columns are stored as JSON in `other_attributes`.

Overpass imports write the downloaded payload and a manifest under `<project>/downloaded data/osm-overpass/`. Local PBF imports do not create a download cache because the source file is already local.

If Nominatim resolves a `place_name` only to a point rather than an administrative polygon, the importer uses the bounding box supplied with that top result and logs a warning describing the substituted area.

4.1.3 Importing from Overture Maps

Use `project.tables.importer.overture()` to import the Overture Maps transportation network for an EPSG:4326 polygon:

```
from shapely.geometry import box

project.tables.importer.overture(
    model_area=box(-112.185, 36.59, -112.179, 36.60),
    modes=("car", "walk"),
    simplify="osmnx",
)
```

The importer always uses the latest Overture Maps release advertised by the Overture STAC catalog. The release used for the import is written to the project `about` table and to the download-cache manifest.

The Overture importer reads the cloud `segment` and `connector` themes, splits segments at intermediate connectors, and derives AequilibraE link attributes from the Overture schema:

- `class` becomes `link_type`;
- connector order defines `a_node` and `b_node`;
- access restrictions determine `direction` where possible;
- global speed-limit rules populate `speed_ab` and `speed_ba`;
- additional Overture properties and rule arrays are stored in `other_attributes`.

The raw Overture tables and a manifest are written under `<project>/downloaded data/overture-cloud/`.

4.1.4 Mode filtering

All network import methods accept `modes` as a sequence of AequilibraE mode names. Supported names are:

- `"car"`;
- `"transit"`;
- `"bicycle"`;
- `"walk"`.

Only links with at least one requested mode are kept. The stored `modes` field contains AequilibraE mode codes, not the full mode names.

4.1.5 Network simplification

OSM and Overture imports can simplify the staged network before it is written to the project database. The `simplify` argument accepts:

- `False`: **no simplification. This is the default** – the imported network mirrors the source;
- `"osmnx"`: simplify with OSMnx;
- `"neatnet"`: simplify with neatnet;
- `True`: shorthand for `"osmnx"`.

Simplification is opt-in because it is lossy, and how lossy depends on the backend. Measured across the benchmark regions (*Import and simplification performance*, 122 successful runs), OSMnx preserves total network length to within a few per cent (median ratio 1.03 against the unsimplified import) while collapsing interstitial nodes; neatnet is more aggressive at removing geometric artifacts and discarded roughly half of total length (median ratio 0.49, range 0.30-1.00) while running a median 9x slower. Import first, inspect, then decide whether to simplify.

`consolidate_tolerance` controls intersection/node consolidation in metres (after automatic projection to a local UTM CRS) for both simplifiers. Set it to `None` to run OSMnx topological simplification without intersection consolidation; for neatnet, where node consolidation is integral to the algorithm, `None` falls back to the default tolerance of 10 metres:

```
project.tables.importer.osm(
    pbf_path="/path/to/extract.osm.pbf",
    simplify="osmnx",
    consolidate_tolerance=None,
)

project.tables.importer.overture(
    model_area=model_area,
    simplify=False,
)
```

Simplified links retain source provenance in `other_attributes`, under `source_ids`, for both backends.

neatnet needs enclosed street blocks to detect face artifacts. A tree-like network (a sparse rural or trail network with no closed loops) is returned unsimplified with a warning rather than failing.

4.1.6 Import and simplification performance

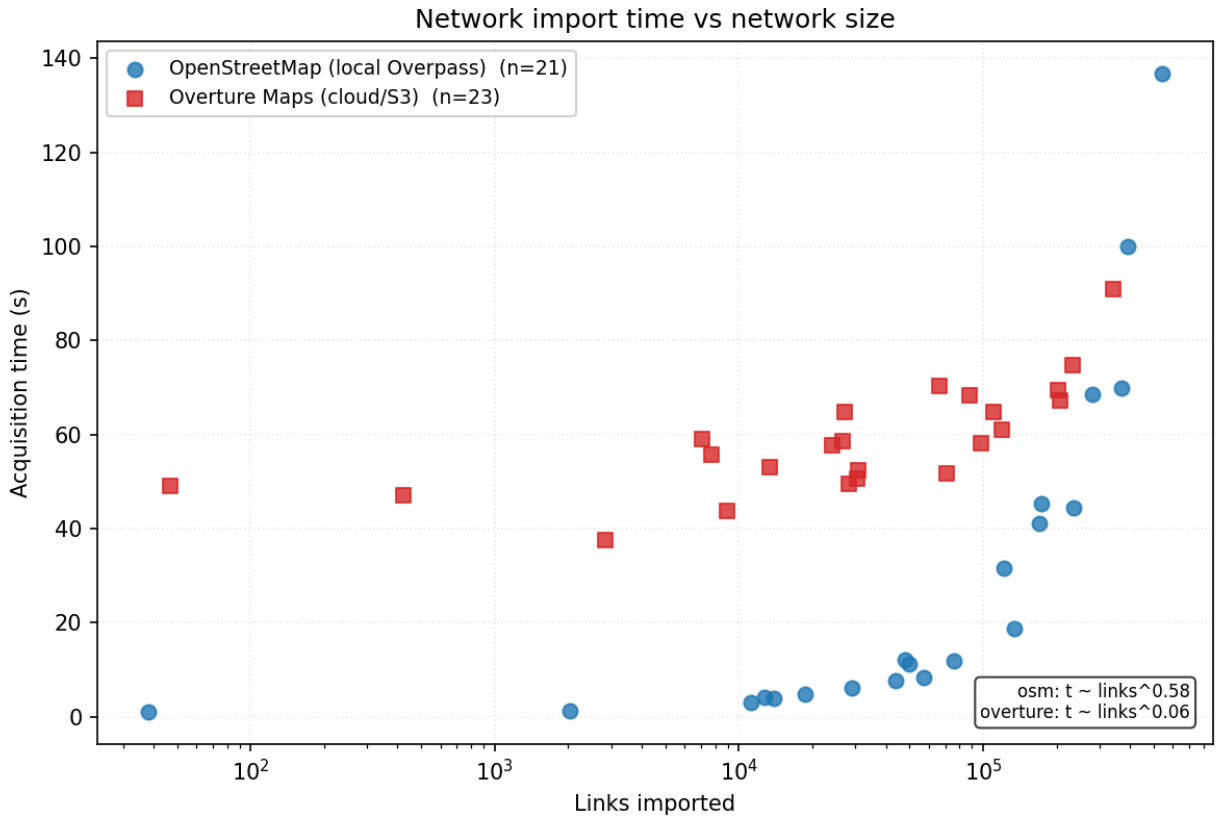
The numbers below come from the stress harness in `benchmarking/network_importer/`, which imports a matrix of regions from both sources with each simplifier and records per-phase wall time. Re-generate them with `python stress_import.py` followed by `python make_charts.py`.

Benchmark environment

Windows 11, 24-core CPU, 205 GB RAM, Python 3.13, osmnx 2.1, geopandas 1.1, pyarrow 24.

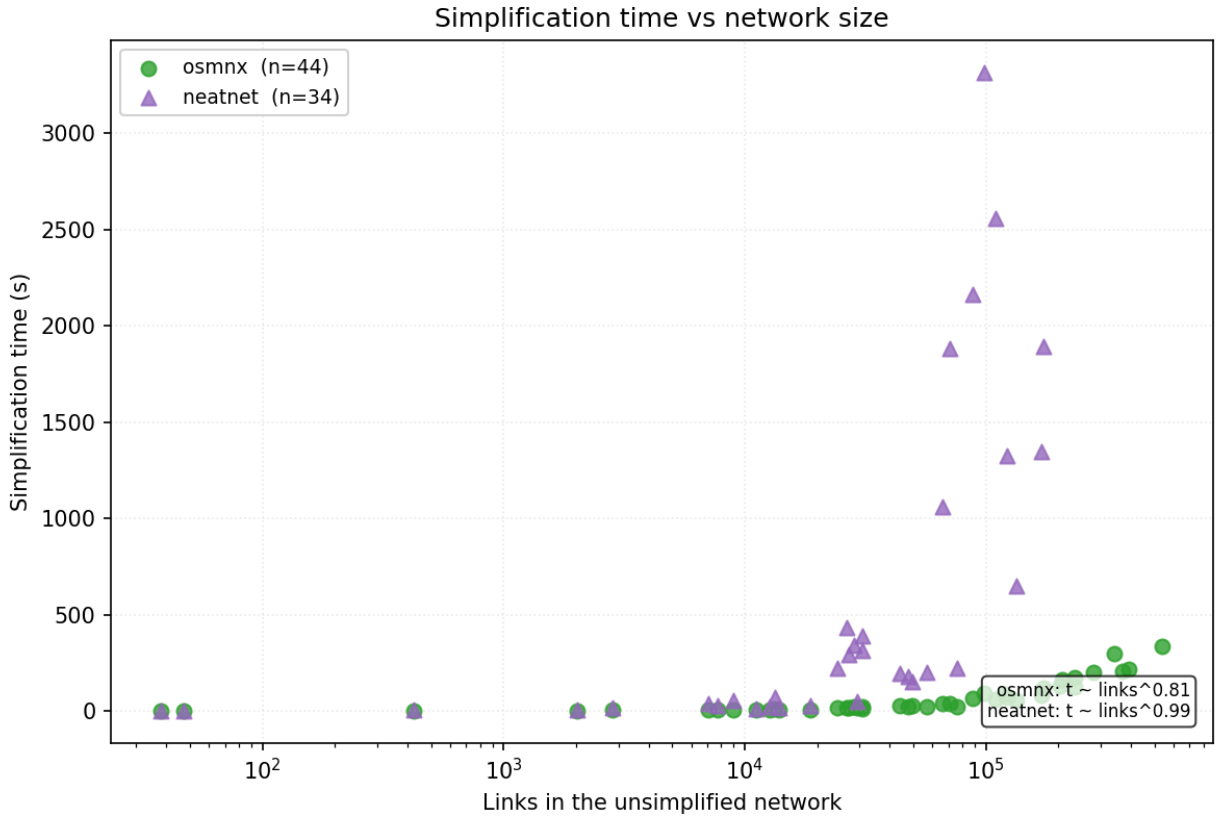
OpenStreetMap timings were measured against a local Overpass server on the same LAN as the machine running the tests – effectively a best case for download latency. A public Overpass endpoint is rate-limited and shared, so expect OSM acquisition to be substantially slower, and dominated by queueing rather than by network size. Overture timings are against the public `overturemaps-us-west-2` S3 bucket over the internet, so they include real cloud latency and are representative.

Import time vs network size



Both sources are dominated by a fixed startup cost at small sizes: Overture pays a roughly 48 s floor for S3 metadata and Parquet reads regardless of how small the area is (fitted exponent 0.06 – essentially flat with size), while a LAN-local Overpass answers in about a second and grows with the network (fitted exponent 0.58).

Simplification time vs network size



Both simplifiers scale close to linearly with link count (fitted exponents of 0.81 for OSMnx and 0.99 for neatnet), but the constant factor between them is large: neatnet ran 1.7-50x slower than OSMnx on the same network, median 9x. That is enough to make neatnet impractical on large metropolitan extracts: in the benchmark run it exceeded a one-hour budget on every one of the five largest regions, for both sources (10 runs), while OSMnx completed the same inputs in 85-337 seconds.

Indicative timings

Median seconds by network size. “Import” is acquisition only, excluding simplification and the database write.

Network size	Import (OSM)	Import (Overture)	osmnx	neatnet	Runs
< 1k	1 s	48 s	0.95 s	2 s	9
1k - 10k	1 s	50 s	4 s	27 s	15
10k - 100k	7 s	58 s	19 s	221 s	63
> 100k	45 s	68 s	129 s	1347 s	35

Warning

neatnet does not merely restructure the network, it removes a substantial share of it: across the benchmark regions the simplified network retained only 30-100% of the unsimplified total length (median 49%). Validate total length and connectivity before using a neatnet-simplified network for assignment.

Both simplifiers also currently introduce self-loops (links whose two endpoints are the same node), which never occur in unsimplified imports: 67 of the 122 successful benchmark runs had at least one. These are not usable for assignment and should be filtered out until the underlying node-consolidation issue is fixed.

4.1.7 Public API summary

The main entry points are:

- `project.tables.importer.osm()`;
- `project.tables.importer.overture()`;
- `project.tables.importer.source()` for explicit source names: `"osm-overpass"`, `"osm-pbf"` and `"overture-cloud"`;
- `project.tables.importer.gmns()`;
- `project.tables.export_to_gmns()` through *aequilibrae.Project.tables*.

Import methods share one namespace on the network object:

```
project.tables.importer.osm(...)
project.tables.importer.overture(...)
project.tables.importer.gmns(...)

project.tables.export_to_gmns(folder)
```

Note

The older flat import methods (`create_from_osm`, `import_from_osm`, `import_from_overture`, `import_network` and `create_from_gmns`) were removed in favour of the importer namespace.

Note

The OSM/Overture importer writes source-specific attributes to the existing `other_attributes` column on `links` and `nodes`. New projects contain these columns. Existing projects created with older schemas must be upgraded or recreated before using the importer.

See also

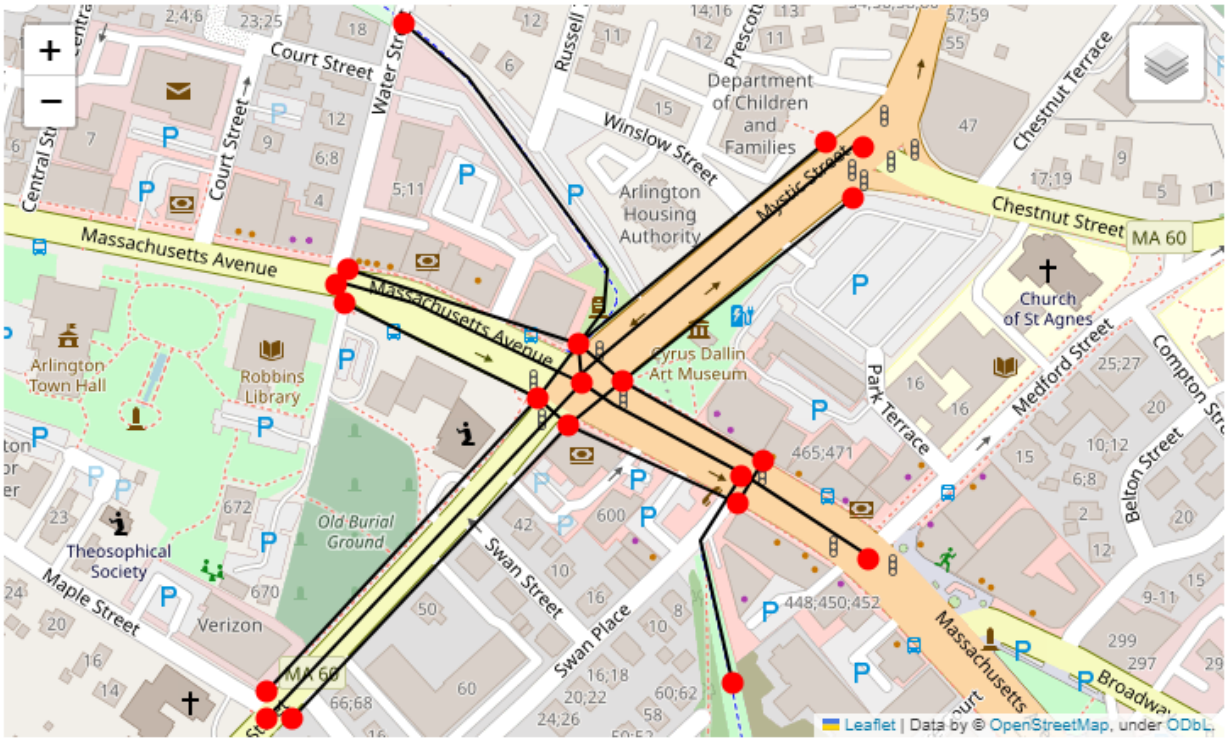
- *Create project from Overture Maps (or OpenStreetMap)*
  Network import example (Overture Maps, with OSM as an alternative)
- *Parameters YAML File*
  Project parameter file

4.1.8 Importing from link layer

It is possible to create an AequilibraE project from a link layer, such as a *.csv file that contains geometry in WKT, for instance. You can check an example with all functions used in *the following example*.

4.1.9 Importing from files in GMNS format

Before importing a network from a source in GMNS format, it is imperative to know in which spatial reference its geometries (links and nodes) were created. If the SRID is different than 4326, it must be passed as an input using the argument `srid`.



It is possible to import the following files from a GMNS source:

- link table;
- node table;
- use_group table;
- geometry table.

You can find the specification for all these tables in the GMNS documentation, here.

By default, `importer.gmns()` reads all required and optional fields specified in the GMNS link and node tables specification. If you need it to read any additional fields as well, you have to modify the AequilibraE parameters as shown in the *example*.

When adding a new field to be read in the parameters.yml file, it is important to keep the "required" key set to False, since you will always be adding a non-required field. Required fields for a specific table are only those defined in the GMNS specification.

Note

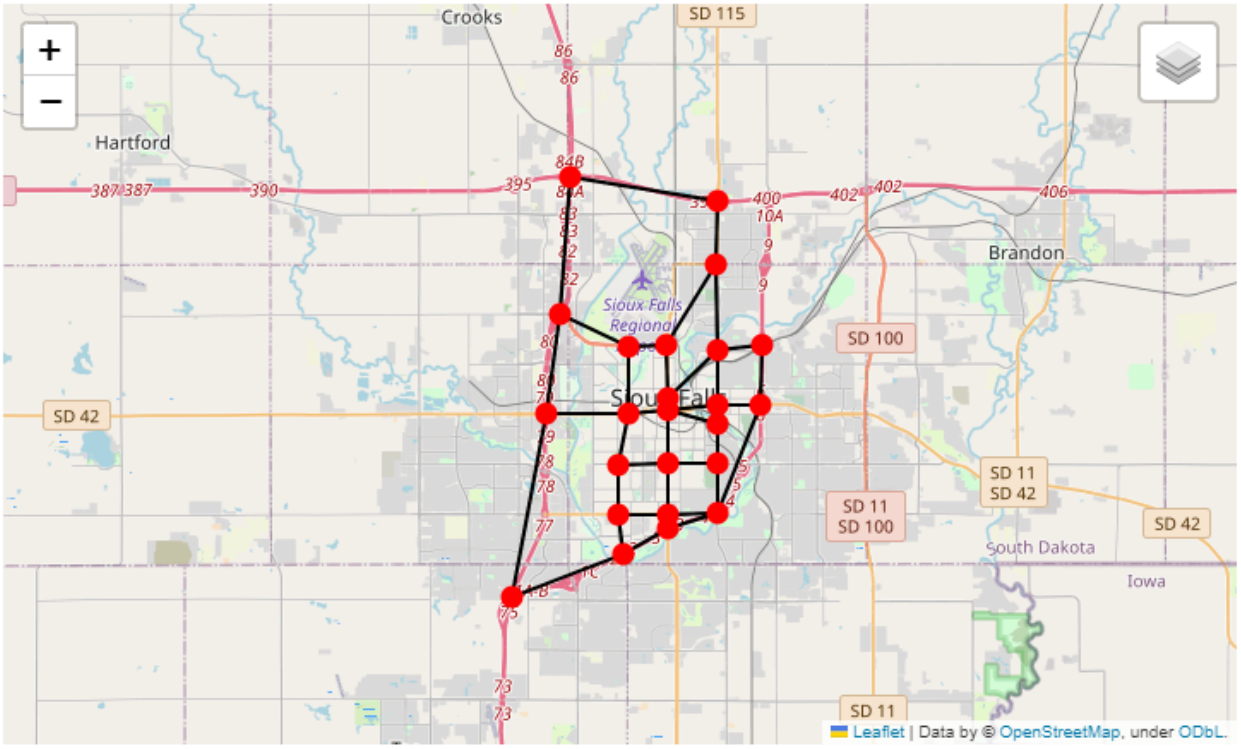
In the AequilibraE nodes table, if a node is to be identified as a centroid, its 'is_centroid' field has to be set to 1. However, this is not part of the GMNS specification. Thus, if you want a node to be identified as a centroid during the import process, in the GMNS node table you have to set the field 'node_type' equals to 'centroid'.

See also

- ***aequilibrae.Project.tables***
  Access to `project.tables.importer.gmns()`
- *Create project from GMNS*
  Usage example

4.1.10 Exporting AequilibraE model to GMNS format

After loading an existing AequilibraE project, you can export it to GMNS format.



It is possible to export an AequilibraE network to the following tables in GMNS format:

- link table
- node table
- use_definition table

This list does not include the optional 'use_group' table, which is an optional argument of the GMNS function, because mode groups are not used in the AequilibraE modes table.

In addition to all GMNS required fields for each of the three exported tables, some other fields are also added as reminder of where the features came from when looking back at the AequilibraE project.

Note

When a node is identified as a centroid in the AequilibraE nodes table, this information is transmitted to the GMNS node table by means of the field 'node_type', which is set to 'centroid' in this case. The 'node_type' field is an optional field listed in the GMNS node table specification.

You can find the GMNS specification here.

See also

- `project.tables.export_to_gmns()` through *aequilibrae.Project.tables*
  Function documentation
- *Exporting network to GMNS*
  Usage example

4.2 Dealing with Geometries

Geometry is a key feature when dealing with transportation infrastructure and actual travel. For this reason, all datasets in AequilibraE that correspond to elements with physical GIS representation, links and nodes in particular, are geo-enabled.

This also means that the AequilibraE API needs to provide an interface to manipulate each element's geometry in a convenient way. This is done using the standard Shapely, and we urge you to study its comprehensive API before attempting to edit a feature's geometry in memory.

As we mentioned in other sections of the documentation, the user is also welcome to use its powerful tools to manipulate your model's geometries, although that is not recommended, as the "training wheels are off".

4.2.1 Data consistency

Data consistency is not achieved as a monolithic piece, but rather through the *treatment* of specific changes to each aspect of all the objects being considered (i.e. nodes and links) and the expected consequence to other tables/elements. To this effect, AequilibraE has triggers covering a comprehensive set of possible operations for links and nodes, covering both spatial and tabular aspects of the data.

Although the behaviour of these trigger is expected to be mostly intuitive to anybody used to editing transportation networks within commercial modeling platforms, we have detailed the behaviour for all different network changes.

This implementation choice is not, however, free of caveats. Due to technological limitations of SQLite, some of the desired behaviors identified cannot be implemented, but such caveats do not impact the usefulness of this implementation or its robustness in face of minimally careful use of the tool.

Note

This documentation, as well as the SQL code it refers to, comes from the seminal work done in TranspoNet by Pedro and Andrew.

4.2.2 Network consistency behaviour

In order for the implementation of this standard to be successful, it is necessary to map all the possible user-driven changes to the underlying data and the behavior the SQLite database needs to demonstrate in order to maintain consistency of the data. The detailed expected behavior is detailed below. As each item in the network is edited, a series of checks and changes to other components are necessary in order to keep the network as a whole consistent. In this section we list all the possible physical (geometrical) changes to each element of the network and what behavior (consequences) we expect from each one of these changes.

Our implementation, in the form of a SQLite database, will be referred to as network from this point on.

Ensuring data consistency as each portion of the data is edited is a two part problem:

1. Knowing what to do when a certain edit is attempted by the user

2. Automatically applying the tests and consistency checks (and changes) required on one

The table below presents all meaningful operations that a user can do to links and nodes, and you can use the table below to navigate between each of the changes to see how they are treated through triggers.

Nodes	Links	Fields
Creating a node	*Deleting a link*	*Link distance*
Deleting a node	*Moving a link extremity*	*Link direction*
Moving a node	*Re-shaping a link*	*Field 'modes' (links and nodes layers)*
Adding a data field	*Deleting a required field*	*Fields 'link_type' (links layer) & 'link_types' (nodes layer)*
Deleting a data field		*Fields 'a_node' and 'b_node'*
Modifying a data entry		

Node layer changes and expected behavior

There are 6 possible changes envisioned for the network nodes layer, being 3 of geographic nature and 3 of data-only nature. The possible variations for each change are also discussed, and all the points where alternative behavior is conceivable are also explored.

Creating a node

There are only three situations when a node is to be created:

- Placement of a link extremity (new or moved) at a position where no node already exists
- Splitting a link in the middle
- Creation of a centroid for later connection to the network

In all cases a unique node ID needs to be generated for the new node, and all other node fields should be empty.

An alternative behavior would be to allow the user to create nodes with no attached links. Although this would not result in inconsistent networks for traffic and transit assignments, this behavior would not be considered valid. All other edits that result in the creation of unconnected nodes or that result in such case should result in an error that prevents such operation

Behavior regarding the fields regarding modes and link types is discussed in their respective table descriptions

Deleting a node

Deleting a node is only allowed in two situations:

- No link is connected to such node (in this case, the deletion of the node should be handled automatically when no link is left connected to such node)
- When only two links are connected to such node. In this case, those two links will be merged, and a standard operation for computing the value of each field will be applied.

For simplicity, the operations are: Weighted average for all numeric fields, copying the fields from the longest link for all non-numeric fields. Length is to be recomputed in the native distance measure of distance for the projection being used.

A node can only be eliminated as a consequence of all links that terminated/ originated at it being eliminated. If the user tries to delete a node, the network should return an error and not perform such operation.

Behavior regarding the fields regarding modes and link types is discussed in their respective table descriptions

Moving a node

There are two possibilities for moving a node: moving to an empty space, and moving on top of another node.

- If a node is moved to an empty space, all links originated/ending at that node will have its shape altered to conform to that new node position and keep the network connected. The alteration of the link happens only by changing the latitude and longitude of the link extremity associated with that node.
- If a node is moved on top of another node, all the links that connected to the node on the bottom have their extremities switched to the node on top. The node on the bottom gets eliminated as a consequence of the behavior listed on *Deleting a node*.

Behavior regarding the fields related to modes and link types is discussed in their respective table descriptions.

See also

- ***Editing network nodes***
  Usage example

Adding a data field

No consistency check is needed other than ensuring that no repeated data field names exist.

Deleting a data field

If the data field whose attempted deletion is mandatory, the network should return an error and not perform such operation. Otherwise the operation can be performed.

Modifying a data entry

If the field being edited is the node_id field, then all the related tables need to be edited as well (e.g. a_b and b_node in the link layer, the node_id tagged to turn restrictions and to transit stops).

Link layer changes and expected behavior

Network links layer also has some possible changes of geographic and data-only nature.

Deleting a link

In case a link is deleted, it is necessary to check for orphan nodes, and deal with them as prescribed in *Deleting a node*. In case one of the link extremities is a centroid (i.e. field `is_centroid=1`), then the node should not be deleted even if orphaned.

Behavior regarding the fields regarding modes and link types is discussed in their respective table descriptions.

Moving a link extremity

This change can happen in two different forms:

- The link extremity is moved to an empty space - In this case, a new node needs to be created, according to the behavior described in *Creating a node*. The information of node ID (A or B node, depending on the extremity) needs to be updated according to the ID for the new node created.
- The link extremity is moved from one node to another - The information of node ID (A or B node, depending on the extremity) needs to be updated according to the ID for the node the link now terminates in. Behavior regarding the fields regarding modes and link types is discussed in their respective table descriptions.

See also

- *Editing network links*
  Usage example

Re-shaping a link

When reshaping a link, the only thing other than we expect to be updated in the link database is their length (or distance, in AequilibraE's field structure). As of now, distance in AequilibraE is **ALWAYS** measured in meters.

See also

- *Splitting network links*
  Usage example

Deleting a required field

Unfortunately, SQLite does not have the resources to prevent a user to remove a data field from the table. For this reason, if the user removes a required field, they will most likely corrupt the project.

Field-specific data consistency

Some data fields are specially sensitive to user changes.

Link distance

Link distance cannot be changed by the user, as it is automatically recalculated using the SpatiaLite function `GeodesicLength`, which always returns distances in meters.

Link direction

Triggers enforce link direction to be -1, 0 or 1, and any other value results in an SQL exception.

Field 'modes' (links and nodes layers)

A series of triggers are associated with the modes field, and they are all described in the *Modes table*.

Fields 'link_type' (links layer) & 'link_types' (nodes layer)

A series of triggers are associated with the modes field, and they are all described in the *Link types table*.

Fields 'a_node' and 'b_node'

The user should not change the a_node and b_node fields, as they are controlled by the triggers that govern the consistency between links and nodes. It is not possible to enforce that users do not change these two fields, as it is not possible to choose the trigger application sequence in SQLite.

Turn restrictions trigger behaviour

The `turn_restrictions` table also has consistency triggers that are relevant for network editing, especially when links or nodes move.

- Node sequence consistency: the movement must define a valid `from_node -> via_node -> to_node` sequence.
- Mode integrity: mode codes in `turn_restrictions.modes` must exist in `modes.mode_id`.
- Mode-overlap protection: duplicate restrictions with overlapping modes for the same node-sequence movement are blocked.
- Geometry maintenance: `turn_restrictions.geometry` is rebuilt from the 3-node movement using SpatiaLite geometry functions, with each leg capped at 30 m and a small offset at the via node for loop movements.

For efficiency and maintainability, geometry is computed in one trigger that fires on movement field updates; insert and node-move events delegate to this same update path.

4.3 Turn Restrictions

AequilibraE supports turn restrictions and turn penalties for traffic assignment, path computation and route choice. This feature enables modelling of prohibited turns (e.g., no left turn signs) and turn penalties (e.g., additional time for turning movements).

4.3.1 Overview

Turn restrictions in AequilibraE are stored in a dedicated database table and can be applied to graphs during path computation, traffic assignment and route choice. When turn restrictions are enabled, AequilibraE uses an arc-based Dijkstra algorithm instead of the standard node-based algorithm, which allows for modelling turn-to-turn transitions with associated costs.

4.3.2 Database Schema

Turn restrictions are stored in the `turn_restrictions` table with the following columns:

- **restriction_id**: Unique identifier for the restriction (generated if omitted)
- **from_node**: Incoming movement origin node
- **via_node**: The turn node
- **to_node**: Outgoing movement destination node
- **penalty**: Turn penalty in time units (same as graph cost field). NULL indicates a prohibited turn. `+inf` is also accepted by the Python API and is normalised to NULL before storage. Negative values are not allowed.
- **modes**: String with the list of modes to which the restriction applies (e.g., “ct”, “c”, “w”).

Turn definitions are node-sequences: `from_node -> via_node -> to_node`.

4.3.3 Trigger Behaviour

Database triggers enforce and maintain turn restriction integrity:

- node-sequence consistency is enforced
- mode codes are validated against the `modes` table
- duplicate restrictions with overlapping mode coverage are blocked

- **restrictions are removed or remapped when linked `links.link_id` records**
  are deleted or changed

When geometry support is enabled, `turn_restrictions.geometry` is generated for the 3-node movement and updated when movement definition or node geometry changes.

See *Turn restrictions trigger behaviour* for network-editing details.

4.3.4 Global U-Turn Setting

The `allow_uturns` setting in the `about` table controls whether U-turns are permitted globally in the network. By default, this is set to `0` (False), meaning U-turns are prohibited.

U-turn detection is node-based: a transition is considered a U-turn when the next directed arc returns to the tail node of the current directed arc.

When U-turns are allowed and graph compression is enabled, nodes where U-turns could occur (nodes with bidirectional links) are preserved during compression to maintain the correct turn possibilities.

4.3.5 Usage

Managing Turn Restrictions

Turn restrictions use the same table API as links and nodes. `get()` returns an immutable record, `data` returns a GeoDataFrame, and `len(turns)` counts records. Missing keys raise `ValueError` unless `get()` is given a default.

Supply `modes` on insert. Database triggers reject missing, empty, unknown or repeated mode IDs, and overlapping restrictions for the same turn. These errors raise `sqlite3.IntegrityError`. Modes are stored in the supplied order.

Updates leave omitted fields unchanged; `penalty=None` prohibits the turn. Use `project.transaction()` to group writes.

```
>>> project = create_example(project_path)

# Sioux Falls free_flow_time is in minutes, so convert it to seconds for these␣
↪examples.
>>> project.tables.links.fields.add("travel_time_seconds", "Free-flow travel time in␣
↪seconds", "REAL")
>>> with project.db_connection as conn:
...     _ = conn.execute("UPDATE links SET travel_time_seconds = free_flow_time * 60.0
↪")

# Get the turn restrictions object
>>> turns = project.tables.turn_restrictions

# Add a prohibited turn (penalty=None means prohibited)
>>> restriction_id = turns.insert(from_node=1, via_node=2, to_node=6, modes="c")

# Add a turn with a 30-second penalty
>>> other_id = turns.insert(from_node=3, via_node=4, to_node=5, penalty=30.0, modes="c
↪")

# Change only the penalty
>>> turns.update(restriction_id, penalty=10.0)

# Get all restrictions, including their generated geometry
>>> df = turns.data
```

(continues on next page)

(continued from previous page)

```
# Remove a specific restriction
>>> turns.delete(restriction_id)

# Clear all restrictions
>>> turns.clear_restrictions()
1
```

Bulk Loading from DataFrame

Use `insert_from()` to load turns from a DataFrame. It returns the inserted IDs. Use `update_from()` with a `restriction_id` column to change existing turns. Include a `modes` column when inserting. Both methods normalise penalties just like single-record writes and leave mode validation to the database. They roll back the whole batch if a row fails.

```
>>> import pandas as pd

>>> restrictions_df = pd.DataFrame({
...     'from_node': [1, 2, 3],
...     'via_node': [2, 6, 4],
...     'to_node': [6, 5, 5],
...     'modes': ['c', 'c', 'c'],
...     'penalty': [None, 15.0, 30.0],  # None or +inf = prohibited
... })

>>> restriction_ids = turns.insert_from(restrictions_df)
```

Setting Global U-Turn Permission

The U-turn setting is managed through the project's `about` table:

```
>>> project.about.update("allow_uturns", infovalue="1")  # Allow U-turns

# Rebuild network and build graphs to apply the change
>>> network = project.road.build_network()
>>> graph = network.build_graph('c', cost='travel_time_seconds')
```

Building Graphs with Turn Restrictions

When you build a network snapshot, turn restrictions are loaded from the database.

```
>>> network = project.road.build_network()
>>> graph = network.build_graph('c', cost='travel_time_seconds')

# Check if a graph has turn restrictions
>>> print(graph.has_turn_restrictions)
True
```

Manual Turn Restriction Setup

You can also supply turn restrictions when building a new graph. An existing graph's topology and turn configuration cannot be changed in place:

```
>>> import pandas as pd

>>> links = project.tables.links.data.set_index("link_id")
>>> turn_df = pd.DataFrame({
...     "from_node": [int(links.loc[7, "a_node"])],
...     "via_node": [int(links.loc[7, "b_node"])],
...     "to_node": [int(links.loc[36, "b_node"])],
...     "penalty": [20.0],
... })

# Build a new graph with these turn restrictions
>>> graph = network.build_graph('c', cost='travel_time_seconds', turn_
↪restrictions=turn_df, allow_uturns=False)

# Build a new graph without turn restrictions, overriding the snapshot's turns with␣
↪an empty table
>>> graph = network.build_graph('c', cost='travel_time_seconds', turn_
↪restrictions=turn_df.iloc[:0])
```

To change the source network's turn configuration for subsequent graphs or assignments, use `dataclasses.replace(network, turn_restrictions=turn_df)`. Network configuration fields cannot be assigned in place.

4.3.6 Traffic Assignment with Turn Restrictions

Turn restrictions are automatically considered during traffic assignment when they are present in the source network. The assignment builds its graphs internally. The turn penalties are applied as additional time costs during the equilibration process.

```
>>> from aequilibrae.traffic import TrafficAssignment, TrafficClass
>>> from aequilibrae.traffic.vdf import bpr

>>> network = project.road.build_network()
>>> demand_matrix = project.matrices.get_matrix("demand.omx", subset=["matrix"])
>>> traffic_class = TrafficClass(
...     mode='c',
...     cost='travel_time_seconds',
...     demand=demand_matrix,
...     name='car',
...     skimming=('travel_time_seconds',),
... )

>>> assignment = TrafficAssignment(
...     network=network,
...     classes=[traffic_class],
...     time_field='travel_time_seconds',
...     capacity_field='capacity',
...     vdf=bpr,
...     algorithm='bfw',
...     name_mapping={'alpha': 0.15, 'beta': 4.0},
```

(continues on next page)

(continued from previous page)

```
... )
>>> _ = assignment.execute()
```

4.3.7 Route Choice with Turn Restrictions

Turn restrictions and turn penalties are automatically considered during route choice when they are present in the graph. They are included in generated route costs and path-size logit calculations.

For imported routes, `RouteChoice.recompute_psl()` validates origin and destination endpoints, link connectivity and turn restrictions before recalculating costs, overlap and probabilities. For assignment from imported routes, pass `recompute_psl=True` to `execute_from_pandas()` or `execute_from_path_files()` to perform the same validation and recalculation.

4.3.8 Performance Considerations

The arc-based Dijkstra algorithm required for turn restrictions has higher memory requirements than the standard node-based algorithm (arrays are indexed by number of links rather than number of nodes). For this reason:

1. Arc-based path finding is only activated when turn restrictions are present
2. If no turn restrictions exist, the standard (faster) algorithm is used
3. Graph compression is adjusted to preserve nodes where U-turns could occur

4.3.9 Turn Penalty Units

Turn penalties must use the same units as the graph's cost function. For traffic assignment, the class cost and assignment `time_field` must also use the same time units. The examples above convert the sample model's `free_flow_time` from minutes to `travel_time_seconds`, so their turn penalties are in seconds.

4.3.10 Limitations

- A* path finding can be used together with turn restrictions.
- Turn restrictions are defined by 3-node movement sequences (no lane-level control)
- Complex turn restrictions involving multiple via-nodes are not supported

4.4 Examples

4.4.1 Network Manipulation

Create project from Overture Maps (or OpenStreetMap)

In this example we show how to create an empty project and populate it with a network. Overture Maps is the recommended default source; importing from OpenStreetMap is also supported and is shown as commented-out code that you can switch to.

Install the optional dependencies first:

```
pip install aequilibrae[create]
```

References

- *Importing from Overture Maps*

- *Importing from OpenStreetMap*

See also

Several functions, methods, classes and modules are used in this example:

- `project.tables.importer.overture()` through *aequilibrae.Project.tables*
- `project.tables.importer.osm()` through *aequilibrae.Project.tables*

```
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

from shapely.geometry import box

from aequilibrae import Project
```

Create an empty project on an arbitrary folder.

```
fldr = join(gettempdir(), uuid4().hex)

project = Project.create(fldr)
```

Both Overture and OSM imports require an EPSG:4326 polygon.

```
model_area = box(-112.185, 36.59, -112.179, 36.60)
```

Import from Overture Maps (the recommended default). The importer always uses the latest release advertised by Overture's STAC catalog and records that release in the project's `about` table.

```
project.tables.importer.overture(model_area=model_area, modes=("car", "walk"),␣
↪simplify=False)
```

To import from OpenStreetMap instead, comment out the Overture call above and uncomment the block below. OSM data is fetched from an Overpass endpoint; for large areas you should point at your own/regional Overpass server and increase the request timeout (the "sleep"/wait budget for Overpass) to avoid throttling. Both are set on the project parameters before importing:

```
params = project.project_parameters
params.parameters["osm"]["overpass_endpoint"] = "https://overpass-api.de/api"  # your␣
↪own/regional server
params.parameters["osm"]["timeout"] = 180  # seconds to wait for Overpass before␣
↪giving up
params.write_back()

project.tables.importer.osm(model_area=model_area, modes=("car", "walk"),␣
↪simplify=False)
# or import a whole place by name:
# project.tables.importer.osm(place_name="Nauru")
# or import from a local extract:
# project.tables.importer.osm(pbf_path="path/to/extract.osm.pbf")
```

```
links = project.tables.links.data
links.explore(color="blue", style_kwds={"weight": 2}, tooltip="link_type")
```

```
project.close()
```

Editing network geometry: Nodes

In this example, we show how to mode a node in the network and look into what happens to the links.

References

- *Node layer changes and expected behavior*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.nodes()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.examples import create_example
from shapely.geometry import Point
import matplotlib.pyplot as plt
```

```
# We create the example project inside our temp folder.
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

Let's move node one from the upper left corner of the image above, a bit to the left and to the bottom.

```
# We also add the node we want to move.
all_nodes = project.tables.nodes
links = project.tables.links
node = all_nodes.get(1)
new_geo = Point(node.geometry.x + 0.02, node.geometry.y - 0.02)
all_nodes.update(node.node_id, geometry=new_geo)
node = all_nodes.get(node.node_id)
```

If you want to show the path in Python.

We do NOT recommend this, though…. It is very slow for real networks.

Let's access our links data using a context manager instead of directly accessing the DataFrame.

```
with project.db_connection as conn:
    link_ids = conn.execute("Select link_id from links;").fetchall()
```

(continues on next page)

(continued from previous page)

```
for lid in link_ids:
    geo = links.get(lid[0]).geometry
    plt.plot(*geo.xy, color="blue")

plt.plot(*node.geometry.xy, "o", color="black")

plt.show()
```

Did you notice the links are matching the node? Look at the original network and see how it used to look like.

```
project.close()
```

Editing network geometry: Links

In this example, we move a link extremity from one point to another and see what happens to the network.

References

- *Link layer changes and expected behavior*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.links()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.examples import create_example
from shapely.geometry import LineString, Point
import matplotlib.pyplot as plt
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

```
all_nodes = project.tables.nodes
links = project.tables.links
```

Let's move node one from the upper left corner of the image above, a bit to the left and to the bottom

```
# We edit the link that goes from node 1 to node 2
link = links.get(1)
node = all_nodes.get(1)
new_extremity = Point(node.geometry.x + 0.02, node.geometry.y - 0.02)
links.update(1, geometry=LineString([node.geometry, new_extremity]))
```

(continues on next page)

(continued from previous page)

```
# and the link that goes from node 2 to node 1
link = links.get(3)
node2 = all_nodes.get(2)
links.update(3, geometry=LineString([new_extremity, node2.geometry]))
```

Because each link is unidirectional, you can no longer go from node 1 to node 2, obviously.

We do NOT recommend this, though…. It is very slow for real networks.

```
link_ids = links.data["link_id"].values.tolist()
for lid in link_ids:
    geo = links.get(lid).geometry
    plt.plot(*geo.xy, color="blue")

node_ids = all_nodes.data["node_id"].values.tolist()
for nid in node_ids:
    geo = all_nodes.get(nid).geometry
    plt.plot(*geo.xy, "o", color="black")

plt.show()
```

Now look at the network and how it used to be.

```
project.close()
```

Exporting network to GMNS

In this example, we export a simple network to GMNS format. The source AequilibraE model used as input for this is the result of the import process (`importer.gmns()`) using the GMNS example of Arlington Signals, which can be found in the GMNS repository on GitHub: https://github.com/zephyr-data-specs/GMNS

References

- *Exporting AequilibraE model to GMNS format*

See also

Several functions, methods, classes and modules are used in this example:

- `project.tables.export_to_gmns()` through *aequilibrae.Project.tables*

```
# Imports
from uuid import uuid4
import os
from tempfile import gettempdir

from aequilibrae.examples import create_example
import folium
import geopandas as gpd
import pandas as pd
```

```
# We load the example project inside a temp folder
fldr = os.path.join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

We export the network to CSV files in GMNS format, that will be saved inside the project folder

```
output_fldr = os.path.join(gettempdir(), uuid4().hex)
if not os.path.exists(output_fldr):
    os.mkdir(output_fldr)

project.tables.export_to_gmns(path=output_fldr)
```

Now, let's plot a map. This map can be compared with the images of the README.md file located in this example repository on GitHub: https://github.com/zephyr-data-specs/GMNS/blob/develop/examples/Arlington_Signals/README.md

```
links = pd.read_csv(os.path.join(output_fldr, "link.csv"))
nodes = pd.read_csv(os.path.join(output_fldr, "node.csv"))
```

We turn the links and nodes DataFrames into GeoDataFrames so we can plot them more easily.

```
links = gpd.GeoDataFrame(links, geometry=gpd.GeoSeries.from_wkt(links["geometry"]),␣
↪crs=4326)
nodes = gpd.GeoDataFrame(nodes, geometry=gpd.GeoSeries.from_xy(nodes["x_coord"],␣
↪nodes["y_coord"]), crs=4326)
```

Let's plot our map!

```
map = links.explore(color="black", style_kwds={"weight": 2}, tool_tip="link_type",␣
↪name="links")
map = nodes.explore(m=map, color="red", style_kwds={"radius": 5, "fillOpacity": 1.0},␣
↪name="nodes")

folium.LayerControl().add_to(map) # Add a layer control button to our map
map
```

```
project.close()
```

Exploring the network on a notebook

In this example, we show how to use Folium to plot a network for different modes.

We will need Folium for this example, and we will focus on creating a layer for each mode in the network, a layer for all links and a layer for all nodes.

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join

from aequilibrae.examples import create_example
import folium
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)

# Let's use the Nauru example project for display
project = create_example(fldr, "nauru")
```

We grab all the links data as a geopandas GeoDataFrame so we can process it easier

```
links = project.tables.links.data
nodes = project.tables.nodes.data
```

And if you want to take a quick look in your GeoDataFrames, you can plot it!

```
# links.plot()
```

Let's create copies of our link layers for each mode

```
bike = links[links["modes"].str.contains("b")]
car = links[links["modes"].str.contains("c")]
transit = links[links["modes"].str.contains("t")]
walk = links[links["modes"].str.contains("w")]
```

And plot out data!

```
map = links.explore(color="gray", style_kwds={"weight": 2}, popup="link_id", tooltip=
↪"modes", name="network_links")
map = nodes.explore(m=map, color="black", style_kwds={"radius": 5, "fillOpacity": 1.0}
↪, name="network_nodes")

map = walk.explore(m=map, color="green", style_kwds={"weight": 3}, popup="link_id",␣
↪tooltip="modes", name="walk")
map = bike.explore(m=map, color="green", style_kwds={"weight": 3}, popup="link_id",␣
↪tooltip="modes", name="bike")
map = car.explore(m=map, color="red", style_kwds={"weight": 3}, popup="link_id",␣
↪tooltip="modes", name="car")
map = transit.explore(m=map, color="yellow", style_kwds={"weight": 3}, popup="link_id
↪", tooltip="modes", name="transit")

folium.LayerControl().add_to(map)
map
```

```
project.close()
```

Editing network geometry: Splitting link

In this example, we split a link right in the middle, while keeping all fields in the database equal. Distance is proportionally computed automatically in the database.

References

- *Link layer changes and expected behavior*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.links()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.examples import create_example
from shapely.ops import substring
import matplotlib.pyplot as plt
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

We will split link 37 right in the middle. Let's get the link and check its length.

```
links = project.tables.links
all_nodes = project.tables.nodes

link = links.get(37)
print(link.distance)
```

The idea is basically to copy a link and allocate the appropriate geometries to split the geometry we use Shapely's substring.

```
new_link_id = links.copy(37)

first_geometry = substring(link.geometry, 0, 0.5, normalized=True)
second_geometry = substring(link.geometry, 0.5, 1, normalized=True)

links.update(37, geometry=first_geometry)
links.update(new_link_id, geometry=second_geometry)
```

Retrieve the updated immutable records from the database.

```
link = links.get(37)
new_link = links.get(new_link_id)
print(link.distance, new_link.distance)
```

```
# We can plot the two links only
plt.clf()
plt.plot(*link.geometry.xy, color="blue")
plt.plot(*new_link.geometry.xy, color="blue")

for node in [link.a_node, link.b_node, new_link.b_node]:
    geo = all_nodes.get(node).geometry
    plt.plot(*geo.xy, "o", color="black")
plt.show()
```

```
# Or we plot the entire network
plt.clf()

link_ids = links.data["link_id"].values.tolist()
for lid in link_ids:
    geo = links.get(lid).geometry
    plt.plot(*geo.xy, color="blue")

node_ids = all_nodes.data["node_id"].values.tolist()
for nid in node_ids:
    geo = all_nodes.get(nid).geometry
    plt.plot(*geo.xy, "o", color="black")

plt.show()
```

```
project.close()
```

Create centroid connectors

We use Coquimbo example to show how we can create centroids and connect them to the existing network efficiently.

We use Folium to visualize the resulting network.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.zones()`

```
# Imports
import folium
from uuid import uuid4
from tempfile import gettempdir
from os.path import join

from aequilibrae.examples import create_example
```

```
# Let's create an arbitrary project folder
fldr = join(gettempdir(), uuid4().hex)

# And create our Coquimbo project
project = create_example(fldr, "coquimbo")
```

As Coquimbo already has centroids and centroid connectors, we should remove them for the sake of this example

```
with project.db_connection as conn:
    conn.execute("DELETE FROM links WHERE name LIKE 'centroid connector%';")
    conn.execute("DELETE FROM nodes WHERE is_centroid=1;")
    conn.commit()

    centroids = conn.execute("SELECT COUNT(node_id) FROM nodes WHERE is_centroid=1;").
↪fetchone()[0]
```

If you want to make sure your deletion process has worked, you can check the number of centroids!

```
print("Current number of centroids: ", centroids)
```

```
zones = project.tables.zones
```

This centroid connector creation is effective because it uses the existing zone layer to create the centroids and connect them to the existing network.

First thing to do is add the centroids to all zones that doesn't have a centroid at the geographic centroid of the zone, using `add_centroids()`, which has a `robust` argument set to `True` as default. This means that it will automatically move the centroid location around to avoid conflicts with existing nodes.

```
zones.add_centroids()
```

Let's connect the mode `c`, that stands for car.

```
mode = "c"
```

Then we connect the centroid to the network, by selecting the desired mode, the number of connectors, the allowed link types for connection, and if one wants to allow the connection to links in other zones. By setting `limit_to_zone=False`, we allow the centroid of one zone to be connected to a link outside the zone itself.

```
zones.connect_mode(mode_id=mode, connectors=1, limit_to_zone=False)
```

It is possible to repeat the process above for a different mode, with different link type, number of connectors and connection allowance, as desired.

Finally, let's plot our data!

```
links = project.tables.links.data
centroids = links[links["link_type"] == "centroid_connector"]
links = links[links["link_type"] != "centroid_connector"]

nodes = project.tables.nodes.data
nodes = nodes[nodes["is_centroid"] == 1]
```

```
map = folium.Map(location=[-29.9568, -71.3456], zoom_start=14)
zones.data.explore(m=map, color="blue", style_kwds={"fillOpacity": 0.05}, name="zones
↪")
centroids.explore(m=map, color="black", style_kwds={"weight": 2.5}, name="centroid_
↪connector")
links.explore(m=map, color="gray", style_kwds={"weight": 1}, name="links")
nodes.explore(m=map, color="red", style_kwds={"radius": 3, "fillOpacity": 1.0}, name=
↪"centroid")

folium.LayerControl().add_to(map)

map
```

```
project.close()
```

Create project from GMNS

In this example, we import a simple network in GMNS format. The source files of this network are publicly available in the GMNS GitHub repository itself.

References

- *Importing from files in GMNS format*

See also

Several functions, methods, classes and modules are used in this example:

- `project.tables.importer.gmns()` through *aequilibrae.Project.tables*

```
# Imports
from uuid import uuid4
from os.path import join
from tempfile import gettempdir

from aequilibrae import Project
import folium
```

```
# We load the example file from the GMNS GitHub repository
gmns_repository = "https://raw.githubusercontent.com/zephyr-data-specs/GMNS"
example_folder = f"{gmns_repository}/main/examples/Arlington_Signals"

link_file = f"{example_folder}/link.csv"
node_file = f"{example_folder}/node.csv"
use_group_file = f"{example_folder}/use_group.csv"
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = Project.create(fldr)
```

In this cell, we modify the AequilibraE parameters.yml file so it contains additional fields to be read in the GMNS link and/or node tables. Remember to always keep the "required" key set to False, since we are adding a non-required field.

```
new_link_fields = {
    "bridge": {"description": "bridge flag", "type": "text", "required": False},
    "tunnel": {"description": "tunnel flag", "type": "text", "required": False},
}
new_node_fields = {
    "port": {"description": "port flag", "type": "text", "required": False},
    "hospital": {"description": "hospital flag", "type": "text", "required": False},
}

par = project.project_parameters
par.parameters["network"]["gmns"]["link"]["fields"].update(new_link_fields)
par.parameters["network"]["gmns"]["node"]["fields"].update(new_node_fields)
```

(continues on next page)

(continued from previous page)

```
par.write_back()
```

As it is specified that the geometries are in the coordinate system EPSG:32619, which is different than the system supported by AequilibraE (EPSG:4326), we inform the srid in the method call:

```
project.tables.importer.gmns(
    link_file_path=link_file, node_file_path=node_file, use_group_path=use_group_file,
↪ srid=32619
)
```

Now, let's plot a map. This map can be compared with the images of the README.md file located in this example repository on GitHub: https://github.com/zephyr-data-specs/GMNS/blob/develop/examples/Arlington_Signals/README.md

```
links = project.tables.links.data
nodes = project.tables.nodes.data
```

```
map = links.explore(color="black", style_kwds={"weight": 2}, tool_tip="link_type",␣
↪name="links")
map = nodes.explore(m=map, color="red", style_kwds={"radius": 5, "fillOpacity": 1.0},␣
↪name="nodes")

folium.LayerControl().add_to(map) # Add a layer control button to our map
map
```

```
project.close()
```

Finding disconnected links

In this example, we show how to find disconnected links in an AequilibraE network.

We use the Nauru example to find disconnected links.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.results.path_results()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from datetime import datetime
import pandas as pd
import numpy as np
from aequilibrae.examples import create_example
from aequilibrae.routing import PathResults
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)
```

(continues on next page)

(continued from previous page)

```
# Let's use the Nauru example project for display
project = create_example(fldr, "nauru")

# Let's analyse the mode car or 'c' in our model
mode = "c"
```

We can create an uncompressed graph without any centroids or changes to the project.

```
# We get the project nodes for reporting and load an independent network snapshot.
nodes = project.tables.nodes
network = project.road.build_network()
# No centroid changes need reverting: the project nodes have not been modified.
```

We set the graph for computation

```
graph = network.build_graph(mode=mode, cost="distance", skimming=["distance"], block_
↪centroid_flows=False)
```

Get the nodes that are part of the car network

```
missing_nodes = nodes.data.query("modes.str.contains(@mode)")["node_id"].values
```

Now we can compute all the path islands we have

```
islands = []
idx_islands = 0

while missing_nodes.shape[0] >= 2:
    print(datetime.now().strftime("%H:%M:%S"), f" - Computing island: {idx_islands}")
    res = PathResults(graph, missing_nodes[0], missing_nodes[1])
    connected = res.node_ids[res.terminal_states != res.search_results.sentinel]
    connected = np.intersect1d(missing_nodes, connected)
    missing_nodes = np.setdiff1d(missing_nodes, connected)
    print(f"    Nodes to find: {missing_nodes.shape[0]:,}")
    df = pd.DataFrame({"node_id": connected, "island": idx_islands})
    islands.append(df)
    idx_islands += 1

print(f"\nWe found {idx_islands} islands")
```

Let's consolidate everything into a single DataFrame

```
islands = pd.concat(islands)

# And save to disk alongside our model
islands.to_csv(join(fldr, "island_outputs_complete.csv"), index=False)
```

If you join the `node_id` field in the CSV file generated above with the `a_node` or `b_node` fields in the links table, you will have the corresponding links in each disjoint island found.

```
project.close()
```

Create project from a link layer

In this example, we show how to create an empty project and populate it with a network coming from a link layer we load from a text file. It can easily be replaced with a different form of loading the data (GeoPandas, for example).

We use Folium to visualize the resulting network.

References

- *Project Data Components*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.links()`
- `aequilibrae.project.network.nodes()`
- `aequilibrae.project.network.modes()`
- `aequilibrae.project.network.link_types()`

```
# Imports
import urllib.request
from os.path import join
from string import ascii_lowercase
from tempfile import gettempdir
from uuid import uuid4

import geopandas as gpd
import pandas as pd
from shapely.wkt import loads as load_wkt

from aequilibrae import Project
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)

project = Project.create(fldr)
```

Now we obtain the link data for our example (in this case from a link layer we will download from the AequilibraE website). With data, we load it on Pandas

```
dest_path = join(fldr, "queluz.csv")
urllib.request.urlretrieve("https://aequilibrae.com/data/queluz.csv", dest_path)

df = pd.read_csv(dest_path)
df = gpd.GeoDataFrame(df.drop(columns=["fid", "WKT"]), geometry=gpd.GeoSeries.from_
↪wkt(df["WKT"]))
```

Let's see if we have to add new link_types to the model before we add links The links we have in the data are:

```
link_types = df.link_type.unique()
```

And the existing link types are

```
lt = project.tables.link_types
existing_types = [link_type.link_type for link_type in lt]
```

We could also get it directly from the project database

```
# with project.db_connection as conn:
#     existing_types = [x[0] for x in conn.execute('Select link_type from link_types
↪')]
```

We add the link types that do not exist yet. The trickier part is to choose a unique link type ID for each link type. You might want to tailor the link type for your use, but here we get letters in alphabetical order.

```
types_to_add = [ltype for ltype in link_types if ltype not in existing_types]
for i, ltype in enumerate(types_to_add):
    lt.insert(
        link_type_id=ascii_lowercase[i],
        link_type=ltype,
        # description='Your custom description here if you have one',
    )
```

We need to use a similar process for modes

```
md = project.tables.modes
existing_modes = {mode.mode_id: mode.mode_name for mode in md}
```

Now let's see the modes we have in the network that we DON'T have already in the model.

We get all the unique mode combinations and merge them into a single string

```
all_variations_string = "".join(df.modes.unique())

# We then get all the unique modes in that string above
all_modes = set(all_variations_string)

# This would all fit nicely in a single line of code, btw. Try it!
```

Now let's add any new mode to the project

```
modes_to_add = [mode for mode in all_modes if mode not in existing_modes]
for mode_id in modes_to_add:
    # You would need to figure out the right name for each one, but this will do
    md.insert(mode_id=mode_id, mode_name=f"Mode_from_original_data_{mode_id}")
    # description='Your custom description here if you have one'
```

We cannot use the existing link_id, so we create a new field to not loose this information

```
links = project.tables.links
link_data = links.fields

# Create the field and add a good description for it
link_data.add("source_id", "link_id from the data source")
```

We can now add all links to the project!

```
links.insert_from(df.assign(source_id=df["link_id"]))
```

We grab all the links data as a geopandas GeoDataFrame so we can process it easier

```
links = project.tables.links.data
```

Let's plot our network!

```
links.explore(color="blue", style_kwds={"weight": 2}, tooltip="link_type")
```

```
project.close()
```

Network simplifier

In this example we use Nauru network to show how one can simplify the network, merging short links into longer ones or turning links into nodes, and saving these changes into the project.

We use Folium to visualize the resulting network.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.tools.network_simplifier()`

```
# Imports
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import branca
import folium

from aequilibrae.project import NetworkSimplifier
from aequilibrae.examples import create_example
```

Let's use the Nauru example project for display

```
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "nauru")
```

To simplify the network, we need to create a compact graph, which requires centroids. As Nauru doesn't have any centroids, we use an arbitrary node as a graph-only centroid.

```
network = project.road.build_network()
centroids = network.centroids
if centroids is None:
    centroids = network.mapping.node_ids[:1]
```

```
# Let's analyse the mode car or 'c' in our model
mode = "c"
```

```
# Let's set the graph for computation
graph = network.build_graph(
    mode, "distance", centroids=centroids, skimming=["distance"], compact=True, block_
↪centroid_flows=False
)
```

```
# No centroid changes need reverting: the graph-only centroid was not written to the␣
↪project.
```

We check the number of links and nodes our project has initially.

```
links_before = project.tables.links.data
nodes_before = project.tables.nodes.data

print("This project initially has {} links and {} nodes".format(links_before.shape[0],
↪ nodes_before.shape[0]))
```

Let's call the `NetworkSimplifier` class. Any changes made to the database using this class are permanent. Make sure you have a backup if necessary.

```
net = NetworkSimplifier(project)
```

When we choose to simplify the network, we pass a graph object to the function, and the output of this operation is

```
net.simplify(graph)
```

Let's plot the previous and actual networks!

```
links_after = net.tables.links.data
nodes_after = net.tables.nodes.data
```

```
fig = branca.element.Figure()

subplot1 = fig.add_subplot(1, 2, 1)
subplot2 = fig.add_subplot(1, 2, 2)

map1 = folium.Map(location=[-0.508371, 166.931142], zoom_start=17)
map1 = links_before.explore(m=map1, color="black", style_kwds={"weight": 2}, name=
↪"links_before")
map1 = nodes_before.explore(m=map1, color="red", style_kwds={"radius": 3, "fillOpacity
↪": 1.0}, name="nodes_before")
folium.LayerControl().add_to(map1)

map2 = folium.Map(location=[-0.508371, 166.931142], zoom_start=17)
map2 = links_after.explore(m=map2, color="black", style_kwds={"weight": 2}, name=
↪"links_after")
map2 = nodes_after.explore(m=map2, color="blue", style_kwds={"radius": 3, "fillOpacity
↪": 1.0}, name="nodes_after")
folium.LayerControl().add_to(map2)

subplot1.add_child(map1)
subplot2.add_child(map2)
```

(continues on next page)

(continued from previous page)

```
fig
```

Differently we can simplify the network by collapsing links into nodes. Notice that this operation modifies the network in the neighborhood.

```
net.collapse_links_into_nodes([903])
```

Let's plot the network once again and check the modifications!

```
links_after = net.tables.links.data
nodes_after = net.tables.nodes.data
```

```
fig = branca.element.Figure()

subplot1 = fig.add_subplot(1, 2, 1)
subplot2 = fig.add_subplot(1, 2, 2)

map1 = folium.Map(location=[-0.509363, 166.928563], zoom_start=18)
map1 = links_before.explore(m=map1, color="black", style_kwds={"weight": 2}, name=
↪"links_before")
map1 = nodes_before.explore(m=map1, color="red", style_kwds={"radius": 3, "fillOpacity
↪": 1.0}, name="nodes_before")
folium.LayerControl().add_to(map1)

map2 = folium.Map(location=[-0.509363, 166.928563], zoom_start=18)
map2 = links_after.explore(m=map2, color="black", style_kwds={"weight": 2}, name=
↪"links_after")
map2 = nodes_after.explore(m=map2, color="blue", style_kwds={"radius": 3, "fillOpacity
↪": 1.0}, name="nodes_after")
folium.LayerControl().add_to(map2)

subplot1.add_child(map1)
subplot2.add_child(map2)

fig
```

```
project.close()
```

CHAPTER

FIVE

DISTRIBUTION PROCEDURES

In the context of transportation modeling, a distribution model tries to estimate the number of trips in each of the matrix cells on the basis of any information available[1].

In the following sections, we present the classes that comprise AequilibraE's distribution module, as well as present a benchmark validation for the IPF procedure and some usage examples.

5.1 Distribution procedure classes

AequilibraE's distribution module comprises three different classes: `GravityApplication`, `GravityCalibration`, and `Ipf`.

5.1.1 `GravityApplication`

This class, as its own name explains, applies a synthetic gravity model, using one of the available deterrence functions: `EXPO`, `POWER`, or `GAMMA`. It requires some parameters, such as:

- Synthetic gravity model (which is an instance of `SyntheticGravityModel`)
- Impedance matrix (`AequilibraeMatrix`);
- Vector (`Pandas.DataFrame`) with data for row and column totals;
- Row and column fields, which are the names of the fields that contain the data for row and column totals.

The synthetic gravity model instance can be either created or loaded, if you have already calibrated a model.

Please check other arguments and parameters that are passed to `GravityApplication` in its documentation.

See also

- ***aequilibrae.distribution.synthetic_gravity_model.SyntheticGravityModel()***
  Function documentation
- ***aequilibrae.distribution.gravity_application.GravityApplication()***
  Function documentation

5.1.2 `GravityCalibration`

Calibrate the model consists in checking if all the parameters set are appropriate. This class, as its own name explains, calibrates a traditional gravity model, using one of the available deterrence functions: `EXPO`, `POWER`, or `GAMMA`. It requires some arguments such as:

[1] Ortúzar, J. de D. and Willumsen, L.G. (2011) Modelling transport. 4th edition. Chichester: Wiley.

- Matrix containing the base trips (`AequilibraeMatrix`);
- Impedance matrix (`AequilibraeMatrix`);
- Deterrence function name.

Please check other arguments and parameters that are passed to `GravityCalibration` in its documentation.

See also

- ***aequilibrae.distribution.gravity_calibration.GravityCalibration()***
  Function documentation

5.1.3 `Ipf`

IPF is an acronym for Iterative Proportional Fitting, also known as Fratar or Furness. The IPF procedure is used to "distribute" future trips based on a growth factor. The procedure can be run with or without an AequilibraE model, with the latter using one of AequilibraE matrices or NumPy arrays as data input.

In the following section, we present the validation of the results produced with AequilibraE's IPF.

See also

- ***aequilibrae.distribution.ipf.Ipf()***
  Function documentation
- ***Running IPF without an AequilibraE model***
  Usage example
- ***Running IPF with NumPy array***
  Usage example

5.2 IPF Performance

The use of iterative proportional fitting (IPF) is quite common on processes involving doubly-constraining matrices, such as synthetic gravity models and fractional split models (aggregate destination-choice models).

As this is a commonly used algorithm, we have implemented it in Cython, where we can take full advantage of multi-core CPUs. We have also implemented the ability of using both 32-bit and 64-bit floating-point seed matrices, which has direct impact on cache use and consequently computational performance.

In this section, we compare the runtime of AequilibraE's current implementation of IPF, with a general IPF algorithm written in pure Python, available here.

The figure below compares AequilibraE's IPF runtime with one core with the benchmark Python code. From the figure below, we can notice that the runtimes were practically the same for the instances with 1,000 zones or less. As the number of zones increases, AequilibraE demonstrated to be slightly faster than the benchmark python code, while applying IPF to a 32-bit NumPy array (np.float32) was significantly faster.

It's worth mentioning that the user can set up a threshold for AequilibraE's IPF function, as well as use more than one core to speed up the fitting process.

As IPF is an embarrassingly-parallel workload, it is more relevant to look at the performance of the AequilibraE implementations, starting by comparing the implementation performance for inputs in 32 vs 64 bits using 32 threads.

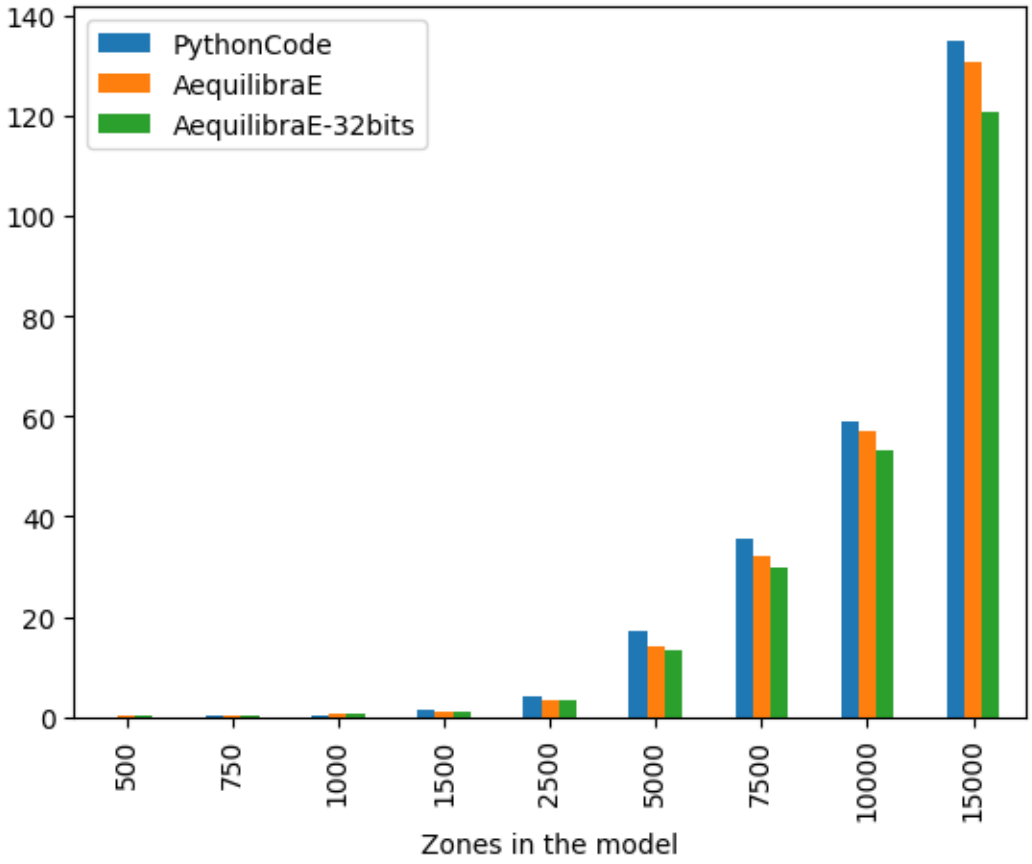


Fig. 1: AequilibraE's IPF runtime

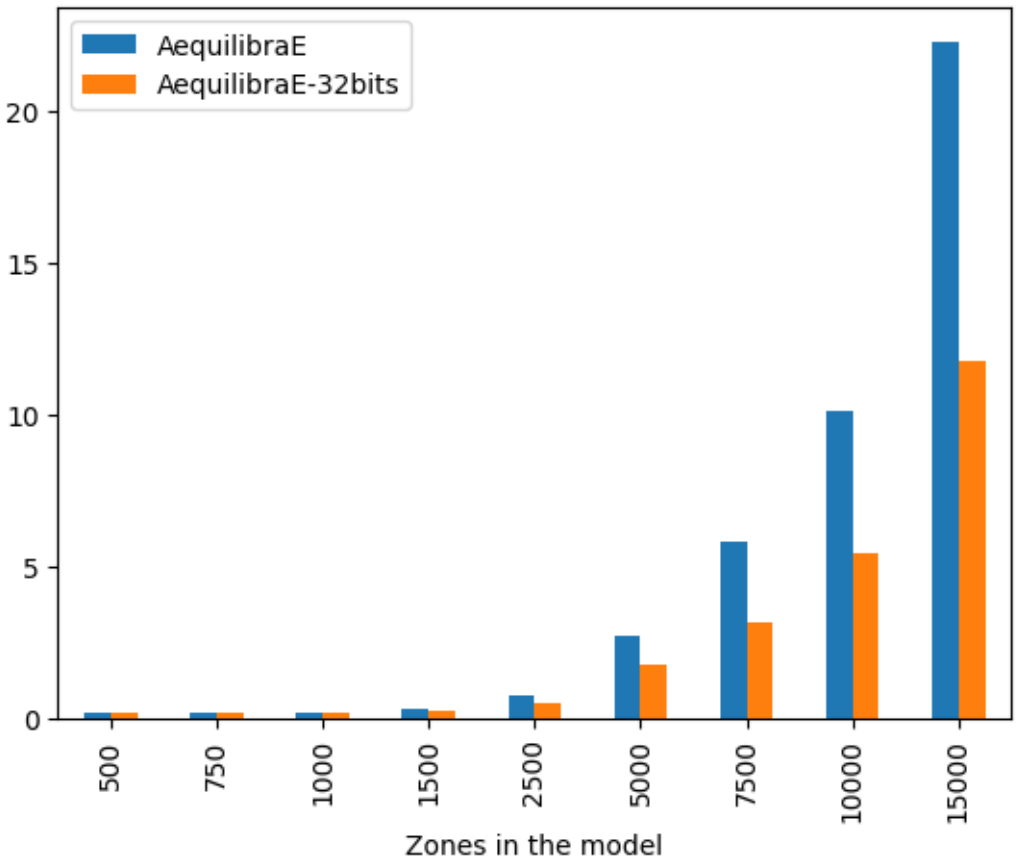


Fig. 2: AequilibraE's IPF runtime 32 vs 64 bits

The difference is staggering, with the 32-bit implementation being twice as fast as the 64-bit one for large matrices. It is also worth noting that differences in results between the outputs between these two versions are incredibly small (RMSE < 1.1e-10), and therefore unlikely to be relevant in most applications.

We can also look at performance gain across matrix sizes and number of cores, and it becomes clear that the 32-bit version scales significantly better than its 64-bit counterpart, showing significant performance gains up to 16 threads, while the latter stops showing much improvement beyond 8 threads, likely due to limitations on cache size.

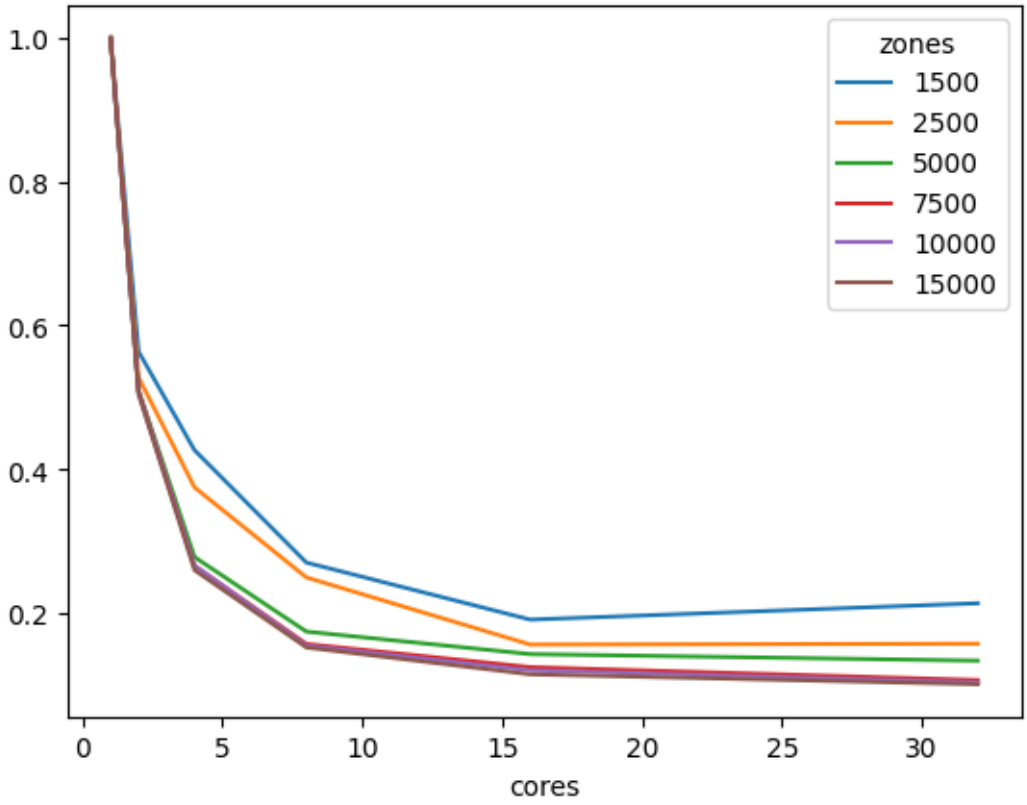


Fig. 3: number of cores used in IPF for 64 bit matrices

In conclusion, AequilibraE's IPF implementation is over 11 times faster than its pure Python counterpart for large matrices on a workstation, largely due to the use of Cython and multi-threading, but also due to the use of a 32-bit version of the algorithm.

These tests were run on a Threadripper 3970x (released in 2019) workstation with 32 cores (64 threads) @ 3.7 GHz and 256 Gb of RAM. The code is provided below for reference.

5.2.1 Reference code

```
from copy import deepcopy
from time import perf_counter
import numpy as np
import pandas as pd
from aequilibrae.distribution.cython.ipf_core import ipf_core
from tqdm import tqdm
```

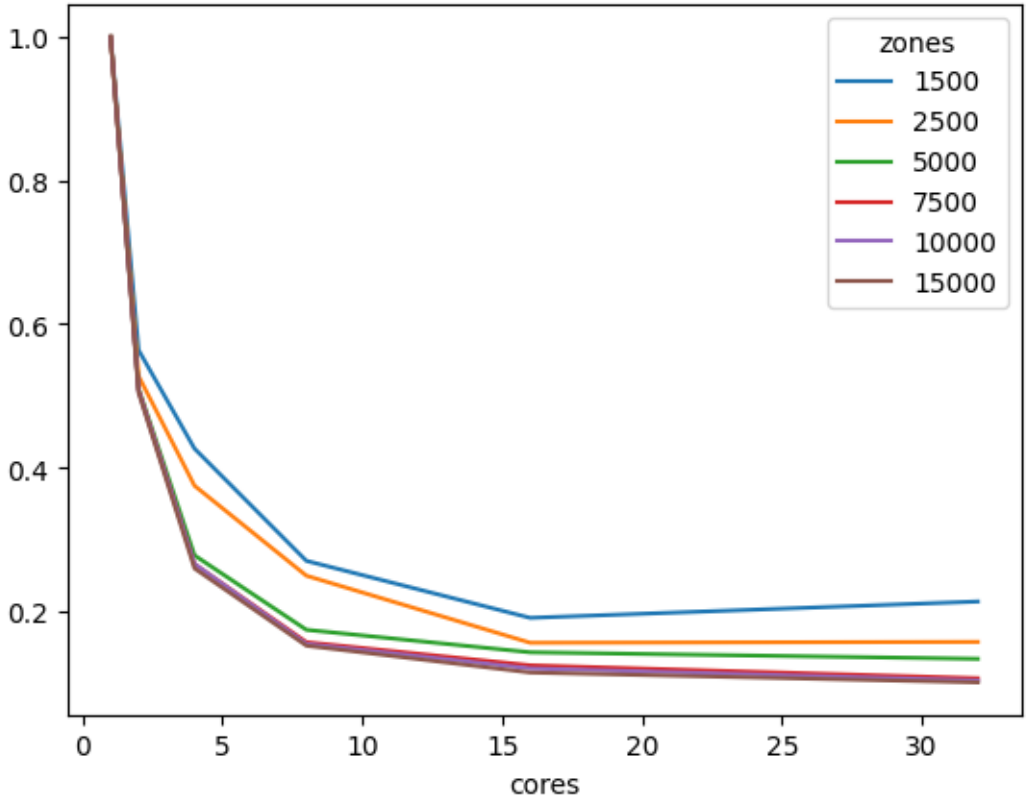


Fig. 4: number of cores used in IPF for 32 bit matrices

```
# From:
# https://github.com/joshchea/python-tdm/blob/master/scripts/CalcDistribution.py

def CalcFratar(ProdA, AttrA, Trips1, maxIter=10):
    '''Calculates fratar trip distribution
       ProdA = Production target as array
       AttrA = Attraction target as array
       Trips1 = Seed trip table for fratar
       maxIter (optional) = maximum iterations, default is 10
       Returns fratared trip table
    '''
    # print('Checking production, attraction balancing:')
    sumP = ProdA.sum()
    sumA = AttrA.sum()
    # print('Production: ', sumP)
    # print('Attraction: ', sumA)
    if sumP != sumA:
        # print('Productions and attractions do not balance, attractions will be␣
↪scaled to productions!')
        AttrA = AttrA*(sumP/sumA)
    else:
        pass
        # print('Production, attraction balancing OK.')
    # Run 2D balancing --->
    for balIter in range(0, maxIter):
        ComputedProductions = Trips1.sum(1)
        ComputedProductions[ComputedProductions == 0] = 1
        OrigFac = (ProdA/ComputedProductions)
        Trips1 = Trips1*OrigFac[:, np.newaxis]

        ComputedAttractions = Trips1.sum(0)
        ComputedAttractions[ComputedAttractions == 0] = 1
        DestFac = (AttrA/ComputedAttractions)
        Trips1 = Trips1*DestFac
    return Trips1
```

```
mat_sizes = [500, 750, 1000, 1500, 2500, 5000, 7500, 10000, 15000]
```

```
#Benchmarking
bench_data = []
cores = 1
repetitions = 5
iterations = 100
for zones in mat_sizes:
    for repeat in tqdm(range(repetitions), f"Repetitions for zone size {zones}"):
        mat1 = np.random.rand(zones, zones)
        target_prod = np.random.rand(zones)
        target_atra = np.random.rand(zones)
        target_atra *= target_prod.sum()/target_atra.sum()

        aeq_mat = deepcopy(mat1)
        # We use a nonsensical negative tolerance to force it to run all iterations
```

(continues on next page)

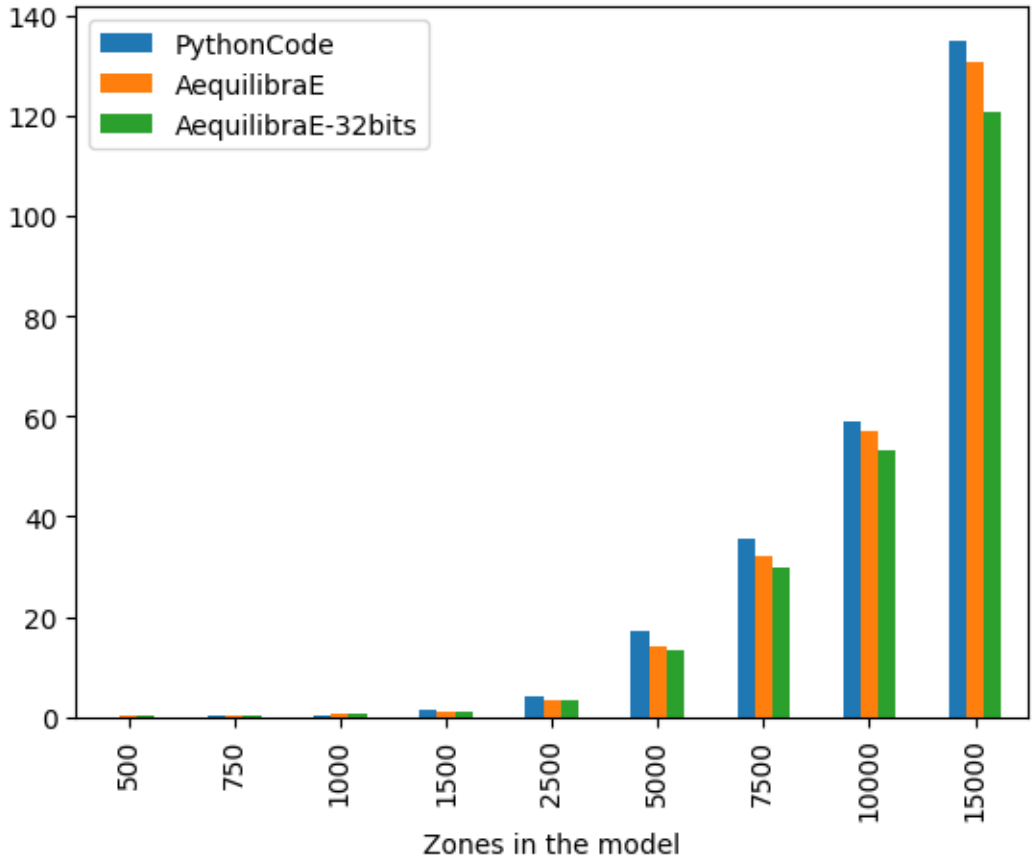
(continued from previous page)

```
        # and set warning for non-convergence to false, as we know it won't converge
        t = perf_counter()
        ipf_core(aeq_mat, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
        aeqt = perf_counter() - t

        aeq_mat32 = np.array(mat1, np.float32)
        # We now run the same thing with a seed matrix in single-precision (float 32␣
↪bits) instead of double as above (64 bits)
        t = perf_counter()
        ipf_core(aeq_mat32, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
        aeqt2 = perf_counter() - t

        bc_mat = deepcopy(mat1)
        t = perf_counter()
        x = CalcFratar(target_prod, target_atra, bc_mat, maxIter=iterations)

        bench_data.append([zones, perf_counter() - t, aeqt, aeqt2])
```

```
Repetitions for zone size 500: 100%|██████████| 5/5 [00:01<00:00,  2.60it/s]
Repetitions for zone size 750: 100%|██████████| 5/5 [00:04<00:00,  1.18it/s]
Repetitions for zone size 1000: 100%|██████████| 5/5 [00:07<00:00,  1.57s/it]
Repetitions for zone size 1500: 100%|██████████| 5/5 [00:19<00:00,  3.88s/it]
Repetitions for zone size 2500: 100%|██████████| 5/5 [00:56<00:00, 11.24s/it]
Repetitions for zone size 5000: 100%|██████████| 5/5 [03:44<00:00, 44.89s/it]
Repetitions for zone size 7500: 100%|██████████| 5/5 [08:09<00:00, 97.89s/it]
Repetitions for zone size 10000: 100%|██████████| 5/5 [14:11<00:00, 170.34s/it]
Repetitions for zone size 15000: 100%|██████████| 5/5 [32:23<00:00, 388.70s/it]
```

```
bench_df = pd.DataFrame(bench_data, columns=["Zones in the model", "PythonCode",
↪"AequilibraE", "AequilibraE-32bits"])
bench_df.groupby(["Zones in the model"]).mean().plot.bar()
```

```
<Axes: xlabel='Zones in the model'>
```




```
bench_df.groupby(["Zones in the model"]).mean()
```

```
#Benchmarking 32 threads
bench_data_parallel = []
cores = 32
repetitions = 5
iterations = 100
for zones in mat_sizes:
    for repeat in tqdm(range(repetitions), f"Repetitions for zone size {zones}"):
        mat1 = np.random.rand(zones, zones)
        target_prod = np.random.rand(zones)
        target_atra = np.random.rand(zones)
        target_atra *= target_prod.sum()/target_atra.sum()

        aeq_mat = deepcopy(mat1)
        # We use a nonsensical negative tolerance to force it to run all iterations
        # and set warning for non-convergence to false, as we know it won't converge
        t = perf_counter()
        ipf_core(aeq_mat, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
        aeqt = perf_counter() - t

        aeq_mat32 = np.array(mat1, np.float32)
        # We now run the same thing with a seed matrix in single-precision (float 32␣
```

(continues on next page)

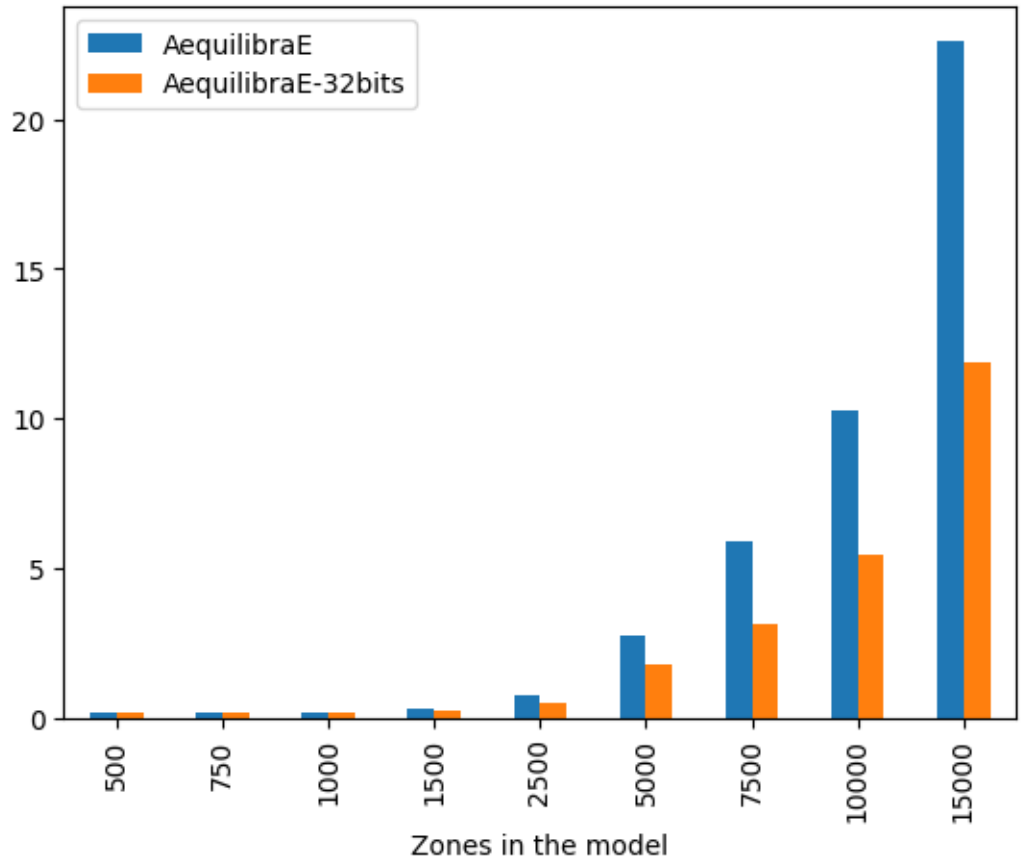
(continued from previous page)

```
↪bits) instead of double as above (64 bits)
        t = perf_counter()
        ipf_core(aeq_mat32, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
        aeqt2 = perf_counter() - t

        rmse = np.sqrt(np.mean((aeq_mat-aeq_mat32)**2))

        bench_data_parallel.append([zones, aeqt, aeqt2, rmse])
```

```
Repetitions for zone size 500: 100%|██████████| 5/5 [00:01<00:00,  2.70it/s]
Repetitions for zone size 750: 100%|██████████| 5/5 [00:01<00:00,  2.64it/s]
Repetitions for zone size 1000: 100%|██████████| 5/5 [00:02<00:00,  2.37it/s]
Repetitions for zone size 1500: 100%|██████████| 5/5 [00:03<00:00,  1.61it/s]
Repetitions for zone size 2500: 100%|██████████| 5/5 [00:07<00:00,  1.41s/it]
Repetitions for zone size 5000: 100%|██████████| 5/5 [00:24<00:00,  4.91s/it]
Repetitions for zone size 7500: 100%|██████████| 5/5 [00:49<00:00,  9.96s/it]
Repetitions for zone size 10000: 100%|██████████| 5/5 [01:26<00:00, 17.29s/it]
Repetitions for zone size 15000: 100%|██████████| 5/5 [03:10<00:00, 38.02s/it]
```

```
bench_df_parallel = pd.DataFrame(bench_data_parallel, columns=["Zones in the model",
↪"AequilibraE", "AequilibraE-32bits", "rmse"])
bench_df_parallel.groupby(["Zones in the model"]).mean()[[ "AequilibraE",
↪"AequilibraE-32bits"]].plot.bar()
```

```
<Axes: xlabel='Zones in the model'>
```



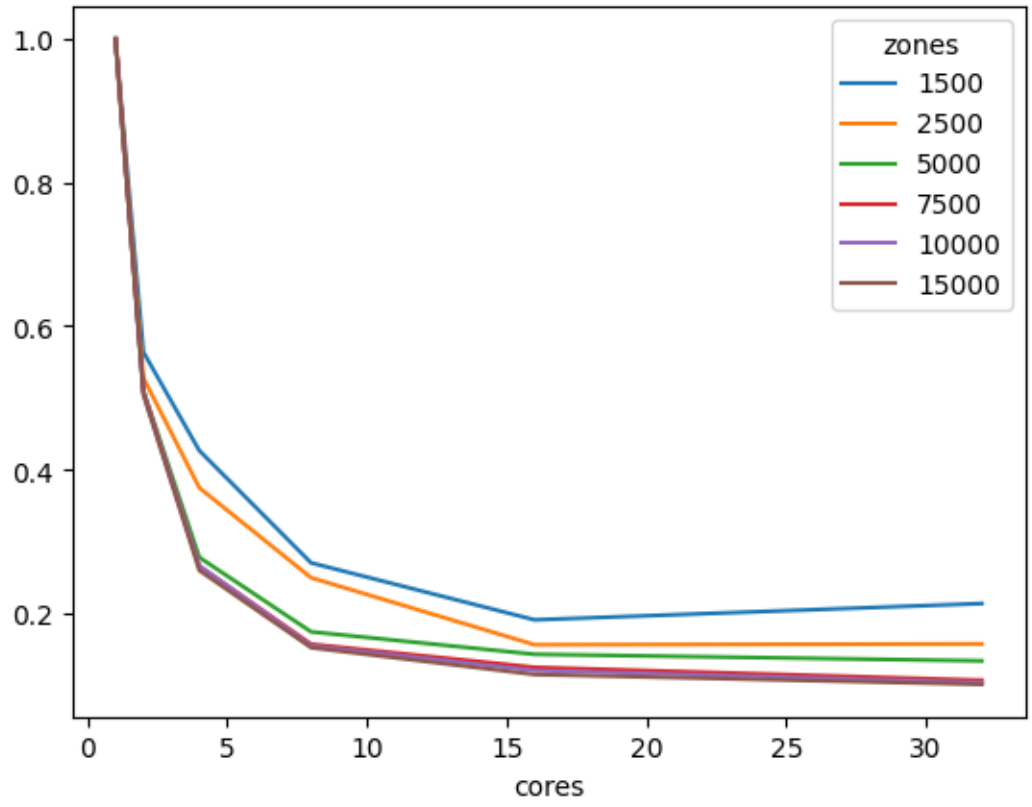

```
bench_df_parallel.groupby(["Zones in the model"]).mean()
```

```
cores_to_use = [1, 2, 4, 8, 16, 32]
```

```
aeq_data = []
repetitions = 1
iterations = 50
for zones in mat_sizes:
    for cores in tqdm(cores_to_use,f"Zone size: {zones}"):
        for repeat in range(repetitions):
            mat1 = np.random.rand(zones, zones)
            target_prod = np.random.rand(zones)
            target_atra = np.random.rand(zones)
            target_atra *= target_prod.sum()/target_atra.sum()

            aeq_mat = np.array(deepcopy(mat1), np.float32)
            t = perf_counter()
            ipf_core(aeq_mat, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
            aeqt = perf_counter() - t

            aeq_data.append([zones, cores, aeqt])
```

```
Zone size: 500: 100%|██████████| 6/6 [00:00<00:00, 12.14it/s]
Zone size: 750: 100%|██████████| 6/6 [00:00<00:00, 10.20it/s]
Zone size: 1000: 100%|██████████| 6/6 [00:00<00:00,  6.87it/s]
Zone size: 1500: 100%|██████████| 6/6 [00:01<00:00,  3.42it/s]
Zone size: 2500: 100%|██████████| 6/6 [00:04<00:00,  1.32it/s]
Zone size: 5000: 100%|██████████| 6/6 [00:16<00:00,  2.73s/it]
Zone size: 7500: 100%|██████████| 6/6 [00:35<00:00,  5.93s/it]
Zone size: 10000: 100%|██████████| 6/6 [01:02<00:00, 10.46s/it]
Zone size: 15000: 100%|██████████| 6/6 [02:21<00:00, 23.62s/it]
```

```
aeq_df = pd.DataFrame(aeq_data, columns=["zones", "cores", "time"])
aeq_df = aeq_df[aeq_df.zones>1000]
aeq_df = aeq_df.groupby(["zones", "cores"]).mean().reset_index()
aeq_df = aeq_df.pivot_table(index="zones", columns="cores", values="time")
for cores in cores_to_use[::-1]:
    aeq_df.loc[:, cores] /= aeq_df[1]
aeq_df.transpose().plot()
aeq_df
```




```
aeq_data = []
repetitions = 1
iterations = 50
for zones in mat_sizes:
    for cores in tqdm(cores_to_use,f"Zone size: {zones}"):
        for repeat in range(repetitions):
```

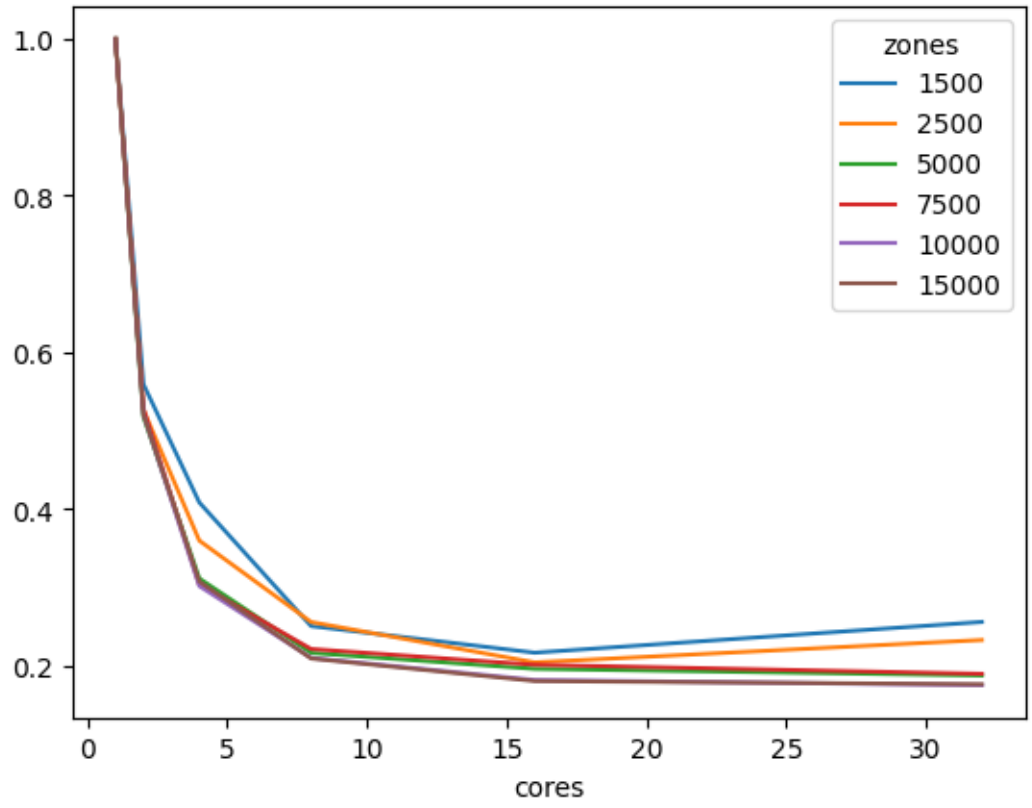
(continues on next page)

(continued from previous page)

```
            mat1 = np.random.rand(zones, zones)
            target_prod = np.random.rand(zones)
            target_atra = np.random.rand(zones)
            target_atra *= target_prod.sum()/target_atra.sum()

            aeq_mat = np.array(deepcopy(mat1), np.float64)
            t = perf_counter()
            ipf_core(aeq_mat, target_prod, target_atra, max_iterations=iterations,␣
↪tolerance=-5, cores=cores, warn=False)
            aeqt = perf_counter() - t

            aeq_data.append([zones, cores, aeqt])
```

```
Zone size: 500: 100%|██████████| 6/6 [00:00<00:00, 12.51it/s]
Zone size: 750: 100%|██████████| 6/6 [00:00<00:00,  9.19it/s]
Zone size: 1000: 100%|██████████| 6/6 [00:00<00:00,  6.50it/s]
Zone size: 1500: 100%|██████████| 6/6 [00:01<00:00,  3.07it/s]
Zone size: 2500: 100%|██████████| 6/6 [00:05<00:00,  1.17it/s]
Zone size: 5000: 100%|██████████| 6/6 [00:18<00:00,  3.14s/it]
Zone size: 7500: 100%|██████████| 6/6 [00:42<00:00,  7.10s/it]
Zone size: 10000: 100%|██████████| 6/6 [01:15<00:00, 12.51s/it]
Zone size: 15000: 100%|██████████| 6/6 [02:47<00:00, 27.93s/it]
```

```
aeq_df = pd.DataFrame(aeq_data, columns=["zones", "cores", "time"])
aeq_df = aeq_df[aeq_df.zones>1000]
aeq_df = aeq_df.groupby(["zones", "cores"]).mean().reset_index()
aeq_df = aeq_df.pivot_table(index="zones", columns="cores", values="time")
for cores in cores_to_use[::-1]:
    aeq_df.loc[:, cores] /= aeq_df[1]
aeq_df.transpose().plot()
aeq_df
```



5.3 Examples

5.3.1 Distribution Procedures

Running IPF with NumPy array

In this example, we show how to use `aequilibrae.distribution.cython.ipf_core`, a high-performance alternative for all those who want to (re)balance values within a matrix making direct use of growth factors. `ipf_core` was built to suit countless applications rather than being limited to trip distribution.

We demonstrate the usage of `ipf_core` with a 4x4 matrix with 64-bit data, which is indeed very small. Additionally, a more comprehensive discussion of the algorithm's performance with a 32-bit or 64-bit seed matrices is provided in *IPF Performance*.

The data used in this example comes from Table 5.6 in Ortúzar & Willumsen (2011).

References

- *IPF Performance*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.distribution.cython.ipf_core()`

```
# Imports
import numpy as np

from aequilibrae.distribution.cython.ipf_core import ipf_core
```

```
matrix = np.array([[5, 50, 100, 200], [50, 5, 100, 300], [50, 100, 5, 100], [100, 200,
↪ 250, 20]], dtype="float64")
future_prod = np.array([400, 460, 400, 702], dtype="float64")
future_attr = np.array([260, 400, 500, 802], dtype="float64")
```

Given our use of default parameter values in the other application of IPF, we should set *tolerance* value to obtain the same result.

```
num_iter, gap = ipf_core(matrix, future_prod, future_attr, tolerance=0.0001)
```

Let's print our updated matrix

```
matrix
```

Notice that the matrix value was updated, and results are the same as in *Running IPF without an AequilibraE model* - and this is no coincidence. Under the hood, when we call `aequilibrae.distribution.Ipf`, we are actually calling the `ipf_core` method.

Running IPF without an AequilibraE model

In this example, we show you how to use AequilibraE's IPF function without a model. This is a complement to the application in *Forecasting*.

Let's consider that you have an OD-matrix, the future production and future attraction values.

How would your trip distribution matrix using IPF look like?

The data used in this example comes from Table 5.6 in Ortúzar & Willumsen (2011).

References

- *AequilibraE Matrix*
- *IPF Performance*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.matrix.aequilibrae_matrix()`
- `aequilibrae.distribution.ipf()`

```
# Imports
from os.path import join
from tempfile import gettempdir

import numpy as np
```

(continues on next page)

(continued from previous page)

```
import pandas as pd

from aequilibrae.distribution import Ipf
from aequilibrae.matrix import AequilibraeMatrix
```

```
folder = gettempdir()
```

```
matrix = np.array([[5, 50, 100, 200], [50, 5, 100, 300], [50, 100, 5, 100], [100, 200,
↪ 250, 20]], dtype="float64")
future_prod = np.array([400, 460, 400, 702], dtype="float64")
future_attr = np.array([260, 400, 500, 802], dtype="float64")

num_zones = matrix.shape[0]
```

```
index = np.arange(1, num_zones + 1)
mtx = AequilibraeMatrix(index=index, index_name="zones")
mtx["matrix"] = matrix
```

```
args = {
    "entries": mtx.index.shape[0],
    "field_names": ["productions", "attractions"],
    "data_types": [np.float64, np.float64],
    "file_path": join(folder, "vectors.aem"),
}

vectors = pd.DataFrame({"productions": future_prod, "attractions": future_attr},␣
↪index=mtx.index)
```

```
args = {
    "matrix": mtx,
    "vectors": vectors,
    "row_field": "productions",
    "column_field": "attractions",
    "nan_as_zero": True,
}
fratar = Ipf(**args)
fratar.fit()
```

```
fratar.output["matrix"]
```

```
for line in fratar.report:
    print(line)
```

5.4 References

CHAPTER

SIX

PATH COMPUTATION

Given AequilibraE's incredibly fast path computation capabilities, one of its important use cases is the computation of paths on general transportation networks and between any two nodes, regardless of their type (centroid or not).

This use case supports the development of a number of computationally intensive systems, such as map-matching GPS data and simulation of Demand Responsive Transport (DRT, e.g. Uber) operators, for example.

Some basic usages of the AequilibraE path module consist on:

1. **Path computation**: computes the path between two arbitrary nodes.
2. **Network skimming**: can compute either the distance, the travel time, or your own cost matrix between a series of nodes.

Regarding computing paths through a network, part of its complexity comes from the fact that transportation models usually house networks for multiple transport modes, so the loads (links) available for a passenger car may be different than those available for a heavy truck, as it happens in practice.

For this reason, road path computation in AequilibraE happens through `Graph` objects built from a `Network`. Users can load an independent network with `project.road.build_network()`, then select a mode and cost field with `network.build_graph()`. Graph topology is fixed after construction, but routing costs can be updated in memory. Networks can also be created from link tables that are *NOT housed inside an AequilibraE model*.

AequilibraE's graphs are the backbone of path computation, skimming and traffic assignment. Besides handling the selection of links available to each mode in an AequilibraE model, graphs also handle the existence of bi-directional links with direction-specific characteristics (e.g. speed limit, congestion levels, tolls, etc.). For this reason, the next section is entirely dedicated to this object.

See also

- ***aequilibrae.routing.PathResults()***
  Class documentation
- ***Path computation***
  Usage example
- ***Network skimming***
  Usage example

6.1 AequilibraE Graphs

A `Network` stores source link attributes, modes, centroids, and turn restrictions. Its `build_graph()` method returns an array-backed `Graph` for routing. The graph has fixed topology. Its costs can change without a rebuild.

6.1.1 Graphs from a model

A project network can supply the source tables:

```
>>> project = create_example(project_path, "coquimbo")
>>> network = project.road.build_network()
>>> graph = network.build_graph('c', cost='distance')
```

The source tables do not retain project connections. Later changes to project data do not change this network or its prepared graphs.

To restrict the source data to an area, supply a polygon:

```
>>> from shapely.geometry import Polygon

>>> polygon = Polygon([(-71.35, -29.95), (-71.35, -29.90), (-71.30, -29.90), (-71.30,␣
↪-29.95), (-71.35, -29.95)])
>>> network = project.road.build_network(limit_to_area=polygon)
```

The area filter uses a spatial index. It does not clip links to the polygon boundary.

6.1.2 Source links and directed costs

The `network.links` DataFrame contains one row per physical link. The `direction` field specifies the allowed directions:

- `1` permits AB traversal.
- `-1` permits BA traversal.
- `0` permits both directions.

Direction-specific fields use the suffixes `_ab` and `_ba`. For example, `free_flow_time_ab` and `free_flow_time_ba` supply the graph field `free_flow_time`. `Network` preserves column names, and field names are case-sensitive. For each direction, a suffixed column takes precedence over the base column.

`network.field_values(name)` returns numeric values in physical-link input row order, with AB before BA for each bidirectional link. `network.link_ids` and `network.directions` use the same order. This order includes links that the graph's mode cannot use.

6.1.3 Cost updates and link closures

To close links without a topology change, set their costs to positive infinity:

```
>>> import numpy as np

>>> costs = network.field_values('distance').copy()
>>> costs[np.isin(network.link_ids, [123, 975])] = np.inf
>>> graph.set_costs(costs)
```

`set_costs()` requires one value per directed network link, not one value per graph link. A later call with finite costs can reopen the links. Negative costs and NaN values are invalid.

For changes to numeric link attributes, edit the source data and build a new graph:

```
>>> network.links.loc[network.links.link_type == 'motorway', 'speed_ab'] = 100
>>> network.links.loc[network.links.link_type == 'motorway', 'speed_ba'] = 100
>>> graph = network.build_graph('c', cost='distance')
```

Changes to source attributes do not update an existing graph or calculate dependent attributes. For example, a speed change does not automatically change travel time. For topology changes (link IDs, endpoints, directions, added/removed links or row order), construct a new `Network` from the revised link table so its mappings are rebuilt. Network configuration fields are fixed, use `dataclasses.replace()` to create a validated replacement when changing modes, centroids or turn restrictions.

6.1.4 Skimming and centroids

Skim fields, centroids, and centroid-flow blocking are graph construction options:

```
>>> graph = network.build_graph(
...     'c', cost='distance',
...     centroids=network.centroids[:5],
...     skimming=['distance'], block_centroid_flows=False,
... )
>>> skimmer = graph.compute_skims()
```

Without an explicit centroid array, the graph uses `network.centroids`. Centroids must be unique, positive node IDs from the network. Their order determines the row and column order of skim matrices. To change centroids or skim fields, build a new graph.

6.1.5 Compact graphs

A compact graph contracts chains of links. Dead-end removal is an additional, separate option. These reductions can reduce routing work for traffic assignment and centroid-to-centroid skimming. Both are opt-in when building a graph, and both require centroids. `PathResults.path` contains local graph-link indices, and `get_network_links()` expands these to physical link IDs and directions in traversal order.

To enable both reductions:

```
>>> compact = network.build_graph('c', cost='distance', compact=True, remove_dead_
↪ends=True)
```

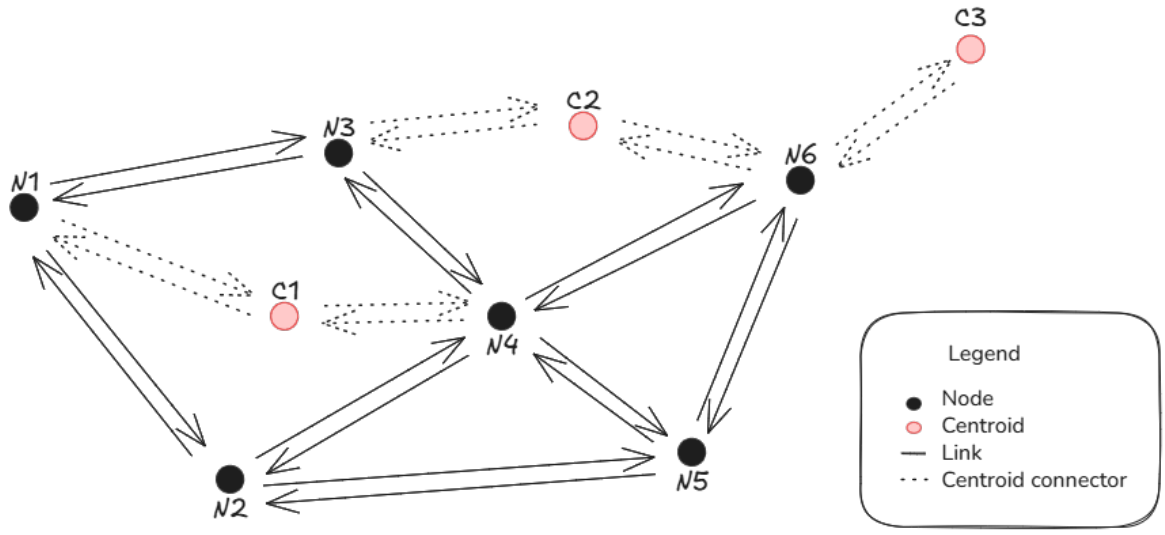
Traffic assignment enables both reductions by default. Each `TrafficClass` can override `compact` and `remove_dead_ends` through its `graph_options`. Congestion calculations still use individual directed network links.

6.1.6 Blocking centroid flows

By default, prepared graphs block paths through intermediate centroids. For turn-based routing, automatic turn prohibitions prevent traversal between centroid connectors at the same centroid.

Suppose a path from N1 to C3 follows N1 -> N3 -> C2 -> N6 -> C3 in the figure. Centroid-flow blocking prohibits the connector-to-connector transition at C2. The path must use another route.

If regular network nodes serve as centroids, consider whether routes must pass through those nodes. Use `block_centroid_flows=False` to permit those routes.



See also

- *aequilibrae.routing.Network*
- *aequilibrae.routing.Graph*
- *aequilibrae.transit.TransitGraph*

6.2 Examples

6.2.1 Path computation

Graph from arbitrary data

In this example, we demonstrate how to create an AequilibraE Graph from an arbitrary network.

The example uses Sioux Falls data from TNTP.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`

```
# Imports
import numpy as np
import pandas as pd

from aequilibrae.routing import Network
```

We start by adding the path to load our arbitrary network.

```
net_file = "https://raw.githubusercontent.com/bstabler/TransportationNetworks/master/
↪SiouxFalls/SiouxFalls_net.tntp"
```

Let's read our data! We'll be using Sioux Falls transportation network data, but without geometric information. The data will be stored in a Pandas DataFrame containing information about initial and final nodes, link distances, travel times, etc.

```
net = pd.read_csv(net_file, skiprows=8, sep="\t", lineterminator="\n", usecols=np.
↪arange(1, 11))
```

The Network requires four fields: link_id, a_node, b_node, and direction.

We need to manipulate the data to add the missing fields (link_id and direction) and rename the node columns accordingly.

```
net.insert(0, "link_id", np.arange(1, net.shape[0] + 1))
net = net.assign(direction=1)
net.rename(columns={"init_node": "a_node", "term_node": "b_node"}, inplace=True)
```

Now we can take a look in our network file

```
net.head()
```

Create the source network, then build a graph with costs and skim fields. All Sioux Falls nodes are centroids, so routes must be able to pass through them.

```
network = Network(net, centroids=np.arange(1, 25))
graph = network.build_graph(
    "c", cost="length", skimming=["length", "free_flow_time"], block_centroid_
↪flows=False,
)
```

Two of AequilibraE's new features consist in directly computing path or skims.

Let's compute the path between nodes 1 and 17…

```
res = graph.compute_path(1, 17)
```

… and print the corresponding nodes…

```
print(res.path_nodes)
```

… and the physical path link IDs and directions. `res.path` contains local graph-link indices instead.

```
print(res.get_network_links())
```

`graph.compute_path()` returns a `PathResults` object. Its `milepost` array contains the cumulative path costs.

`graph.compute_skims()` runs `NetworkSkimming` and returns the skimmer.

```
skm = graph.compute_skims()
```

Let's get the values for 'free_flow_time' matrix.

```
skims = skm.results.skims
skims["free_flow_time"]
```

Now we're all set!

Graph image credits to Behance-network icons created by Sumitsaengtong - Flaticon

Path computation with in-memory turn restrictions

In this example, we show how to build a new graph with turn restrictions from a network already loaded in memory.

We use the Coquimbo example and compare paths before and after prohibiting one specific turn.

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.routing.Network.build_graph()*
- *aequilibrae.routing.Graph.compute_path()*

Imports

```
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from aequilibrae.examples import create_example
```

We create the example project inside our temp folder

```
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "coquimbo")
```

Load the network snapshot and get the car graph.

```
network = project.road.build_network()
graph = network.build_graph(mode="c", cost="distance", block_centroid_flows=False)
```

We choose origin/destination.

```
origin = 32343
destination = 22041
```

Path without turn restrictions.

```
res_before = graph.compute_path(origin, destination)
```

Build prohibited turns from node triples sampled along the baseline path.

```
path_nodes = [int(x) for x in res_before.path_nodes]
triples = [(path_nodes[i], path_nodes[i + 1], path_nodes[i + 2]) for i in range(max(0,
↪ len(path_nodes) - 2))]
turn_restrictions = pd.DataFrame(
    {
        "from_node": [t[0] for t in triples[:4]],
        "via_node": [t[1] for t in triples[:4]],
        "to_node": [t[2] for t in triples[:4]],
```

(continues on next page)

(continued from previous page)

```
        "penalty": [np.nan, np.nan, np.nan, np.nan],
    }
)
```

Build a new graph with turn restrictions in memory, with node-based U-turns globally prohibited (i.e. transitions that return to the previous node).

```
graph = network.build_graph(
    mode="c", cost="distance", turn_restrictions=turn_restrictions,
    allow_uturns=False, block_centroid_flows=False,
)
```

Path with the custom turn prohibition.

```
res_after = graph.compute_path(origin, destination)
```

Plot both paths.

```
links = project.tables.links.data.set_index("link_id")
link_ids_before, directions_before = res_before.get_network_links()
link_ids_after, directions_after = res_after.get_network_links()
links_before = links.loc[link_ids_before]
links_after = links.loc[link_ids_after]
```

```
m = links_before.explore(color="blue", style_kwds={'weight':5})
m = links_after.explore(m=m, color="red", style_kwds={'weight':5})

#%
m
```

Close the project

```
project.close()
```

Path computation

In this example, we show how to perform path computation for Coquimbo, a city in La Serena Metropolitan Area in Chile.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- `aequilibrae.routing.results.path_results()`

Imports

```
from os.path import join
from tempfile import gettempdir
from uuid import uuid4
```

(continues on next page)

(continued from previous page)

```
import geopandas as gpd

from aequilibrae.routing import estimate_heuristic_scale
from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

We create the example project inside our temp folder

```
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Path Computation

Load the network snapshot We get warnings that several fields in the project are filled with ``NaN``s.

```
network = project.road.build_network()
# This is true, but we won't use those fields.
```

We build the graph for cars, minimise distance, and skim time and distance.

```
graph = network.build_graph(mode="c", cost="distance", skimming=["travel_time",
↪"distance"])
```

we'll also see what modes are available.

```
network.modes
```

Let's create a path results object from the graph and compute a path from node 32343 (near the airport) to 22041 (near Fort Lambert, overlooking Coquimbo Bay).

```
res = graph.compute_path(32343, 22041)
```

Computing paths directly from the graph is more straightforward, though we could alternatively use the `PathResults` class to achieve the same result.

from aequilibrae.routing import PathResults res = PathResults(graph, 32343, 22041)

`graph.compute_path` has already performed this setup above.

We can get the sequence of nodes we traverse

```
res.path_nodes
```

We can get the physical link IDs and directions we traverse. `res.path` contains local graph-link indices instead.

```
res.get_network_links()
```

We can get the mileposts for our sequence of nodes

```
res.milepost
```

Additionally, you can also provide `early_exit=True` or `a_star=True` to *compute_path* to adjust its path-finding behaviour.

Providing `early_exit=True` allows you to quit the path-finding procedure once it discovers the destination. This setup works better for topographically close origin-destination pairs. However, exiting early may cause subsequent calls to `update_trace` to recompute the tree in cases where it typically wouldn't.

```
res = graph.compute_path(32343, 22041, early_exit=True)
```

To guide the search towards the destination, provide `a_star=True` to use *A** with a heuristic. `update_trace` reuses finalised paths, or searches again if needed, retaining the algorithm, heuristic, scale and heap. Note that a_star takes precedence over early_exit. Euclidean distance needs node coordinates projected to a suitable local CRS and a scale parameter to transform it into a cost value.

```
lonlat = network.lonlat
points = gpd.GeoSeries(gpd.points_from_xy(lonlat.lon, lonlat.lat), index=lonlat.index,
↪ crs=4326)
points = points.to_crs(points.estimate_utm_crs())
coordinates = points.get_coordinates()

scale = estimate_heuristic_scale(graph, coordinates)
res = graph.compute_path(32343, 22041, a_star=True, coordinates=coordinates,␣
↪heuristic_scale=scale)
```

If you are using *a_star*, it is possible to use different heuristics to compute the path. By default, a Euclidean heuristic is used, and we can view the available heuristics via:

```
res.get_heuristics()
```

To use geographic distance without projecting the coordinates, choose "haversine". Pass the network's longitude/latitude data explicitly to both the scale estimator and the search.

```
scale = estimate_heuristic_scale(graph, lonlat, heuristic="haversine")
res = graph.compute_path(
    32343, 22041, a_star=True, heuristic="haversine", coordinates=lonlat, heuristic_
↪scale=scale
)
```

Suppose you want to adjust the path to the University of La Serena instead of Fort Lambert. It is possible to adjust the existing path computation for this alteration. The following code allows both *early_exit* and *A** settings to persist when calling `update_trace`. If you'd like to adjust them for subsequent path re-computations, call `compute_path` with the desired settings. Notice that this procedure is much faster when you have large networks and the search for the previous destination happened to include the new destination.

```
res.update_trace(73131)

res.path_nodes
```

If you want to show the path in Python.

We do NOT recommend this, though… It is very slow for real networks.

```
links = project.tables.links.data.set_index("link_id")
link_ids, directions = res.get_network_links()
links = links.loc[link_ids]
```

```
links.explore(color="blue", style_kwds={"weight": 5})
```

```
project.close()
```

Network skimming

In this example, we show how to perform network skimming for Coquimbo, a city in La Serena Metropolitan Area in Chile.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- `aequilibrae.routing.network_skimming()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Network Skimming

```
import numpy as np
```

Load the network snapshot

```
network = project.road.build_network()
# Several fields in the project are filled with ``NaN``s.
# This is true, but we won't use those fields.
```

we also see what modes are available

```
network.modes
```

(continues on next page)

(continued from previous page)

```
# let's say we want to minimise the distance
# And will skim distance while we are at it, other fields like ``free_flow_time`` or␣
↪``travel_time``
# can be added here as well

# But let's say we only want a skim matrix for nodes 28-40, and 49-90 (inclusive),
# these happen to be a selection of western centroids.
centroids = np.array(list(range(28, 41)) + list(range(49, 91)))
graph = network.build_graph(mode="c", cost="distance", skimming=["distance"],␣
↪centroids=centroids, compact=True)
```

And run the skimming

```
skm = graph.compute_skims()
```

Building network skims directly from the graph is more straightforward, though we could alternatively use the class `NetworkSkimming` to achieve the same result.

```
# from aequilibrae.routing import NetworkSkimming

# skm = NetworkSkimming(graph)
# skm.execute()
```

The result is an AequilibraeMatrix object

```
skims = skm.results.skims

# Which we can manipulate directly in memory, if we wish
skims.imat["distance", :3, :3]
```

Or access each matrix, lets just look at the first 3x3

```
skims["distance"][:3, :3]
```

We can save it to the project if we want

```
project.matrices.save_matrix(
    "base_skims.omx",
    skims,
    procedure="Network skimming",
    procedure_id=skm.procedure_id,
    timestamp=skm.procedure_date,
)
```

We can also open this skim file to write something to its description

```
matrices = project.matrices
with matrices.get_matrix("base_skims.omx", mode="r+") as matrix:
    matrix.metadata["description"] = "minimised distance while also skimming distance␣
↪for just a few nodes"
```

```
project.close()
```

CHAPTER

SEVEN

STATIC TRAFFIC ASSIGNMENT

Performing traffic assignment or computing paths through a network is always a little different in each platform, and in AequilibraE is no exception, but we strive to make the static traffic assignment process as simple as possible so that seasoned modelers can easily migrate their models and workflows to the platform.

Although modeling with AequilibraE should feel somewhat familiar to seasoned modelers, especially those used to programming, the mechanics of traffic assignment in AequilibraE might be foreign to part of our audience, so this section of the documentation covers the mechanics behind some of these procedures, along with a brief look at their motivation.

7.1 Traffic Assignment Procedure

Along with a network data model, traffic assignment is the most technically challenging portion to develop in a modelling platform, especially if you want it to be *fast*. In AequilibraE, we aim to make it as fast as possible, without making it overly complex to use, develop and maintain, although we know that *complex* is subjective.

Running traffic assignment in AequilibraE consists in creating the traffic classes that are going to be assigned, passing them to a traffic assignment object with the assignment parameters, and running the assignment.

7.1.1 `TrafficClass`

The `TrafficClass` object defines a single user class independently of the network. It holds the mode, cost field, demand matrix, optional skimming fields, and any fixed-cost or select-link configuration. The required information is:

- **mode**: the network mode identifier (for example `"c"` for cars)
- **cost**: the free-flow generalised cost field used for route choice
- **demand**: the `AequilibraeMatrix` or `AequilibraeSparseMatrix` containing OD demand for that class
- **name**: an optional class label; when omitted, it defaults to the mode

```
>>> from aequilibrae.traffic import TrafficClass
>>> from aequilibrae.examples import create_example

>>> project = create_example(project_path)

# Get the matrices for cars and trucks
>>> matrix_car = project.matrices.get_matrix("demand_mc.omx", subset=["car"])
>>> matrix_truck = project.matrices.get_matrix("demand_mc.omx", subset=["trucks"])

# Create the traffic classes
>>> tc_car = TrafficClass(mode="c", cost="free_flow_time", demand=matrix_car,
...                       name="car", skimming=("free_flow_time", "distance"),
...                       graph_options={"block_centroid_flows": False})
```

(continues on next page)

(continued from previous page)

```
>>> tc_truck = TrafficClass(mode="T", cost="free_flow_time", demand=matrix_truck,
...                         name="truck", skimming=("free_flow_time", "distance"),
...                         graph_options={"block_centroid_flows": False})
```

Routing policies such as `block_centroid_flows` are supplied through `graph_options`. It accepts `compact`, `remove_dead_ends`, `block_centroid_flows` and `allow_uturns`. When `allow_uturns` is omitted, the network's setting is used. The assignment builds the required graphs internally from the network.

It is also possible to set the common assignment parameters during class definition:

- **pce**: passenger-car equivalent for multi-class assignment; default is `1.0`.
- **vot**: value of time used to convert fixed costs to time units; default is `1.0`.
- **fixed_cost_field**: link field providing a monetary or other fixed cost in units compatible with `vot`. `TrafficClass` converts it to time by multiplying by `fixed_cost_multiplier` and dividing by `vot`.
- **fixed_cost_multiplier**: scale applied to the fixed cost field.

```
>>> tc_truck = TrafficClass(
...     mode="T",
...     cost="free_flow_time",
...     demand=matrix_truck,
...     name="truck",
...     pce=2.5,
...     vot=0.35,
...     fixed_cost_field="distance",
...     fixed_cost_multiplier=1.0,
...     skimming=("free_flow_time", "distance"),
...     graph_options={"block_centroid_flows": False},
... )
```

7.1.2 `TrafficAssignment`

```
>>> from aequilibrae.traffic import TrafficAssignment
>>> from aequilibrae.traffic.vdf import bpr

>>> network = project.road.build_network()
>>> assignment = TrafficAssignment(
...     network=network,
...     project=project,
...     classes=[tc_car, tc_truck],
...     time_field="free_flow_time",
...     capacity_field="capacity",
...     vdf=bpr,
...     algorithm="bfw",
...     name_mapping={"alpha": "b", "beta": "power"},
... )
```

AequilibraE's traffic assignment is organised around a single constructor that retains a standalone network and current class configuration, or copies source data from a project network. It then prepares each class's routing graph during execution. Do not mutate a standalone network while an assignment uses it.

- **network**: the source network object or project network containing the links and modes
- **project**: the project used when saving results or skims from an independent network

- **classes**: one or more fully-specified `TrafficClass` objects
- **vdf**: the volume-delay function (VDF) to be used, being one of `BPR`, `BPR2`, `CONICAL`, `INRETS`, or `AKCELIK`. Or a custom `VDF` object.
- **name_mapping**: parameter names for the VDF, either as network fields or numeric constants
- **time_field**: link field representing free-flow travel time
- **capacity_field**: link field representing capacity
- **algorithm**: assignment algorithm, such as `"bfw"`, `"fw"`, `"msa"`, or `"all-or-nothing"`

```
>>> assignment = TrafficAssignment(
...     network=network,
...     project=project,
...     classes=[tc_car, tc_truck],
...     time_field="free_flow_time",
...     capacity_field="capacity",
...     vdf=bpr,
...     algorithm="bfw",
...     name_mapping={"alpha": 0.15, "beta": 4.0},
... )
```

Skimming while assigning

AequilibraE can skim during assignment and keeps both the latest all-or-nothing skims and the blended skims for the accepted solution.

This is the case because, strictly speaking, the equilibrium travel time is the one resulting at the end of the last assignment iteration, while the most correct distance and toll skims, for example, are those resulting from the blended skim of all iterations.

The user can select the fields they want to skim, as well save the skims to disk (and the project) at the end of the assignment procedure, where skims are tagged with suffixes *"_final"* and *"_blended"* for easy identification.

The `skimming` fields are set on each `TrafficClass` before constructing the assignment, as above.

```
>>> _ = assignment.execute(max_iterations=10, cores=1)
>>> assignment.save_skims("one_matrix_name", which_ones="all")
['one_matrix_name_car.omx', 'one_matrix_name_truck.omx']
```

Combined link results

`assignment.results()` and `execute()` return class load columns named `<demand_matrix>_ab`, `_ba` and `_tot`, without traffic-class prefixes. Result column names must be unique across all classes and shared network metrics, such as `PCE_tot`. Conflicts are checked before execution, ignoring case to match SQLite result tables.

Unassigned demand

Unassigned demand has an unreachable destination and is not loaded on the network. `assignment.class_results["car"].unassigned_demand` gives the total for the accepted solution, excluding intrazonal demand.

Assigning sparse matrices

Modern Activity-Based models (and even some trip-based and tour-based ones) result on incredibly sparse demand matrices, which opens up a significant opportunity to save time during assignment by using early-exiting during the path-computation phase of assignment.

AequilibraE is capable of leveraging this opportunity, and it does so automatically whenever the user **does NOT set skimming fields on the traffic classes**.

In this case, AequilibraE has a convenient method to skim the final iteration of the assignment as it had been computed during the assignment itself. This method call requires a new iteration of path computation to be made, but assignments with highly sparse matrices and more than 10 iterations would still experience a significant speedup, which comes at the cost of not having blended skims as a sub-product of the assignment.

Time savings of up to 40% should be achievable in cases of micro-simulated ABMs with a large number of zones and over 100 iterations of assignment.

```
>>> skims = assignment.skim_congested(skim_fields=["distance"], return_matrices=True)
>>> assignment.save_skims("another_matrix_name")
['another_matrix_name_car.omx', 'another_matrix_name_truck.omx']
```

The list of fields defined by the user for skimming is added to the congested time and the assignment cost from the last iteration of the assignment by default. These matrices are named *__congested_time__* and *__assignment_cost__* respectively. The returned dictionary maps traffic class names to `SkimmingOutputs` objects. Congested time includes link travel time and turn delay, but excludes fixed costs. Assignment cost includes those fixed costs, converted to time units with the traffic classes value of time.

See the the example *Assigning sparse matrices* for a more practical explanation of this feature.

Volume-delay function

AequilibraE supports five Volume Delay Functions (VDFs): BPR, BPR2, Conical, INRETS, and Akcelik. Each VDF has distinct characteristics that make it suitable for different modelling scenarios.

For detailed information on VDF formulations, parameter guidance, and usage recommendations, see the dedicated *Volume Delay Functions* page, which includes:

- Mathematical formulas and technical descriptions
- Default parameter values and calibration guidance
- Behavioural charts showing V/C ratios from 0 to 3
- Origin and background for each function
- Recommendations for choosing the right VDF

Quick reference:

- **BPR**: Standard highway assignment (α=0.15, β=4.0)
- **BPR2**: Enhanced over-capacity penalty (α=0.15, β=4.0)
- **Conical**: Smooth theoretical function (α=0.15, β=4.0)
- **INRETS**: Urban arterials (α=1.0)
- **Akcelik**: Signalised intersections (α=0.25, τ=0.8)

7.1.3 Setting Preloads

We can also optionally include a preload vector for constant flows which are not being otherwise modelled. For example, this can be used to account for scheduled public transport vehicles, adding an equivalent load to each link along the route accordingly. AequilibraE supports various conditions for which PT trips to include in the preload, and allows the user to specify the PCE for each type of vehicle in the public transport network.

To create a preload for public transport vehicles operating between 8 AM to 10 AM, do the following:

```
# Times are specified in seconds from midnight
>>> transit = project.scenario.create_transit_database()
>>> preload = transit.build_pt_preload(start=8*3600, end=10*3600)
```

Pass a nonempty preload table to the assignment constructor using `preload=preload`. The table must contain `link_id`, `direction` and `preload` columns; its flow values are in PCE units.

7.1.4 Executing an Assignment

Finally, run traffic assignment!

```
>>> results = assignment.execute(max_iterations=10, cores=1)

>>> project.close()
```

7.1.5 References

7.2 Volume Delay Functions

Volume Delay Functions (VDFs) are mathematical functions that describe the relationship between link travel time and traffic volume. They are essential components of traffic assignment models, representing how congestion affects travel times as more vehicles use a link.

AequilibraE implements five different VDF formulations, each with distinct characteristics that make them suitable for different modelling traditions.

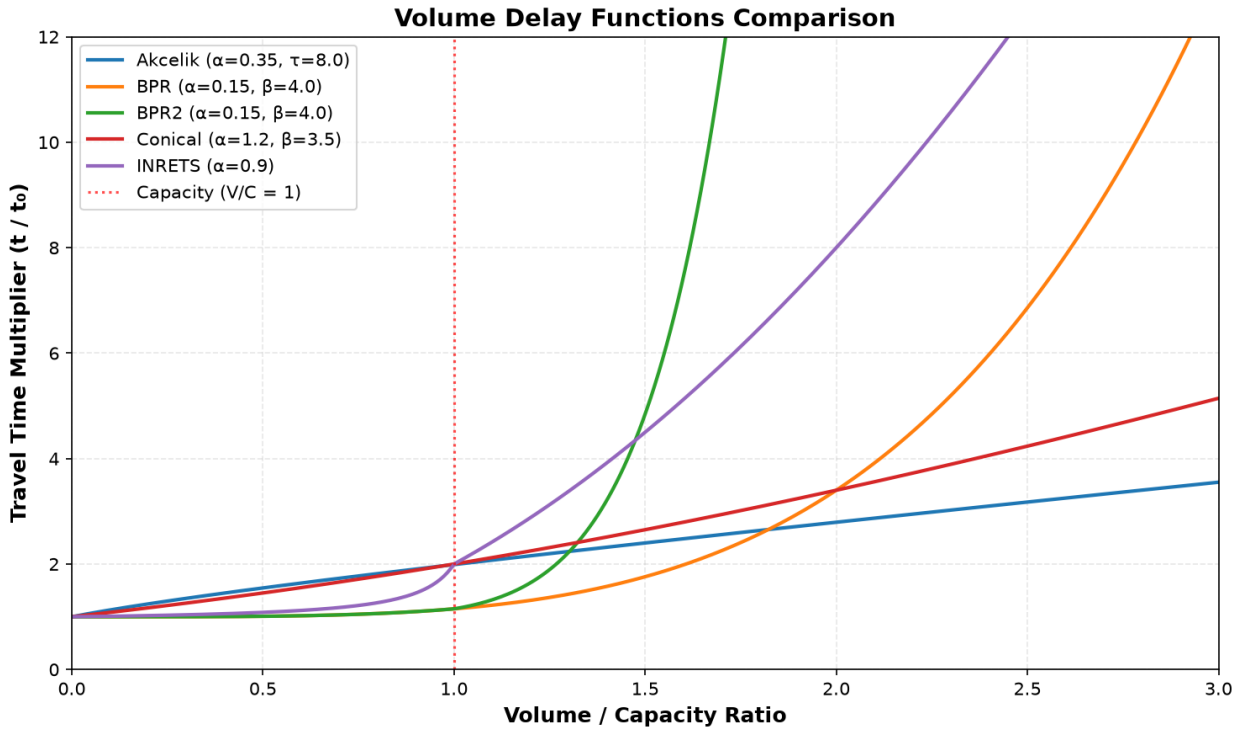
7.2.1 Available VDF Functions

AequilibraE currently supports the following VDF functions:

- **BPR** - Bureau of Public Roads (traditional)
- **BPR2** - Modified BPR with enhanced sensitivity after capacity
- **Conical** - Spiess' Conical function
- **INRETS** - French INRETS function
- **Akcelik** - Akcelik's function for signalized intersections

7.2.2 VDF Comparison

The following chart compares the behaviour of all available VDF functions with their example values:

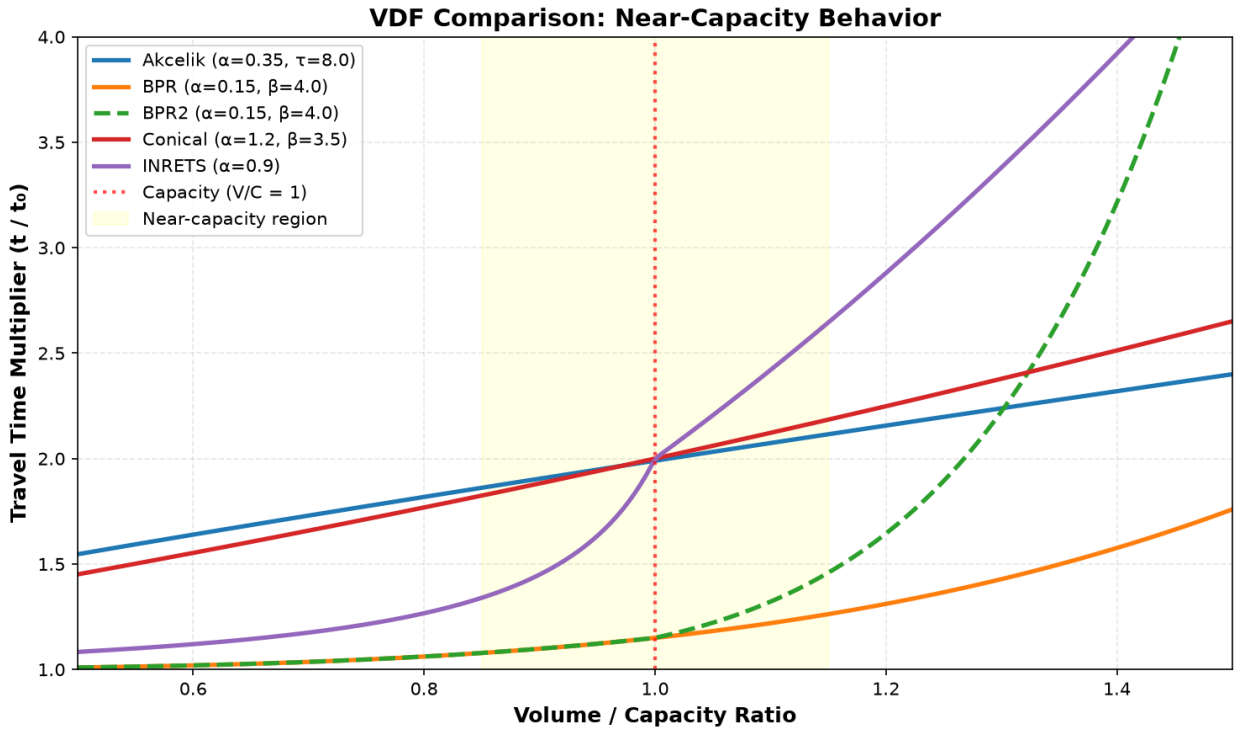


Key observations:

- All functions show increasing travel times as the Volume/Capacity (V/C) ratio increases
- Functions differ significantly in their behaviour near and above capacity (V/C = 1)
- BPR and Conical are smooth throughout the range
- BPR2 and INRETS have distinct behavioural changes at capacity
- Akcelik shows moderate growth rates suitable for signalized intersections

Near-Capacity Behavior

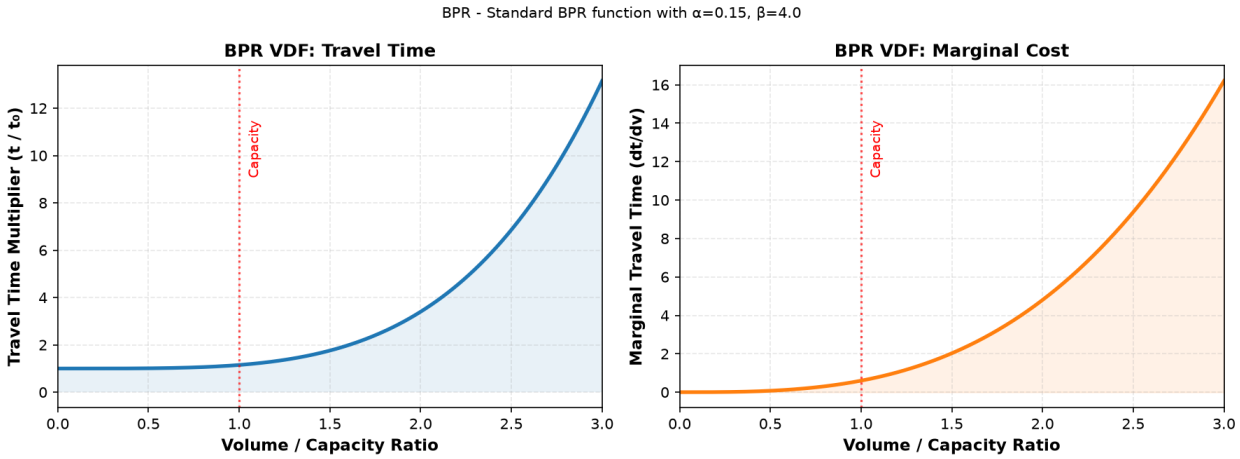
The behaviour of VDFs near capacity is particularly important for congested urban networks:



This zoomed view (V/C from 0.5 to 1.5) highlights the differences in how each function transitions from free-flow to congested conditions.

7.2.3 Detailed VDF Descriptions

BPR (Bureau of Public Roads)



Mathematical Formula:

$$t = t_0 \left(1 + \alpha \left(\frac{v}{c}\right)^{\beta}\right)$$

Where:

- t = congested travel time
- t_0 = free-flow travel time

- v = volume (flow) on the link
- c = capacity of the link
- α = calibration parameter
- β = calibration parameter (power)

Standard Parameters:

- $\alpha = 0.15$
- $\beta = 4.0$

Origin and Background:

The BPR function was developed by the U.S. Bureau of Public Roads in the 1960s and has become the most widely used VDF in transportation planning. Its simplicity and effectiveness have made it the standard choice for highway assignment models worldwide.

Characteristics:

- Continuous and smooth across all volume levels
- Monotonically increasing (no discontinuities)
- Convex function ensuring convergence in assignment algorithms
- Well-suited for highway networks
- Standard parameters are based on extensive empirical studies

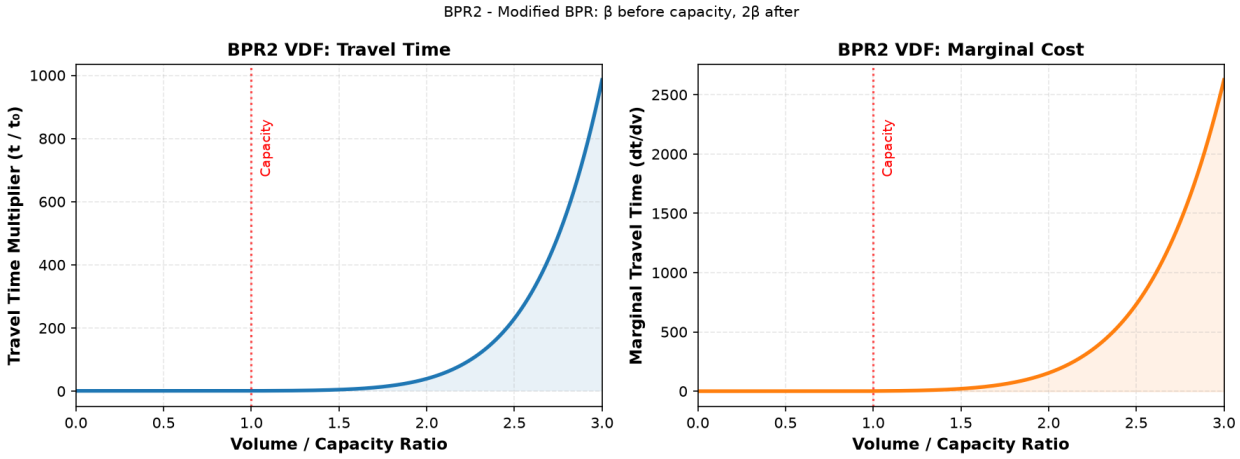
When to Use:

- Highway and freeway networks
- Long-distance travel modelling
- When computational stability is critical
- As a baseline for comparison with other functions
- When empirical data supports the standard parameters

Limitations:

- May underestimate congestion effects at very high V/C ratios
- Less suitable for signalized urban arterials

BPR2 (Modified BPR)



Mathematical Formula:

Before capacity ($v \leq c$):

$$t = t_0 \left(1 + \alpha \left(\frac{v}{c}\right)^{\beta}\right)$$

After capacity ($v > c$):

$$t = t_0 \left(1 + \alpha \left(\frac{v}{c}\right)^{2\beta}\right)$$

Standard Parameters:

- $\alpha = 0.15$
- $\beta = 4.0$

Origin and Background:

BPR2 is a modification of the traditional BPR function designed to better represent the rapid deterioration of traffic conditions when demand exceeds capacity. The doubling of the exponent after capacity creates a steeper penalty for over-capacity conditions.

Characteristics:

- Piecewise function with a transition at V/C = 1
- Much steeper increase in travel time after capacity is exceeded
- Maintains BPR behaviour below capacity
- Non-differentiable at V/C = 1 (but continuous)
- More aggressive congestion penalty than standard BPR

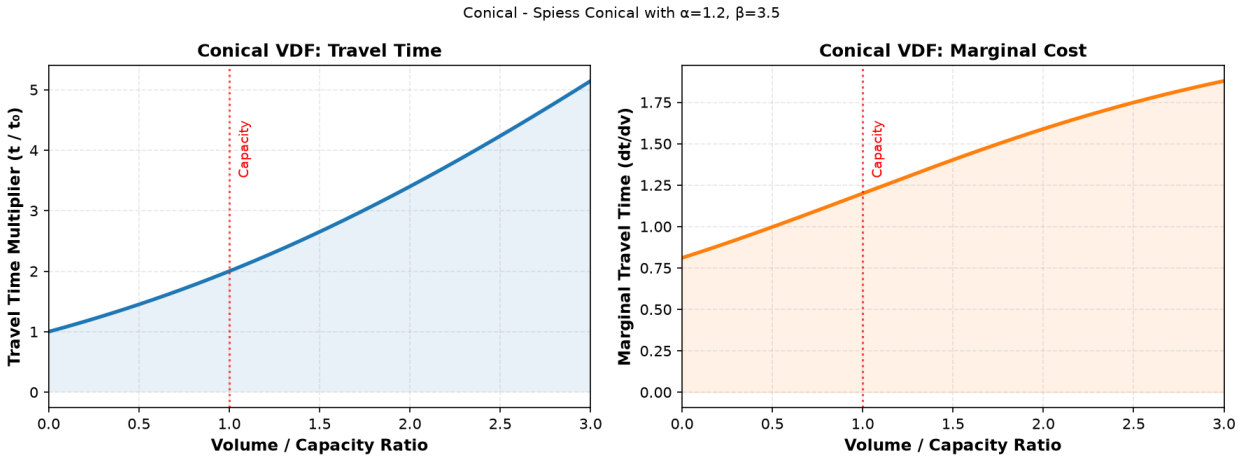
When to Use:

- Networks with strong capacity constraints
- Modeling scenarios where over-capacity conditions should be heavily penalized
- Mixed networks with both highways and congested urban roads
- When trying to prevent unrealistic over-assignment

Limitations:

- Non-differentiable point at capacity may cause convergence issues in some algorithms
- The sharp transition may not reflect real-world gradual congestion buildup
- Requires careful calibration of the transition behaviour

Conical (Spiess)



Mathematical Formula:

$$t = t_0 \left(2 + \sqrt{\alpha^2 \left(1 - \frac{v}{c}\right)^2 + \beta^2} - \alpha \left(1 - \frac{v}{c}\right) - \beta\right)$$

With $\beta = \frac{2\alpha - 1}{2\alpha - 2}$

Standard Parameters:

- $\alpha = 2.0$ (must be $\alpha > 1$)
- $\beta = 1.5$, according to the above relation

Origin and Background:

Developed by Heinz Spiess in 1990, the Conical function was designed to overcome some theoretical limitations of the BPR function while maintaining computational tractability. It ensures positive derivatives everywhere and has desirable mathematical properties for convergence.

Characteristics:

- Infinitely differentiable (smooth everywhere)
- Asymptotic behaviour as V/C approaches infinity
- Guaranteed positive marginal costs
- Strong theoretical foundation
- Good convergence properties in equilibrium algorithms

When to Use:

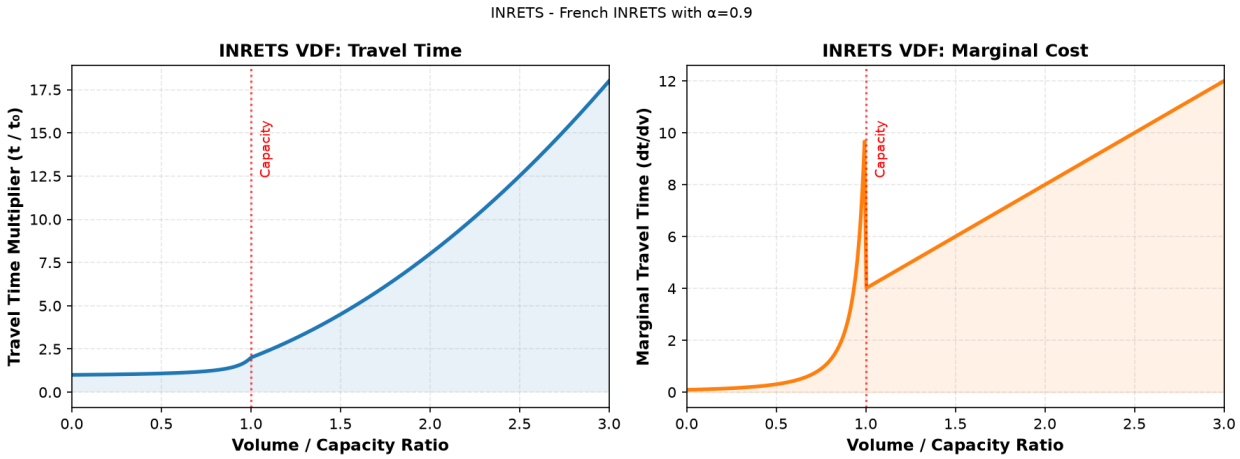
- When theoretical rigor is important
- Transit assignment (where it was originally designed)
- Networks requiring guaranteed convergence

- Academic studies and benchmarking
- When smooth derivatives are required throughout

Limitations:

- Less intuitive parameterization than BPR
- May require specialized calibration
- Not as widely validated in practice as BPR
- Parameters have different interpretations than BPR

INRETS (French)



Mathematical Formula:

Before capacity ($v \leq c$):

$$t = t_0 \frac{1.1 - \alpha \frac{v}{c}}{1.1 - \frac{v}{c}}$$

After capacity ($v > c$):

$$t = t_0 \frac{1.1 - \alpha}{0.1} \left(\frac{v}{c}\right)^2$$

Standard Parameters:

- $\alpha = 1.0$ (must be ≤ 1.0)

Origin and Background:

The INRETS function was developed by the French National Institute for Transport and Safety Research (Institut National de Recherche sur les Transports et leur Sécurité). It was designed specifically for French urban networks and reflects European traffic flow characteristics.

Characteristics:

- Piecewise function with distinct before/after capacity behaviour
- The α parameter must be less than or equal to 1.0
- Hyperbolic behaviour before capacity
- Quadratic behaviour after capacity

- Designed for urban arterial roads
- Reflects observed traffic patterns in French cities

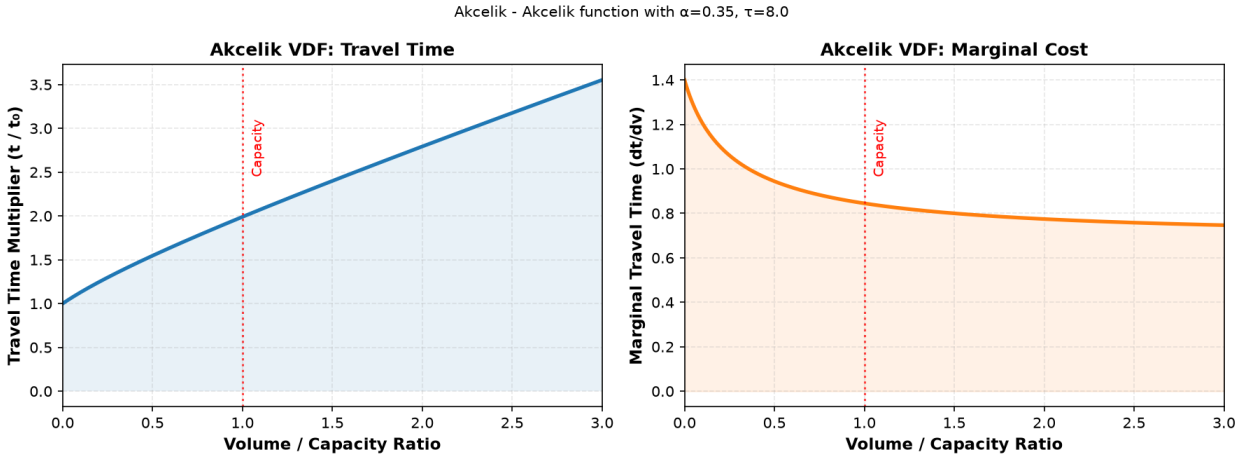
When to Use:

- Urban arterial networks
- European-style road networks
- When modelling contexts similar to French urban environments
- Networks with well-defined capacity constraints
- Calibrated models for specific urban areas

Limitations:

- Restricted parameter range ($\alpha \leq 1.0$)
- Non-differentiable at V/C = 1
- Less widely used outside of Europe
- May require local calibration
- Dramatic change in behaviour at capacity may not suit all networks

Akcelik



Mathematical Formula in the Literature:

$$t = t_0 + 0.25 \cdot \mathrm{T}_f \left(z + \sqrt{z^2 + \frac{8\tau v}{\mathrm{T}_f \cdot c^2}} \right)$$

Where $z = \frac{v}{c} - 1$, T_f is the duration of the analysis period, and t and t_0 are in the units time per distance (inverse of speed), not time. Importantly, the time unit component of T_f, t, and t_0 must match.

The units in this formulation do not match the units of other VDF functions implemented in AequilibraE. To provide a consistent API and usage semantics, we implement the equivalent formulation below.

Mathematical Formula in the AequilibraE:

$$t = t_0 + \text{length} \cdot \alpha \left(z + \sqrt{z^2 + \frac{\tau v}{c^2}} \right)$$

Where $z = \frac{v}{c} - 1$, and t and t_0 are in time units. This formulation introduces a new parameter α, and absorbs the $8/\mathrm{T}_f$ factor into τ.

Standard Parameters:

- $\alpha = 0.25 \cdot \mathrm{T}_f$
- $\tau = 0.1 \cdot 8/\mathrm{T}_f$ (see note below)

Note

Different than other VDF functions, Akcelik depends on link length and consistency between its parameters and the units for speed and link length. As it is common for models to use distance in meters, speed in kilometers per hour and to assign time periods longer than one hour, the α and τ parameters should be calibrated accordingly.

Origin and Background:

Developed by Rahmi Akcelik, this function was specifically designed for signalized intersections and urban arterials. It incorporates queue theory and reflects the delay characteristics of traffic signals.

Important Note on τ Parameter:

In standard Akcelik formulations, the function includes a factor of $8/\mathrm{T}_f$ in the formula. However, in AequilibraE's implementation, this factor of 8 has been absorbed into the τ parameter.

What this means for users:

- If academic literature references a τ value (e.g., 0.1), you must multiply it by 8 before setting it in AequilibraE, then adjust by the duration of the analysis period.
- Example: To use $\tau = 0.1$, set `tau = 0.8 / T_f` in AequilibraE.
- Example: To use $\tau = 0.15$, set `tau = 1.2 / T_f` in AequilibraE.
- A value of 0.8 corresponds to a standard $\tau = 0.1$ for the unit time period.

Characteristics:

- Specifically designed for signalized intersections
- Includes queue-based delay components
- Smooth transition through capacity
- More moderate growth than BPR at high V/C ratios
- Grounded in traffic signal theory

When to Use:

- Urban networks with signalized intersections
- Arterial roads with frequent signals
- Australian and some Asian modelling contexts (where it's more common)

Limitations:

- Less intuitive than BPR for general highway modelling
- Parameter interpretation requires understanding of signal timing
- May underestimate delay on uninterrupted flow facilities
- Less validated for freeway applications

7.2.4 Parameter Selection Guidelines

General Recommendations

For BPR and BPR2:

- $\alpha = 0.15$ and $\beta = 4.0$ are widely accepted for highways
- Urban arterials may benefit from α values between 0.15 and 0.25
- Higher β values (5-8) create sharper congestion curves
- Lower β values (2-3) create more gradual curves

For Conical:

- Values satisfying $2.0 \leq \alpha \leq 9.0$ have been recommended by the Hampton Roads Transportation Planning Organization (2020), depending on the types of roads being modelled.
- Fine-tuning may require understanding of the specific mathematical properties

For INRETS:

- $\alpha = 1.0$ is standard
- Must satisfy $\alpha \leq 1.0$
- Higher values create steeper curves before capacity

For Akcelik:

- $\alpha = 0.25$ is typical for signalized intersections
- τ should reflect local traffic signal characteristics
- Remember to use $8 \times \tau$ in AequilibraE

Calibration Considerations

When calibrating VDF parameters:

1. **Use observed data**: Speed-flow or travel time-volume data from your network
2. **Consider facility types**: Highways, arterials, and local streets may need different parameters
3. **Validate results**: Compare assigned volumes and speeds against counts
4. **Test sensitivity**: Understand how parameter changes affect results
5. **Check convergence**: Ensure your chosen parameters allow the algorithm to converge
6. **Match local conditions**: Standard parameters may not suit all geographic contexts

7.2.5 Setting VDF Parameters in AequilibraE

In AequilibraE, VDF parameters can be set either as network-wide constants or as link-specific attributes:

Network-wide parameters:

```
from aequilibrae.traffic import TrafficAssignment, TrafficClass
from aequilibrae.traffic.vdf import bpr

assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
```

(continues on next page)

(continued from previous page)

```
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=bpr,
    algorithm="bfw",
    name_mapping={"alpha": 0.15, "beta": 4.0},
)
```

Link-specific parameters:

```
# Parameters can reference field names in your network
assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=bpr,
    algorithm="bfw",
    name_mapping={"alpha": "alpha_field", "beta": "beta_field"},
)
```

This flexibility allows you to:

- Use different parameters for different facility types
- Implement spatially varying congestion characteristics
- Calibrate specific links or corridors independently

7.2.6 Choosing the Right VDF

Decision Framework

Use this decision tree to help select an appropriate VDF:

For Highway Networks:

- Start with **BPR** - it's well-validated and robust
- Consider **BPR2** if you need stronger over-capacity penalties
- Use **Conical** for theoretical/academic studies

For Urban Networks:

- Use **Akcelik** for signalized arterial networks
- Consider **INRETS** for European-style urban contexts
- **BPR** works as a general fallback

For Mixed Networks:

- **BPR** or **BPR2** provide good general performance
- Consider using link-specific parameters to vary behaviour by facility type

For Transit Assignment:

- **Conical** was originally designed for transit applications

Computational Considerations

- **BPR** and **Akcelik**: Smooth everywhere, excellent convergence
- **Conical**: Best mathematical properties, guaranteed convergence
- **BPR2** and **INRETS**: Non-differentiable at capacity, may need more iterations
- All functions are computationally efficient in AequilibraE

Performance Comparison

While individual network characteristics vary, general patterns include:

- **BPR**: Most widely validated, good general performance
- **BPR2**: Better for preventing unrealistic over-capacity assignment
- **Conical**: Excellent convergence behaviour, less intuitive calibration
- **INRETS**: Good for specific urban contexts, may need local calibration
- **Akcelik**: Best for signalized urban networks

7.2.7 Examples

Basic Usage

```
from aequilibrae.traffic import TrafficAssignment, TrafficClass
from aequilibrae.traffic.vdf import bpr, bpr2, akcelik, builtin_vdfs

# Check available VDF functions
vdfs = builtin_vdfs()
print(vdfs.keys())
# Output: dict_keys(['bpr', 'bpr2', 'conical', 'inrets', 'akcelik'])

# Setup traffic assignment with BPR
assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=vdfs["bpr"],
    algorithm="bfw",
    name_mapping={"alpha": 0.15, "beta": 4.0},
)
```

Using Different VDFs

```
# Highway network - standard BPR
highway_assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=bpr,
    algorithm="bfw",
```

(continues on next page)

(continued from previous page)

```
    name_mapping={"alpha": 0.15, "beta": 4.0},
)

# Urban network with signals - Akcelik
signal_assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=akcelik,
    algorithm="bfw",
    name_mapping={"alpha": 0.25, "tau": 0.8, "length": "distance"},
)

# Network with strong capacity constraints - BPR2
strong_capacity_assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=bpr2,
    algorithm="bfw",
    name_mapping={"alpha": 0.15, "beta": 4.0},
)
```

Link-Specific Parameters

```
# Assume your network has fields 'alpha_field' and 'beta_field'
# with values calibrated for each link
variable_assignment = TrafficAssignment(
    network=project.road.build_network(),
    classes=[traffic_class],
    time_field="free_flow_time",
    capacity_field="capacity",
    vdf=bpr,
    algorithm="bfw",
    name_mapping={"alpha": "alpha_field", "beta": "beta_field"},
)
```

7.2.8 Creating Custom VDFs

Custom VDFs can be created in three different ways. When using the python constructor of the VDF object, the VDF function and its derivative can be input as either a `Callable`, or as a string to be interpreted by NumExpr. Alternatively, the string representation and other data can be placed in a project's yaml file. In all cases, if the derivative is not supplied, then it will be computed numerically using the central difference scheme.

The VDF callable (`func`) and its derivative (`d_func`) write their results to their first argument, which is an output array. The VDF receives that array as `congested_time`, and the derivative receives it as `delta`. The exact calls are shown by these methods in the `VDF` class:

```
def apply_vdf(self, *, congested_time, link_flows, fftime, capacity, cores: int,␣
↪**link_attributes):
```

(continues on next page)

(continued from previous page)

```
    self.func(congested_time, link_flows, fftime, capacity, cores, **link_attributes)

def apply_derivative(self, *, delta, link_flows, fftime, capacity, cores: int, **link_
↪attributes):
    self.d_func(delta, link_flows, fftime, capacity, cores, **link_attributes)
```

A Python VDF callable must accept `congested_time, link_flows, fftime, capacity, cores`, followed by any custom link attributes. Its derivative must accept `delta, link_flows, fftime, capacity, cores`, followed by the same custom link attributes.

The following example defines the SANDAG modified two-part additive formulation[1] as a Python VDF.

```
# define custom vdf function
def SANDAG_modified_two_part_additive(
    congested_time,
    link_flows,
    fftime,
    capacity,
    cores,
    alpha_1,
    beta_1,
    alpha_2,
    beta_2,
    green_to_cycle_ratio,
    cycle_time,
    intersection_capacity,
):
    voc = link_flows / capacity
    voc_intersection = link_flows / intersection_capacity
    term_1 = fftime * (
        1.0 + alpha_1 * voc**beta_1
    )
    term_2 = (
        0.5
        * cycle_time
        * (1 - green_to_cycle_ratio) ** 2
        * (1.0 + alpha_2 * voc_intersection**beta_2)
    )
    congested_time[:] = term_1 + term_2

# define custom vdf derivative
def derivative_SANDAG_modified_two_part_additive(
    delta,
    link_flows,
    fftime,
    capacity,
    cores,
    alpha_1,
    beta_1,
    alpha_2,
```

(continues on next page)

[1] SANDAG (2011). *Activity-Based Model Development and Calibration: C04 Model Development*. Available: https://tfresource.org/topics/SANDAG_C04Report.pdf

(continued from previous page)

```
    beta_2,
    green_to_cycle_ratio,
    cycle_time,
    intersection_capacity,
):
    voc = link_flows / capacity
    voc_intersection = link_flows / intersection_capacity
    term_1 = (
        fftime
        * alpha_1
        * beta_1
        * voc ** (beta_1 - 1)
        / capacity
    )
    term_2 = (
        0.5
        * cycle_time
        * (1 - green_to_cycle_ratio) ** 2
        * alpha_2
        * beta_2
        * voc_intersection ** (beta_2 - 1)
        / intersection_capacity
    )
    delta[:] = term_1 + term_2

# define the specification
SANDAG_spec = {
    "alpha_1": {"fill_NA": 0.15, "bounds": (0.0, float("inf"))},
    "beta_1": {"fill_NA": 4.0, "bounds": (1.0, float("inf"))},
    "alpha_2": {"fill_NA": 0.15, "bounds": (0.0, float("inf"))},
    "beta_2": {"fill_NA": 4.0, "bounds": (1.0, float("inf"))},
    "green_to_cycle_ratio": {"bounds": (0.0, 1.0)},
    "cycle_time": {"bounds": (0.0, float("inf"))},
    "intersection_capacity": {"bounds": (0.0, float("inf"))}
}

# uses both the defined function and derivative
sandag_vdf = VDF(
    "SANDAG",
    SANDAG_modified_two_part_additive,
    SANDAG_spec,
    derivative_SANDAG_modified_two_part_additive,
)

# only uses the definition of the vdf function
# derivative is calculated numerically using finite difference
sandag_finite_difference_derivative_vdf = VDF(
    "SANDAG",
    SANDAG_modified_two_part_additive,
    SANDAG_spec,
)
```

The following shows the SANDAG modified two-part additive formulation being created using a string representation to

be evaluated by NumExpr.

```
SANDAG_string_representation = (
    "fftime * (1.0 + alpha_1 * (link_flows / capacity)**beta_1) + "
    "0.5 * cycle_time * (1 - green_to_cycle_ratio)**2 * "
    "(1.0 + alpha_2 * (link_flows / intersection_capacity)**beta_2)"
)

derivative_SANDAG_string_representation = (
    "fftime * alpha_1 * beta_1 * (link_flows / capacity) ** (beta_1 - 1) / capacity +
↪"
    "0.5 * cycle_time * (1 - green_to_cycle_ratio) ** 2 * "
    "alpha_2 * beta_2 * (link_flows / intersection_capacity) ** "
    "(beta_2 - 1) / intersection_capacity"
)

# string representations of the function and its derivative will be interpreted by␣
↪NumExpr
sandag_string_vdf = VDF(
  "SANDAG",
  SANDAG_string_representation,
  SANDAG_spec,
  derivative_SANDAG_string_representation
)
```

This VDF could also be read in from a project's yaml file. The "vdfs" entry of the parameters file would contain:

```
vdfs:
  SANDAG:
    functional_form: >
      fftime * (1.0 + alpha_1 * (link_flows/capacity)**beta_1) +
      0.5 * cycle_time * (1 - green_to_cycle_ratio)**2 * (1.0 + alpha_2 * (link_flows/
↪intersection_capacity)**beta_2)
    derivative_functional_form: >
      (fftime * alpha_1 * beta_1 * (link_flows / capacity) ** (beta_1 - 1) / capacity
      + 0.5 * cycle_time * (1 - green_to_cycle_ratio) ** 2
      * alpha_2 * beta_2 * (link_flows / intersection_capacity) ** (beta_2 - 1) /␣
↪intersection_capacity)
    spec:
      alpha_1:
        fill_NA: 0.15
        bounds: [0.0, .inf]
      beta_1:
        fill_NA: 4.0
        bounds: [1.0, .inf]
      alpha_2:
        fill_NA: 0.15
        bounds: [0.0, .inf]
      beta_2:
        fill_NA: 4.0
        bounds: [1.0, .inf]
      green_to_cycle_ratio:
        bounds: [0.0, 1.0]
      cycle_time:
```

(continues on next page)

(continued from previous page)

```
      bounds: [0.0, .inf]
    intersection_capacity:
      bounds: [0.0, .inf]
```

This can then be loaded with:

```
# includes all built-in preset VDFs and any VDFs defined in the project's parameters
all_vdfs = project.project_parameters.get_vdfs()

# only the VDFs explicitly defined in the parameters file
custom_vdfs = project.project_parameters.get_vdfs(exclude_builtins=True)
sandag_yaml_vdf = custom_vdfs["SANDAG"]
```

If a name in the "vdfs" entry collides with one of the built in presets (for example "bpr"), `get_vdfs` raises a `ValueError`.

Checking Custom VDFs

The `check_valid` function on a `VDF` object checks whether the VDF satisfies the following criteria for a well-behaved congestion function described by Spiess (1990):

1. The VDF is strictly increasing - when there is more traffic, the delay increases
2. (part of) For no volume on the link, the congested time is the free flow travel time
3. The VDF's derivative exists and is strictly increasing, to ensure that it is convex

It achieves this by evaluating the VDF and its derivative at a specified number of values of volume between 0 and 3, with a capacity assumed to be 1. It then finds the values of volume / capacity where these conditions were violated, and prints out the result.

For example, the built-in INRETS VDF is non-convex:

```
from aequilibrae.traffic.vdf import inrets

num_points = 300
link_attributes = {"alpha": np.full(num_points, 0.9, dtype=np.float64)}

# true,         true,               true,                    false
valid_0_value, increasing_f_vals, nonnegative_derivative, convex = inrets.check_
↪valid(num_points, link_attributes)

# Prints out:
# The VDF is non-convex due to its derivative decreasing at these values of volume/
↪capacity: [np.float64(1.0033444816053512)]
```

7.2.9 References and Further Reading

BPR Function:

- Bureau of Public Roads (1964). *Traffic Assignment Manual*. U.S. Department of Commerce.

Conical Function:

- Spiess, H. (1990). "Technical Note—Conical Volume-Delay Functions." *Transportation Science*, 24(2): 153-158. https://doi.org/10.1287/trsc.24.2.153

- Hampton Roads Transportation Planning Organization (2020). *Regional Travel Demand Model V2 Methodology Report*. Available: https://www.hrtpo.org/uploads/docs/2020_HamptonRoads_Modelv2_MethodologyReport.pdf (accessed February 2026)

Akcelik Function:

- Akcelik, R. (1991). "Travel Time Functions for Transport Planning Purposes: Davidson's Function, Its Time Dependent Form and an Alternative Travel Time Function." *Australian Road Research*, 21(3): 49-59.

Custom VDF example:

General Traffic Assignment:

- Sheffi, Y. (1985). *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*. Prentice-Hall, Englewood Cliffs, NJ.
- Patriksson, M. (1994). *The Traffic Assignment Problem: Models and Methods*. VSP, Utrecht.

Multi-class Assignment:

- Zill, J., Camargo, P., Veitch, T., Daisy, N. (2019). "Toll Choice and Stochastic User Equilibrium: Ticking All the Boxes." *Transportation Research Record*, 2673(4): 930-940. https://doi.org/10.1177/0361198119837496

7.3 Traffic Assignment Insights

While single-class equilibrium traffic assignment[1] is mathematically simple, multi-class traffic assignment[2], especially when including monetary costs (e.g. tolls) and multiple classes with different passenger-car equivalent (PCE) factors, requires more sophisticated mathematics.

As it is to be expected, strict convergence of multi-class equilibrium assignments comes at the cost of specific technical requirements and more advanced equilibration algorithms have slightly different requirements.

7.3.1 Technical requirements

This documentation is not intended to discuss in detail the mathematical requirements of multi-class traffic assignment, which can be found on[3].

A few requirements, however, need to be made clear.

- All traffic classes shall have identical free-flow travel times throughout the network
- Each class shall have an unique passenger-car equivalency (PCE) factor for all links
- Volume-delay functions shall be monotonically increasing. *Well behaved* functions are always something we are after

For the conjugate and biconjugate Frank-Wolfe algorithms it is also necessary that the VDFs are differentiable.

7.3.2 Cost function

AequilibraE supports class-specific cost functions, where each class can include the following:

- Passenger-car equivalent (PCE)
- Link-based fixed financial cost components

[1] Wardrop, J.G. (1952) "Some theoretical aspects of road traffic research." Proceedings of the Institution of Civil Engineers 1952, 1(3):325-362. Available in: https://www.icevirtuallibrary.com/doi/abs/10.1680/ipeds.1952.11259

[2] Marcotte, P., Patriksson, M. (2007) "Chapter 10 Traffic Equilibrium - Handbooks in Operations Research and Management Science, Vol 14", Elsevier. Editors Barnhart, C., Laporte, G. https://doi.org/10.1016/S0927-0507(06)14010-4

[3] Zill, J., Camargo, P., Veitch, T., Daisy, N. (2019) "Toll Choice and Stochastic User Equilibrium: Ticking All the Boxes", Transportation Research Record, 2673(4):930-940. Available in: https://doi.org/10.1177%2F0361198119837496

- Value-of-Time (VoT)

7.3.3 Convergence criteria

Convergence in AequilibraE is measured solely in terms of relative gap, which is a somewhat old recommendation[4], but it is still the most used measure in practice, and is detailed below.

$$RelGap = \frac{\sum_a V_a^* * C_a - \sum_a V_a^{AoN} * C_a}{\sum_a V_a^* * C_a}$$

The algorithm's two stop criteria currently used are the maximum number of iterations and the target Relative Gap, as specified above. These two parameters are described in detail in the *Assignment* section, in the *Parameters YAML File*.

7.3.4 Available algorithms

All algorithms have been implemented as a single software class, as the differences between them are simply the step direction and step size after each iteration of all-or-nothing assignment, as shown in the table below

Algorithm	Step direction	Step size
Method of Successive Avergaes	All-or-Nothing Assignment (AoN)	Function of the iteration number
Frank-Wolfe	All-or-Nothing Assignment (AoN)	Optimal value derived from Wardrop's principle
Biconjugate Frank-Wolfe	Biconjugate direction (Current and two previous AoN)	Optimal value derived from Wardrop's principle
Conjugate Frank-Wolfe	Conjugate direction (Current and previous AoN)	Optimal value derived from Wardrop's principle

Note

Our implementations of the conjugate and biconjugate Frank-Wolfe methods should be inherently proportional[5], but we have not yet carried the appropriate testing that would be required for an empirical proof.

Method of Successive Averages (MSA)

This algorithm has been included largely for historical reasons, and we see very little reason to use it. Yet, it has been implemented with the appropriate computation of relative gap computation and supports all the analysis features available.

Frank-Wolfe (FW)

The implementation of Frank-Wolfe in AequilibraE is extremely simple from an implementation point of view, as we use a generic optimizer from SciPy as an engine for the line search, and it is a standard implementation of the algorithm introduced by LeBlanc in 1975[6].

[4] Rose, G., Daskin, M., Koppelman, F. (1988) "An examination of convergence error in equilibrium traffic assignment models", Transportation Research Part B, 22(4):261-274. Available in: https://doi.org/10.1016/0191-2615(88)90003-3

[5] Florian, M., Morosan, C.D. (2014) "On uniqueness and proportionality in multi-class equilibrium assignment", Transportation Research Part B, 70:261-274. Available in: https://doi.org/10.1016/j.trb.2014.06.011

[6] LeBlanc, L.J., Morlok, E.K., Pierskalla, W.P. (1975) "An efficient approach to solving the road network equilibrium traffic assignment problem". Transportation Research, 9(5):309-318. Available in: https://doi.org/10.1016/0041-1647(75)90030-1

Biconjugate Frank-Wolfe (BFW)

The biconjugate Frank-Wolfe algorithm is currently the fastest converging link-based traffic assignment algorithm used in practice, and it is the recommended algorithm for AequilibraE users. Due to its need for previous iteration data, it **requires more memory** during runtime, but very large networks should still fit nicely in systems with 16Gb of RAM.

Conjugate Frank-Wolfe

The conjugate direction algorithm was introduced in 2013[7], which is quite recent if you consider that the Frank-Wolfe algorithm was first applied in the early 1970's, and it was introduced at the same time as its Biconjugate evolution, so it was born outdated.

7.3.5 Implementation details & tricks

A few implementation details and tricks are worth mentioning not because they are needed to use the software, but because they were things we grappled with during implementation, and it would be a shame not register it for those looking to implement their own variations of this algorithm or to slight change it for their own purposes.

- The relative gap is computed with the cost used to compute the All-or-Nothing portion of the iteration, using the accepted flow at that same iterate. Travel costs are recomputed only **AFTER** checking for convergence.
- Every descent algorithm uses the same exact line search: the step size is the root of the directional derivative of the Beckmann objective over `[0, 1]`, which is the line search the conjugate-direction theory assumes. No algorithm-specific cap is applied to the step.
- In some instances, Frank-Wolfe is extremely unstable during the first iterations on assignment, resulting on numerical errors on our line search. We found that setting the step size to the corresponding MSA value (1/current iteration) resulted in the problem quickly becoming stable and moving towards a state where the line search started working properly. This technique was generalized to the conjugate and biconjugate Frank-Wolfe algorithms.

7.3.6 Multi-threaded implementation

AequilibraE's All-or-Nothing assignment (the basis of all the other algorithms) has been parallelized in Python using the threading library, which is possible due to the work we have done with memory management to release Python's Global Interpreter Lock.

Other opportunities for parallelization, such as the computation of costs and its derivatives (required during the line-search optimization step), as well as all linear combination operations for vectors and matrices have been achieved through the use of OpenMP in pure Cython code. These implementations can be found on a file called `parallel_numpy.pyx` if you are curious to look at.

Much of the gains of going back to Cython to parallelize these functions came from making in-place computation using previously existing arrays, as the instantiation of large NumPy arrays can be computationally expensive.

7.3.7 Handling the network

The other important topic when dealing with multi-class assignment is to have a single consistent handling of networks, as in the end there is only physical network across all modes, regardless of access differences to each mode (e.g. truck lanes, high-occupancy lanes, etc.). This handling is often done with something called a *super-network*.

A shared `NetworkMapping` gives all classes a consistent directed network-link order for costs, congestion and reporting. Each class's graph includes only links available to its mode.

Each `GraphMapping` maps its retained routing links back to the shared network order, including expansions of contracted chains. Link loads are projected through these mappings, so classes can use different routing topologies while reporting flows on the same physical network.

[7] Mitradjieva, M., Lindberg, P.O. (2013) "The Stiff Is Moving—Conjugate Direction Frank-Wolfe Methods with Applications to Traffic Assignment". Transportation Science, 47(2):280-293. Available in: https://doi.org/10.1287/trsc.1120.0409

The `Network` and its graph-building methods maintain these correspondences for both project networks and networks constructed directly from link tables.

7.3.8 References

7.4 Traffic Assignment Validation

Like any elaborate algorithm that processes large volumes of data through complex computations, traffic assignment procedures can always be subject to at least one very reasonable question: Are the results right?

For this reason, we have used all equilibrium traffic assignment algorithms available in AequilibraE to solve standard instances used in academia for comparing algorithm results.

Instances can be downloaded here.

All tests were performed with the AequilibraE version 1.7.0.

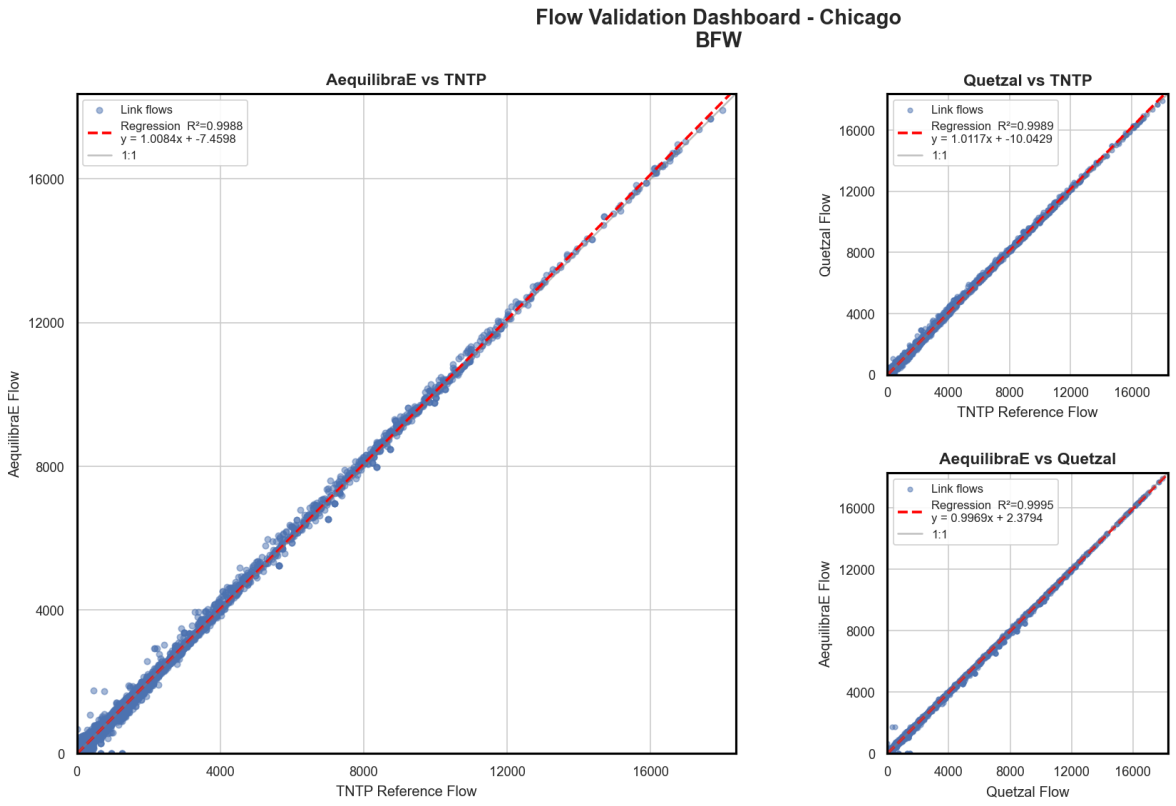
As shown below, the results produced by AequilibraE are within expected, although some differences have been found, particularly for Anaheim. We suspect that there are issues with the reference results and welcome further investigations.

Chicago

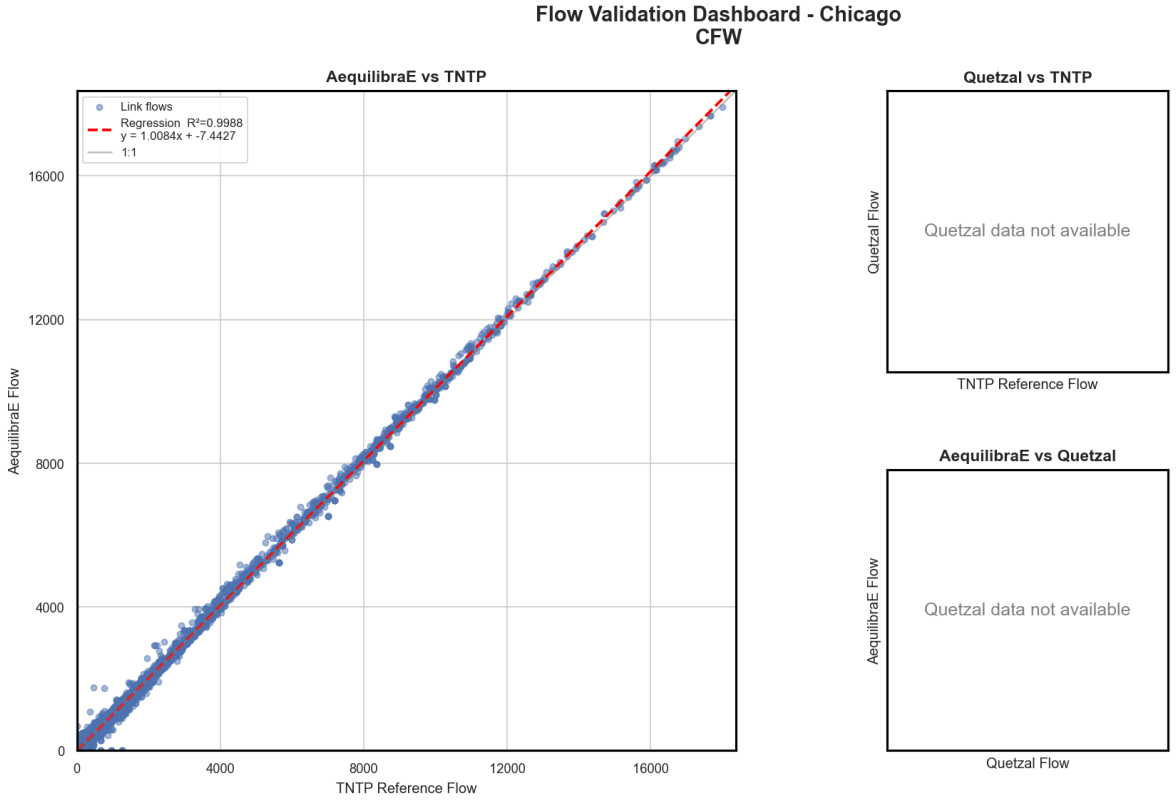
Network stats

- Links: 39,018
- Nodes: 12,982
- Zones: 1,790

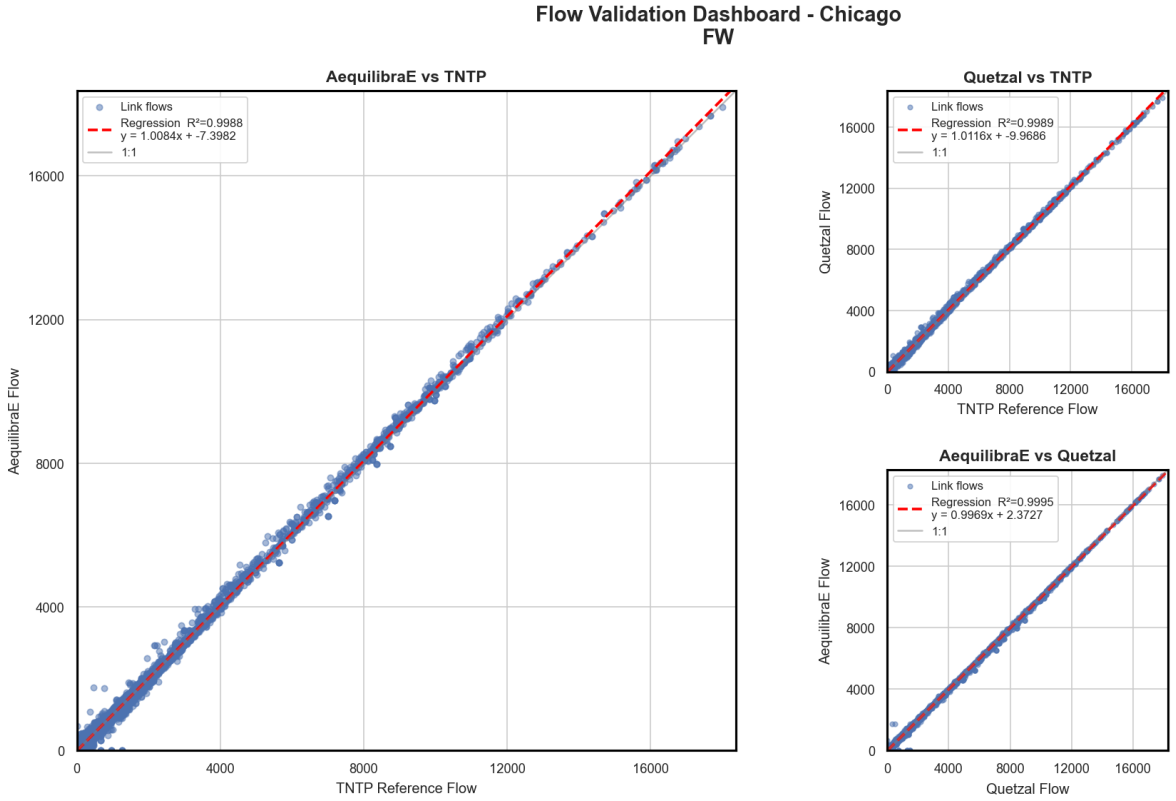
biconjugate Frank-Wolfe



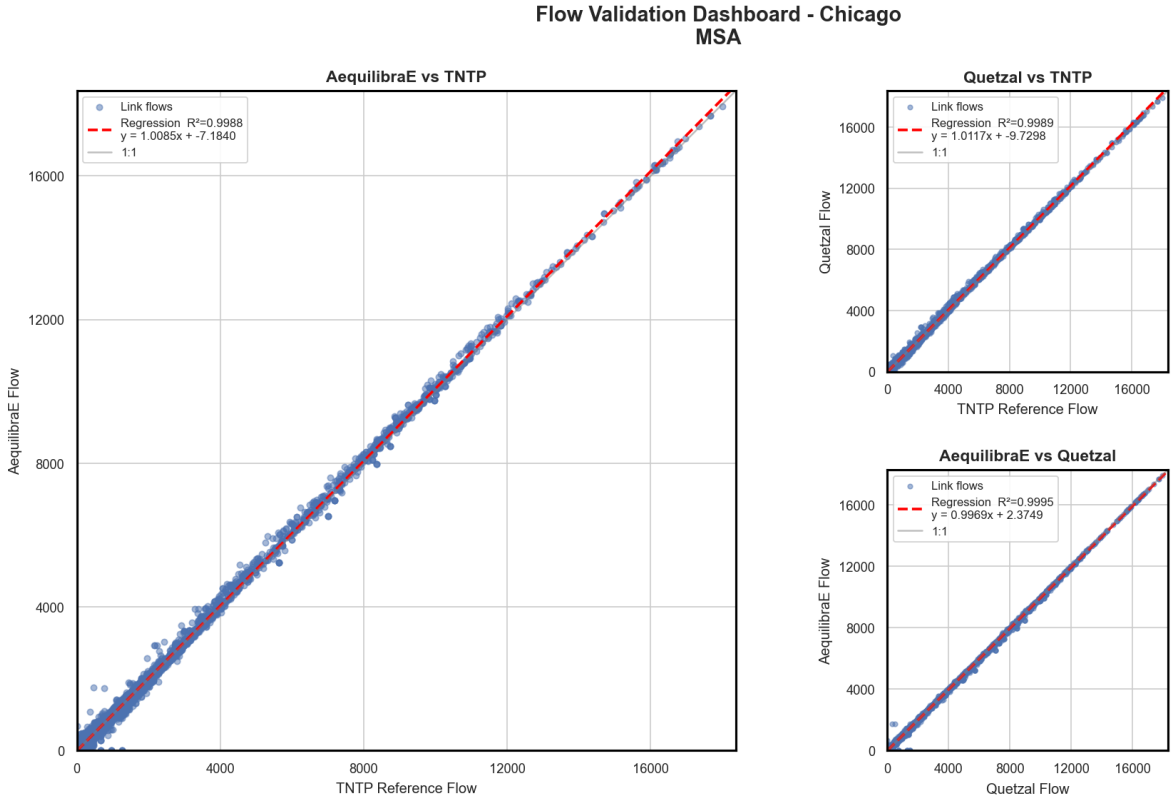
Conjugate Frank-Wolfe



Frank-Wolfe


Flow Validation Dashboard - Chicago
FW
AequilibraE vs TNTP
Link flows
Regression R²=0.9988
y = 1.0084x + -7.3982
1:1
AequilibraE Flow
TNTP Reference Flow
Quetzal vs TNTP
Link flows
Regression R²=0.9989
y = 1.0116x + -9.9686
1:1
Quetzal Flow
TNTP Reference Flow
AequilibraE vs Quetzal
Link flows
Regression R²=0.9995
y = 0.9969x + 2.3727
1:1
AequilibraE Flow
Quetzal Flow

MSA

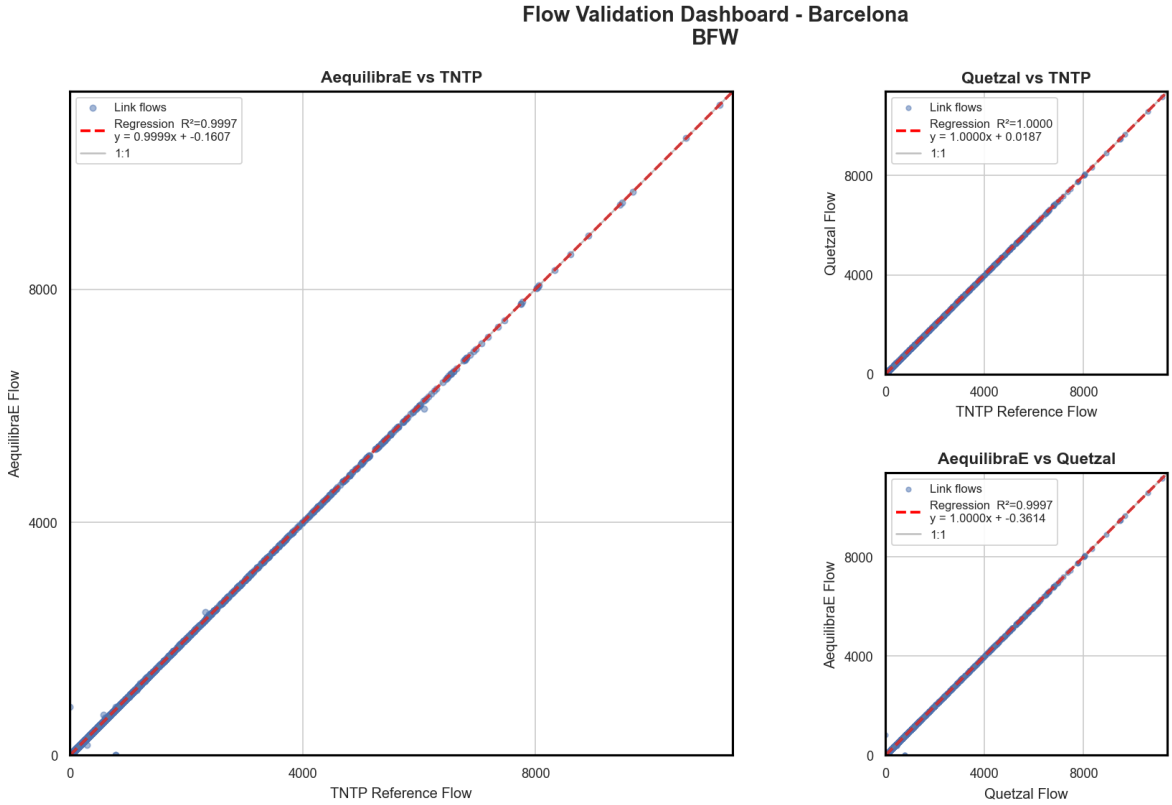


Barcelona

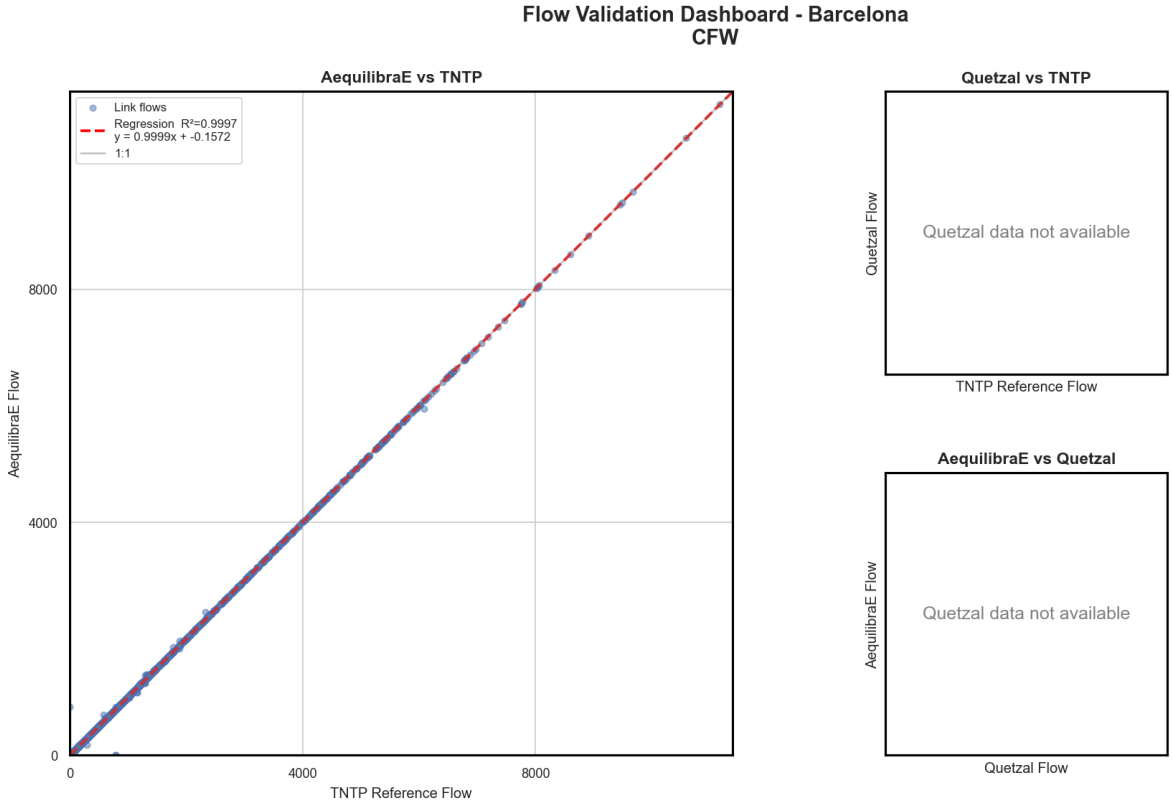
Network stats

- Links: 2,522
- Nodes: 1,020
- Zones: 110

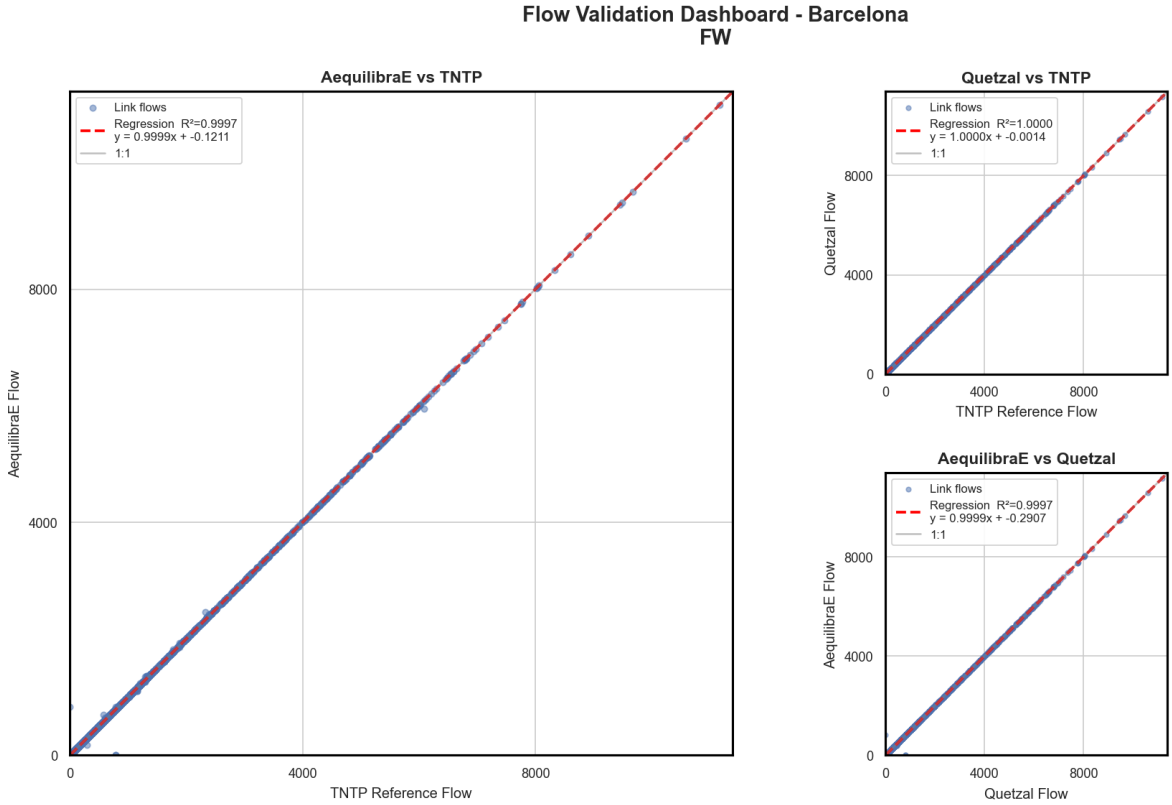
biconjugate Frank-Wolfe



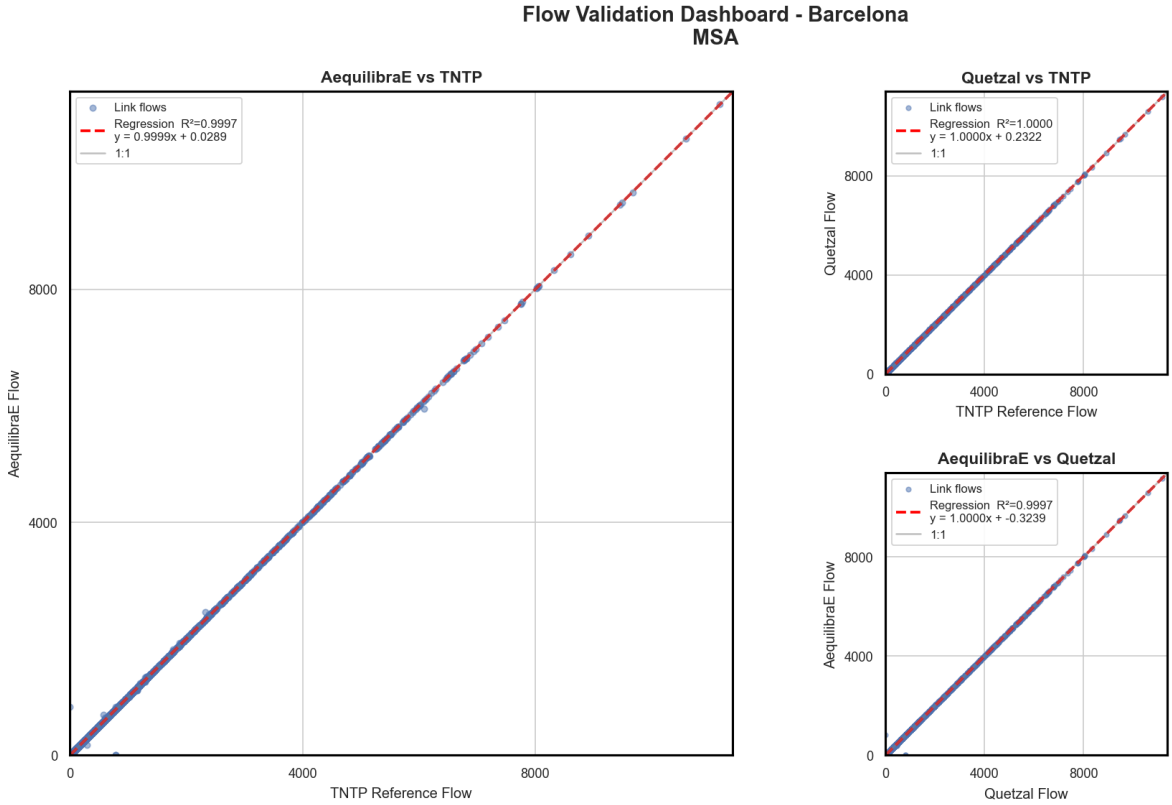
Conjugate Frank-Wolfe



Frank-Wolfe


Flow Validation Dashboard - Barcelona
FW
AequilibraE vs TNTP
Link flows
Regression R²=0.9997
y = 0.9999x + -0.1211
1:1
AequilibraE Flow
TNTP Reference Flow
0
4000
8000
Quetzal vs TNTP
Regression R²=1.0000
y = 1.0000x + -0.0014
Quetzal Flow
AequilibraE vs Quetzal
Regression R²=0.9997
y = 0.9999x + -0.2907

MSA

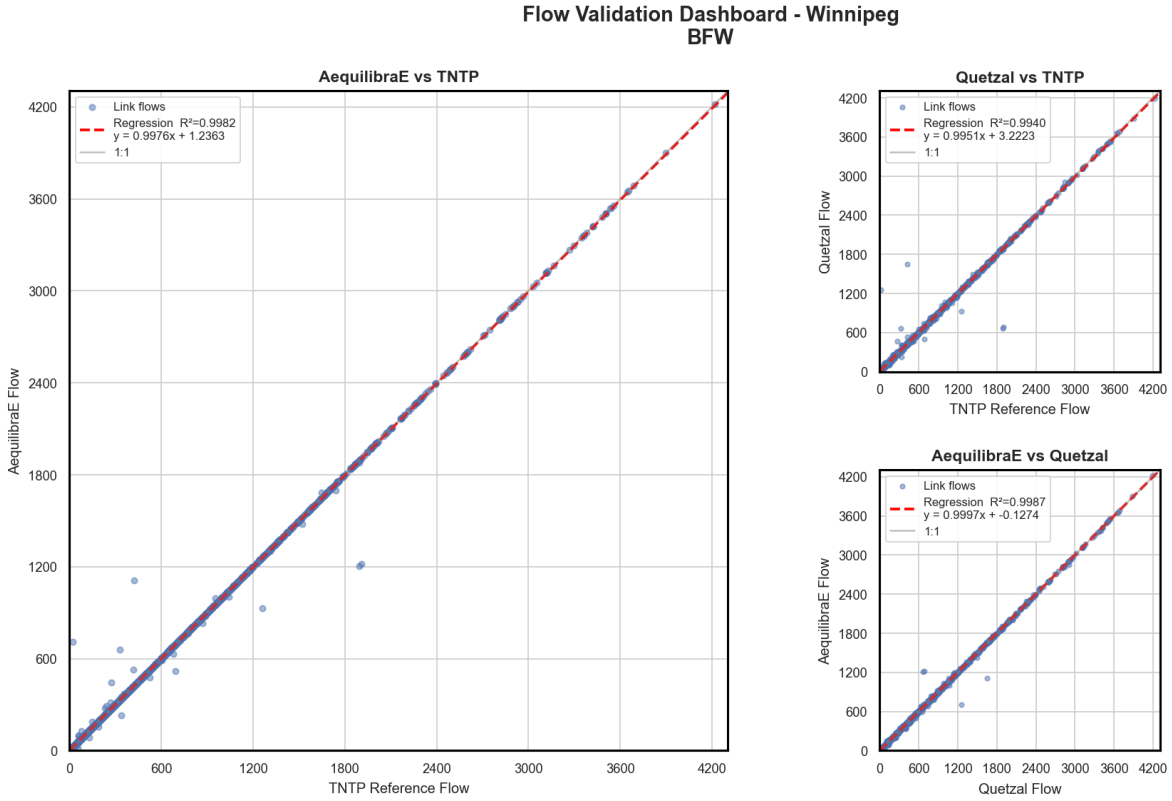


Winnipeg

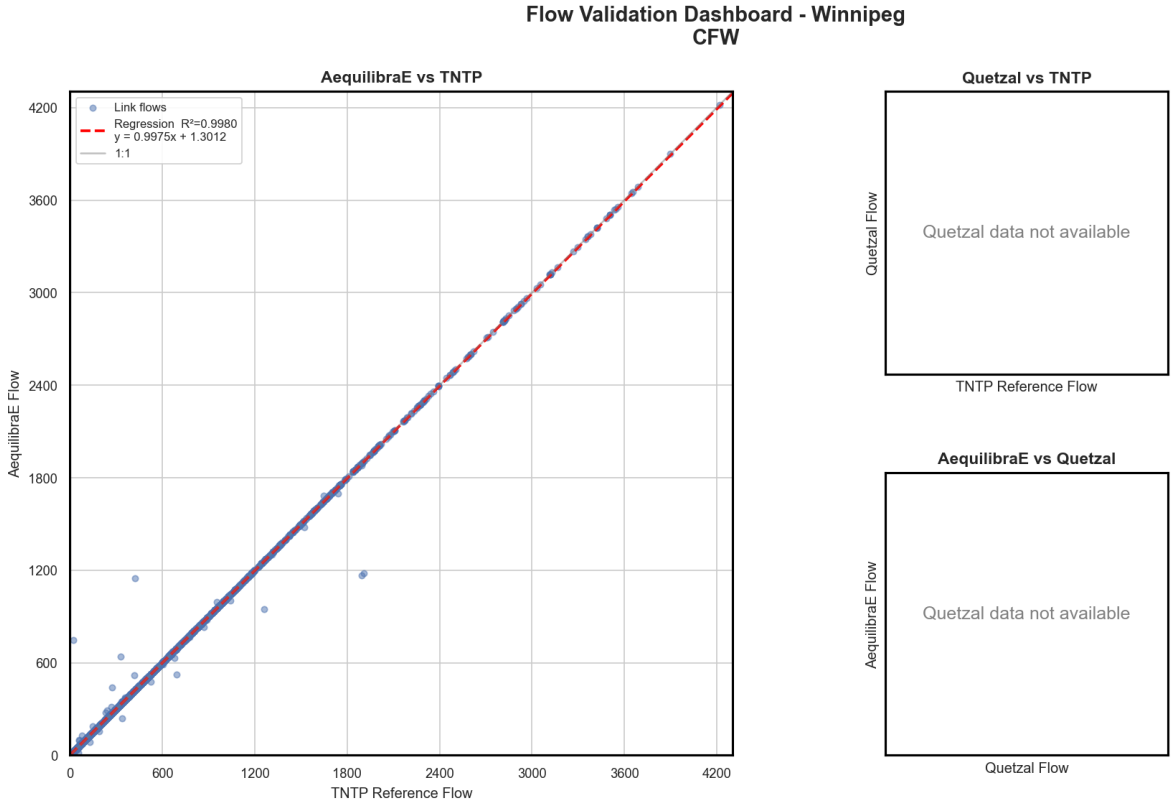
Network stats

- Links: 914
- Nodes: 416
- Zones: 38

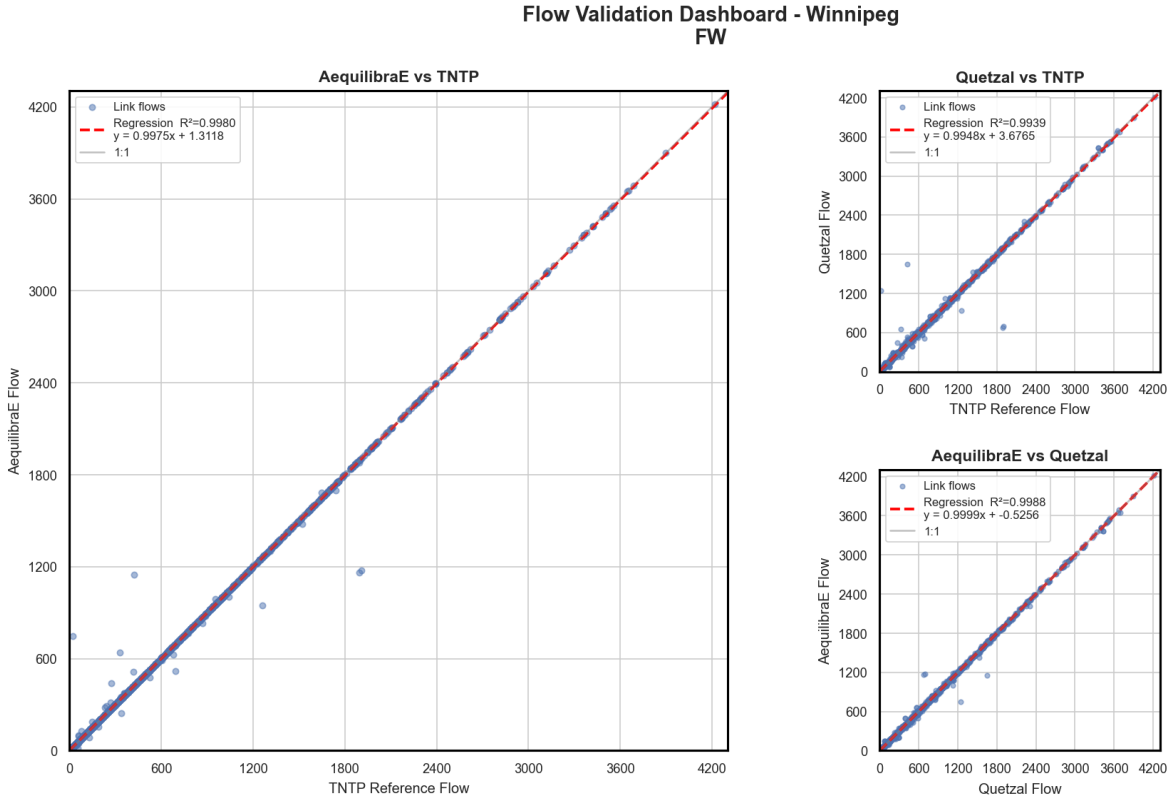
biconjugate Frank-Wolfe



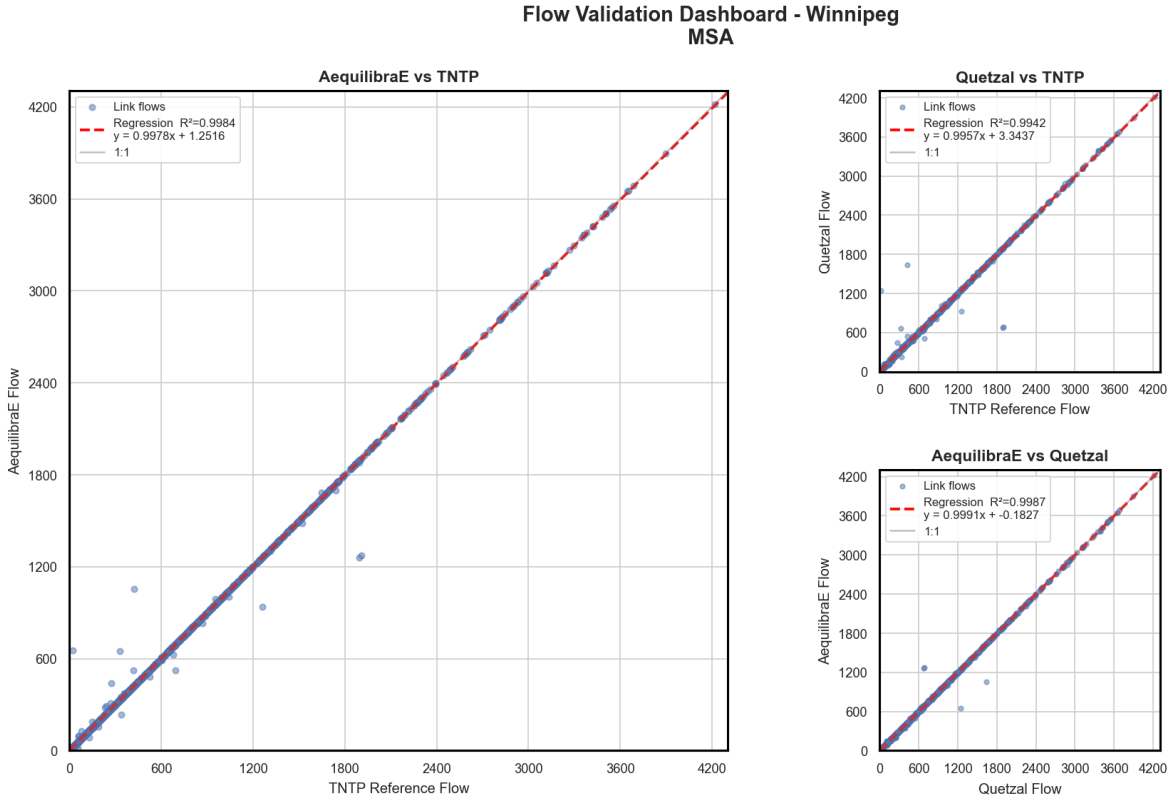
Conjugate Frank-Wolfe



Frank-Wolfe


Flow Validation Dashboard - Winnipeg
FW
AequilibraE vs TNTP
Link flows
Regression R²=0.9980
y = 0.9975x + 1.3118
1:1
AequilibraE Flow
TNTP Reference Flow
Quetzal vs TNTP
Link flows
Regression R²=0.9939
y = 0.9948x + 3.6765
1:1
Quetzal Flow
TNTP Reference Flow
AequilibraE vs Quetzal
Link flows
Regression R²=0.9988
y = 0.9999x + -0.5256
1:1
AequilibraE Flow
Quetzal Flow

MSA

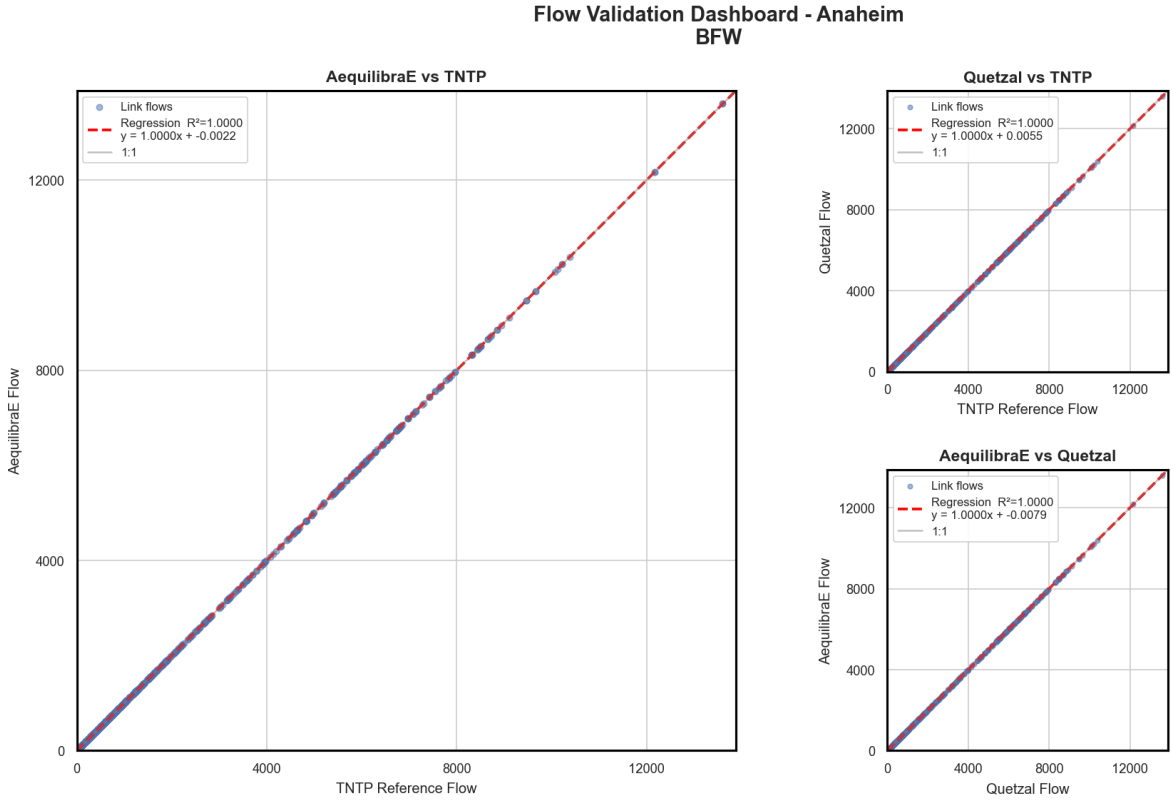


Anaheim

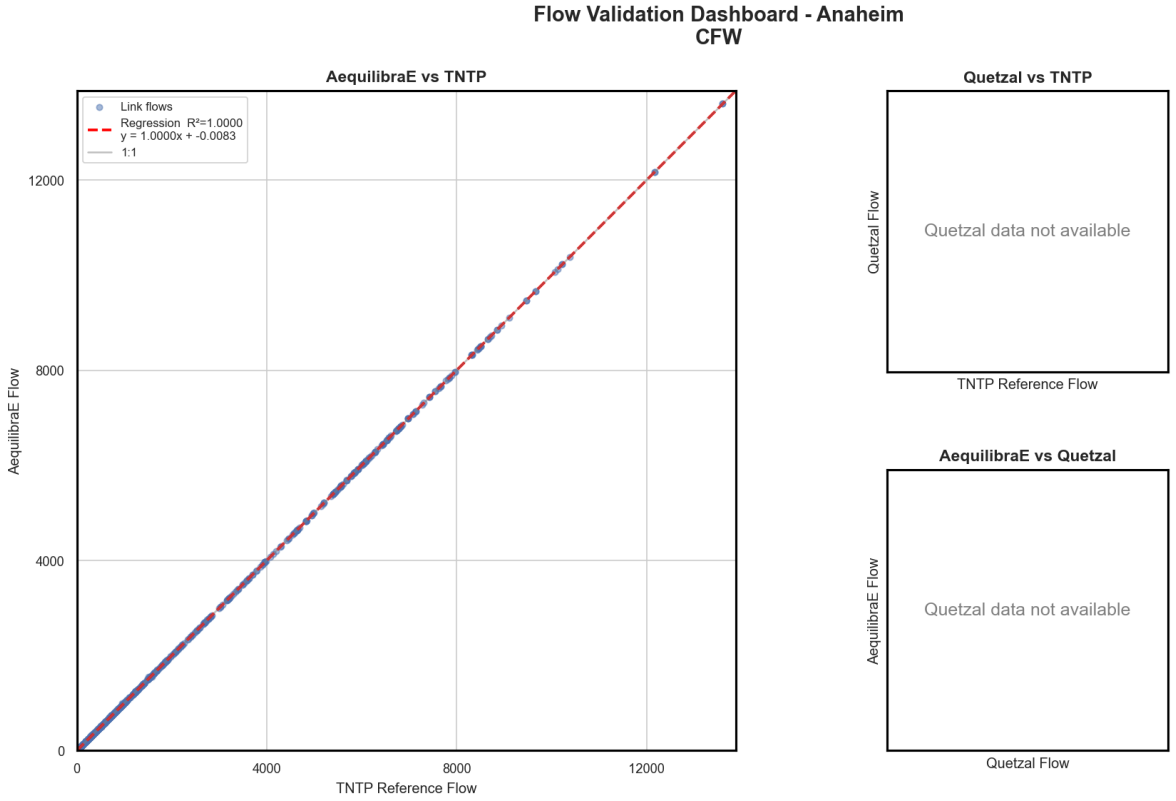
Network stats

- Links: 914
- Nodes: 416
- Zones: 38

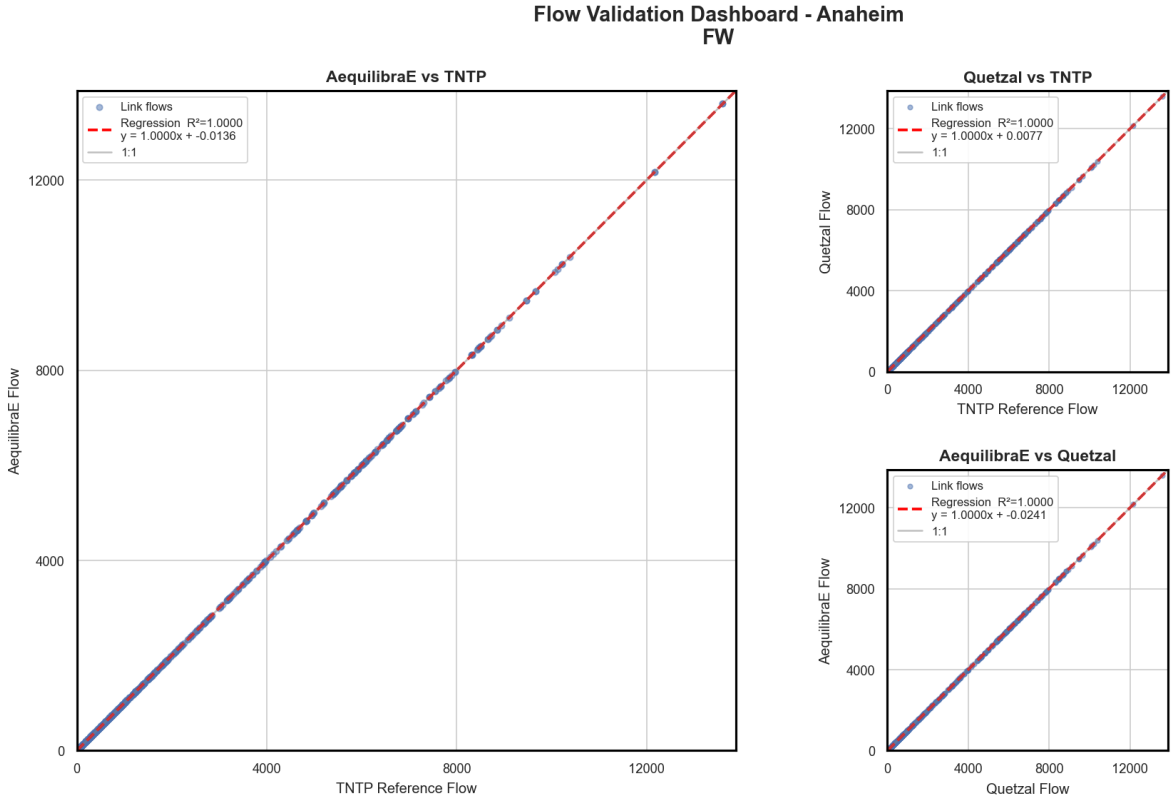
biconjugate Frank-Wolfe



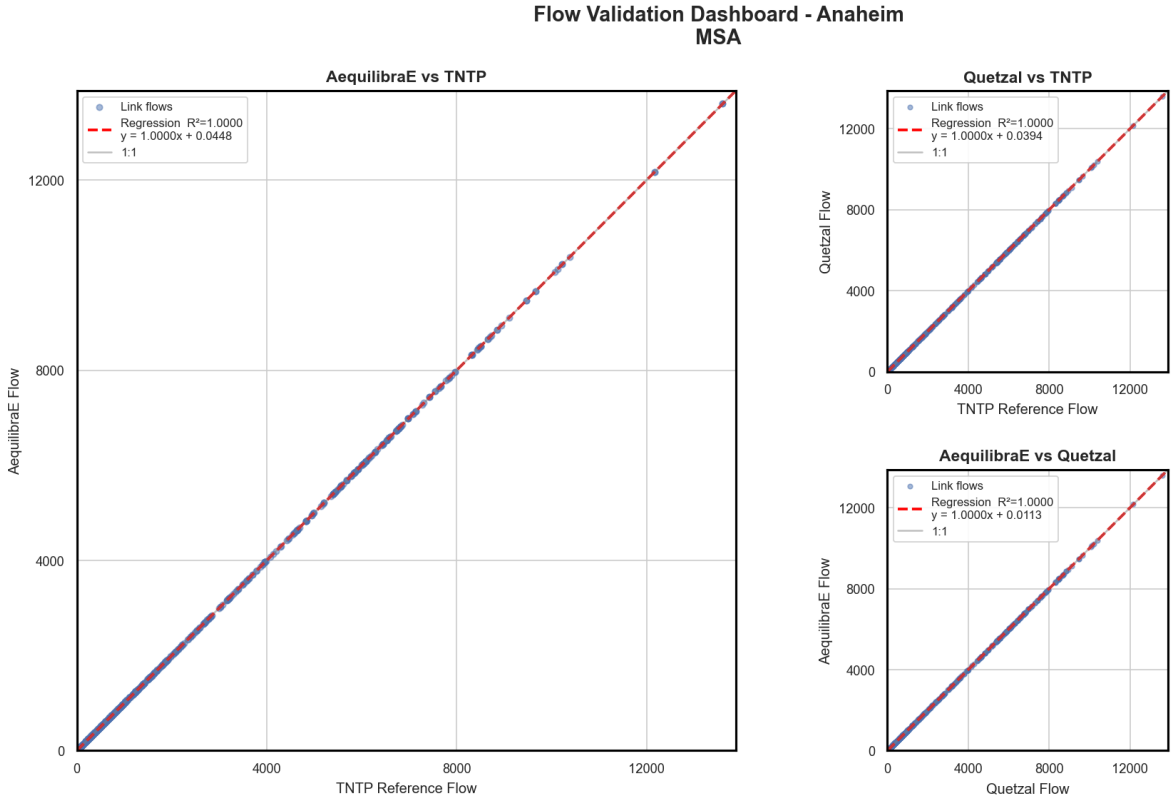
Conjugate Frank-Wolfe



Frank-Wolfe


Flow Validation Dashboard - Anaheim
FW
AequilibraE vs TNTP
Link flows
Regression R²=1.0000
y = 1.0000x + -0.0136
1:1
AequilibraE Flow
TNTP Reference Flow
0
4000
8000
12000
Quetzal vs TNTP
Link flows
Regression R²=1.0000
y = 1.0000x + 0.0077
1:1
Quetzal Flow
TNTP Reference Flow
AequilibraE vs Quetzal
Link flows
Regression R²=1.0000
y = 1.0000x + -0.0241
1:1
AequilibraE Flow
Quetzal Flow

MSA

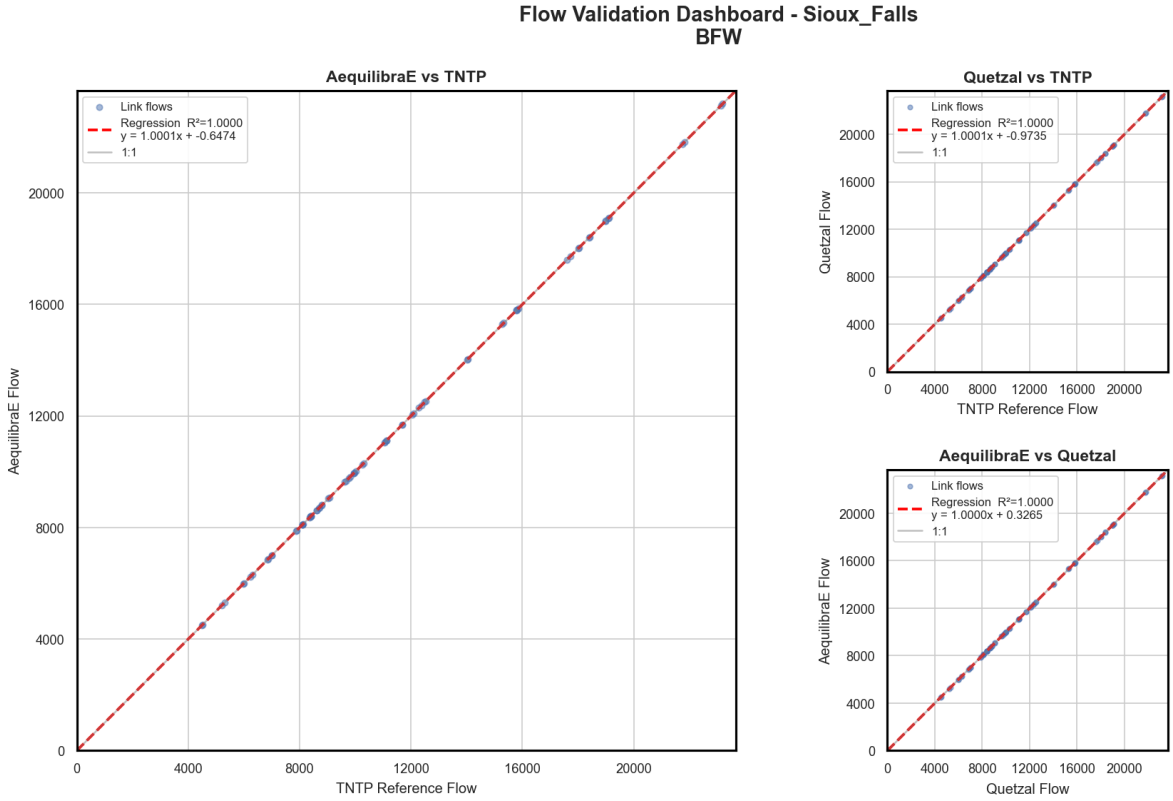


Sioux Falls

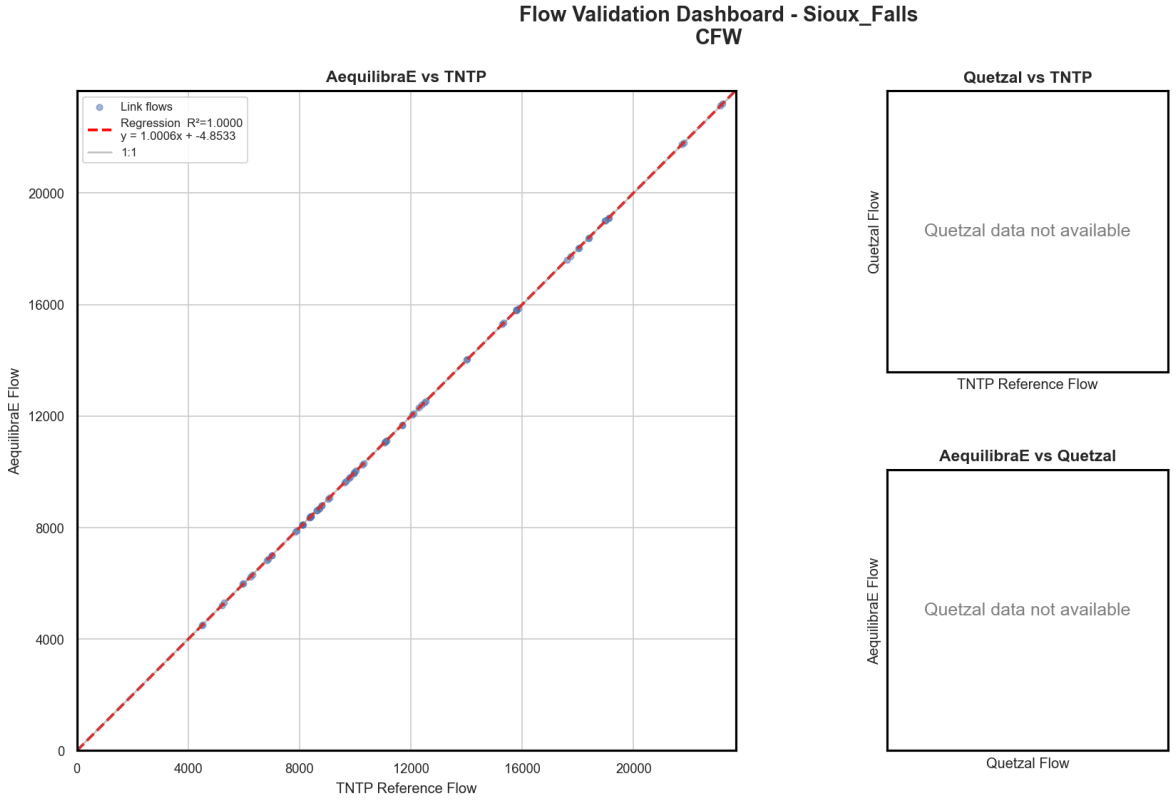
Network stats

- Links: 76
- Nodes: 24
- Zones: 24

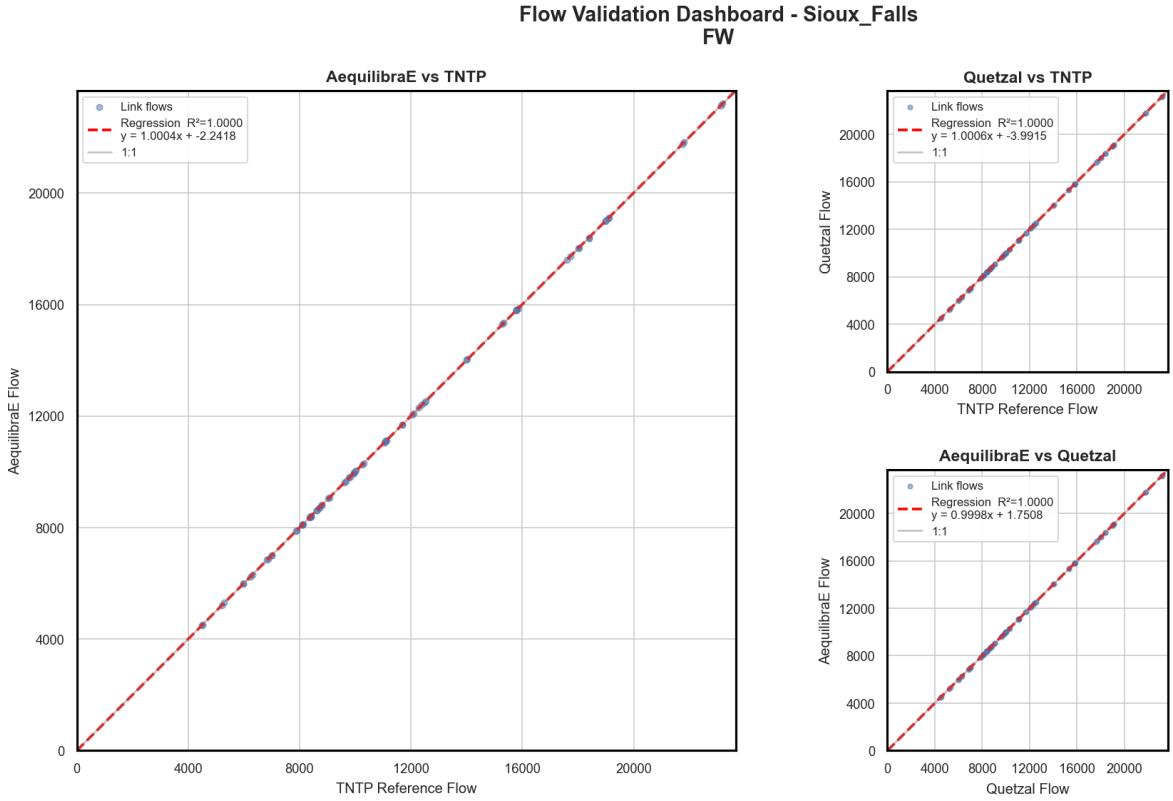
biconjugate Frank-Wolfe



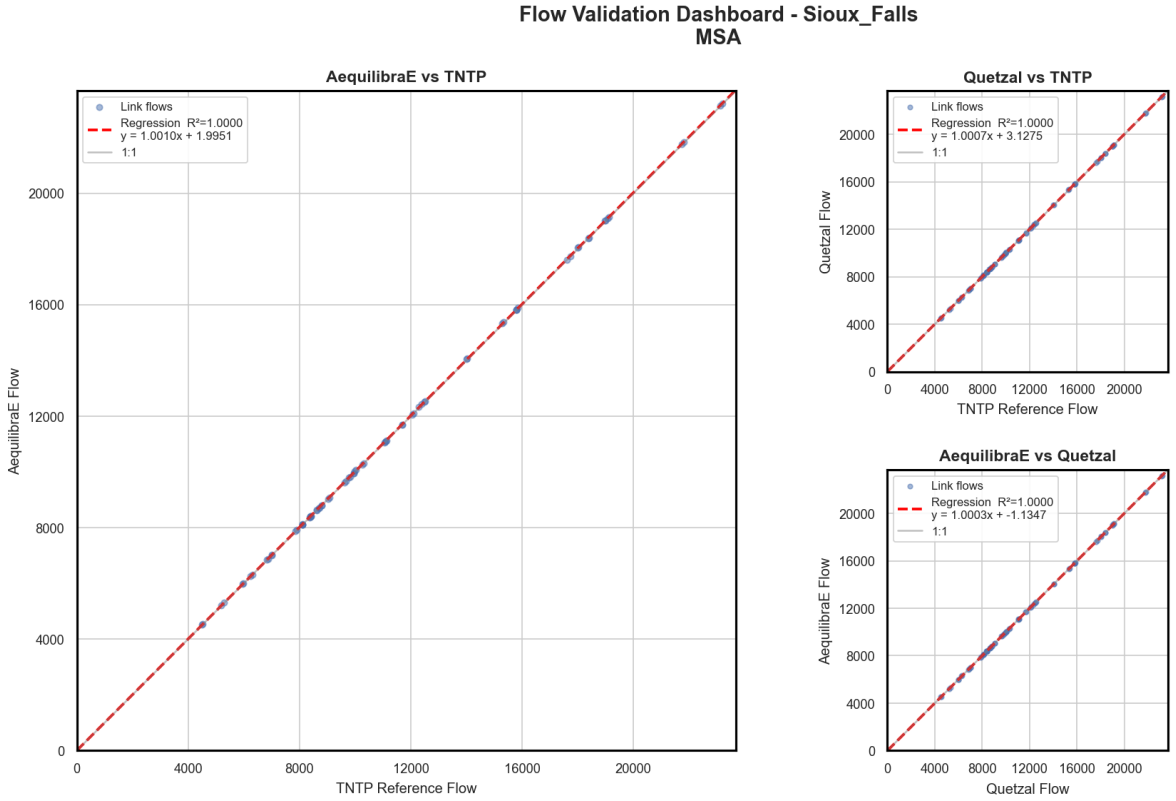
Conjugate Frank-Wolfe



Frank-Wolfe


Flow Validation Dashboard - Sioux_Falls
FW
AequilibraE vs TNTP
Link flows
Regression R²=1.0000
y = 1.0004x + -2.2418
1:1
AequilibraE Flow
TNTP Reference Flow
Quetzal vs TNTP
Link flows
Regression R²=1.0000
y = 1.0006x + -3.9915
1:1
Quetzal Flow
TNTP Reference Flow
AequilibraE vs Quetzal
Link flows
Regression R²=1.0000
y = 0.9998x + 1.7508
1:1
AequilibraE Flow
Quetzal Flow

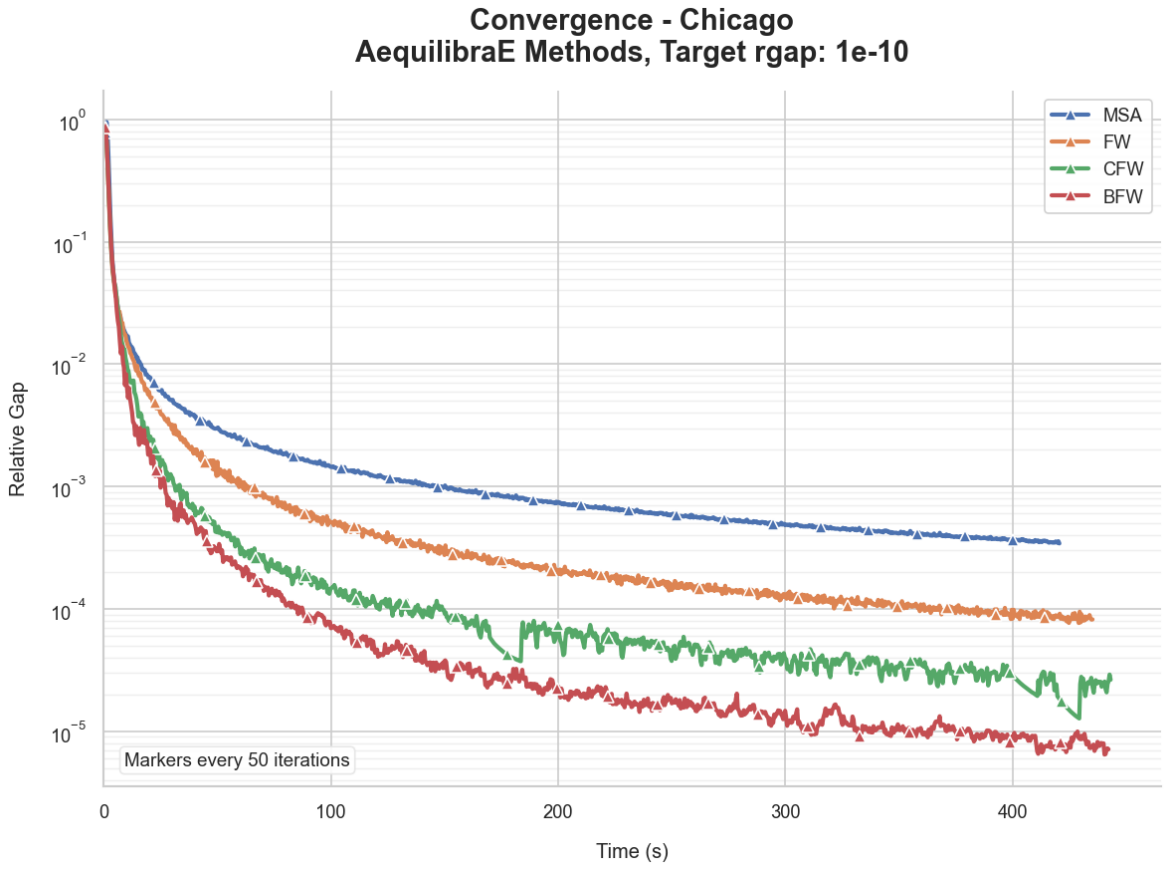
MSA



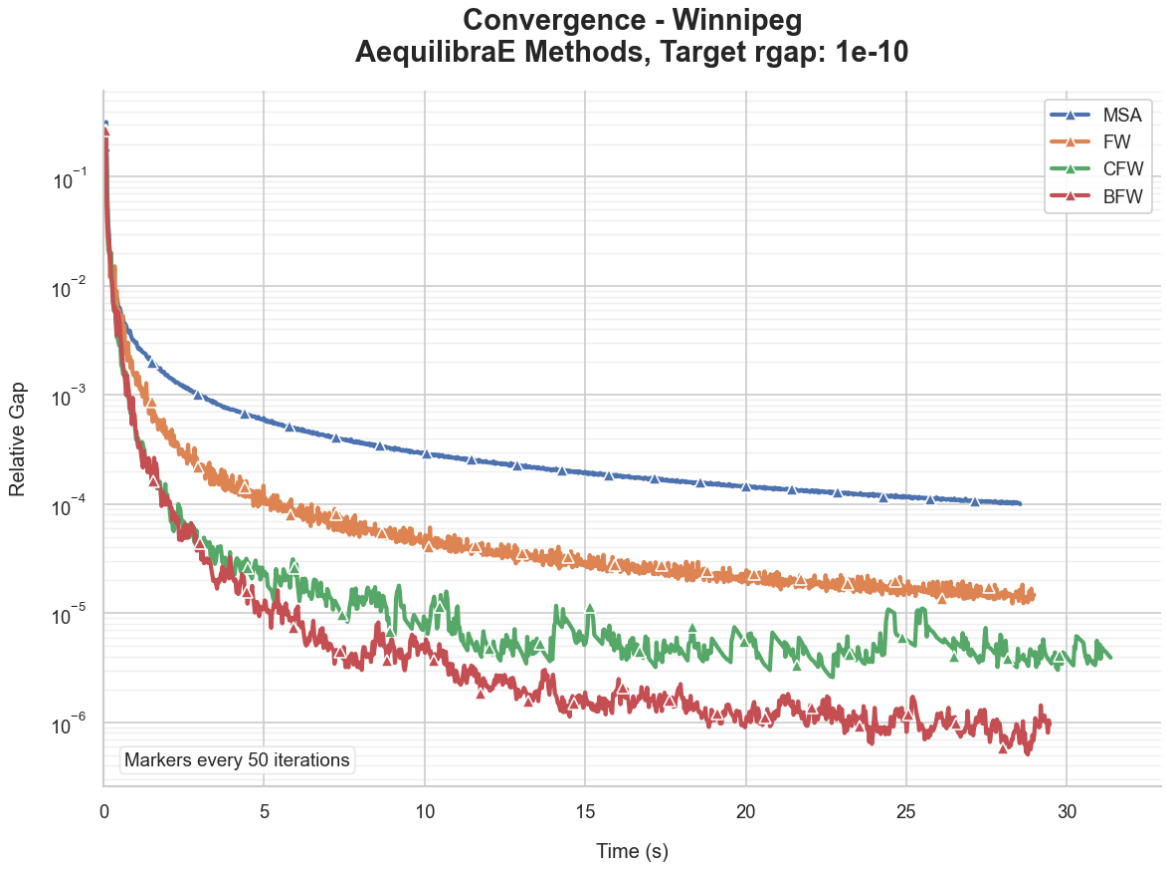
7.4.1 Convergence Study

Besides validating the final results from the algorithms, we have also compared how well they converge for the largest instance we have tested (Chicago Regional), as that instance has a comparable size to real-world models.

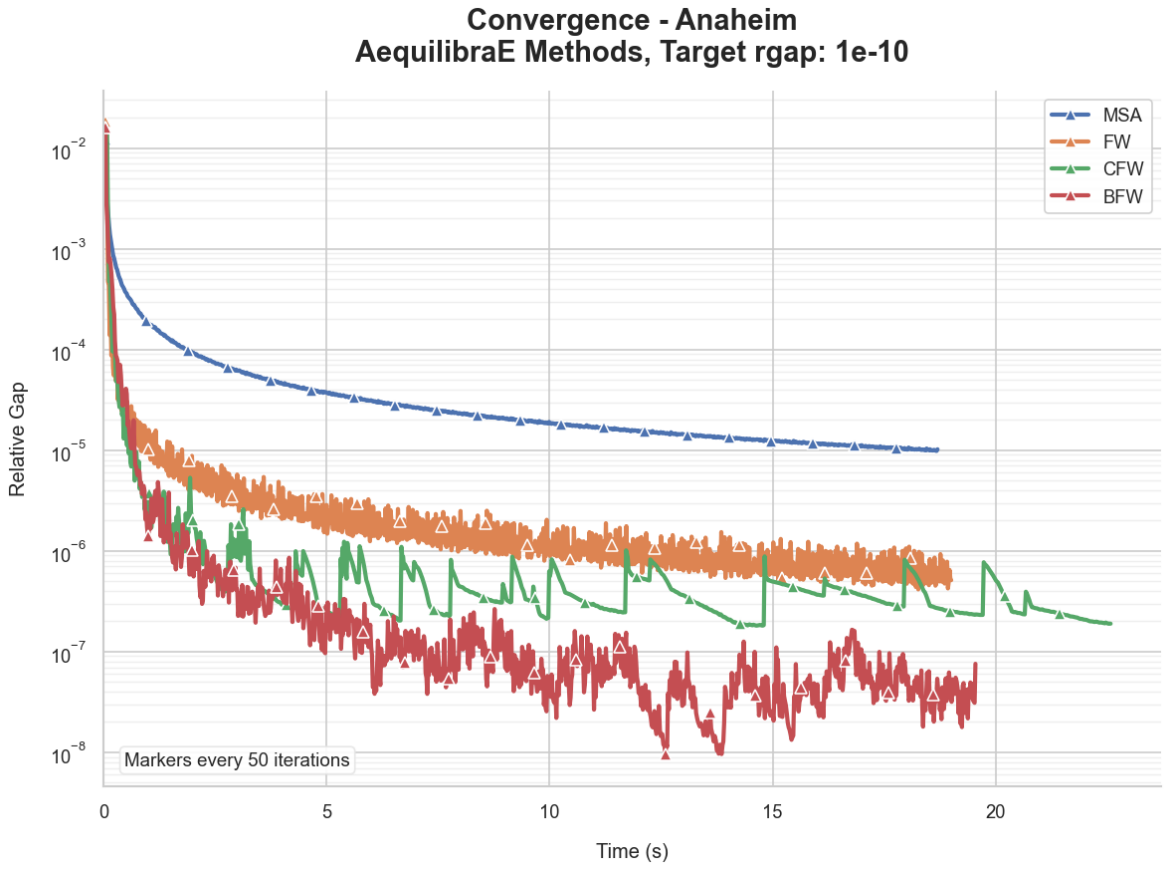
Chicago


Convergence - Chicago
AequilibraE Methods, Target rgap: 1e-10
MSA
FW
CFW
BFW
Relative Gap
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
Markers every 50 iterations
0
100
200
300
400
Time (s)

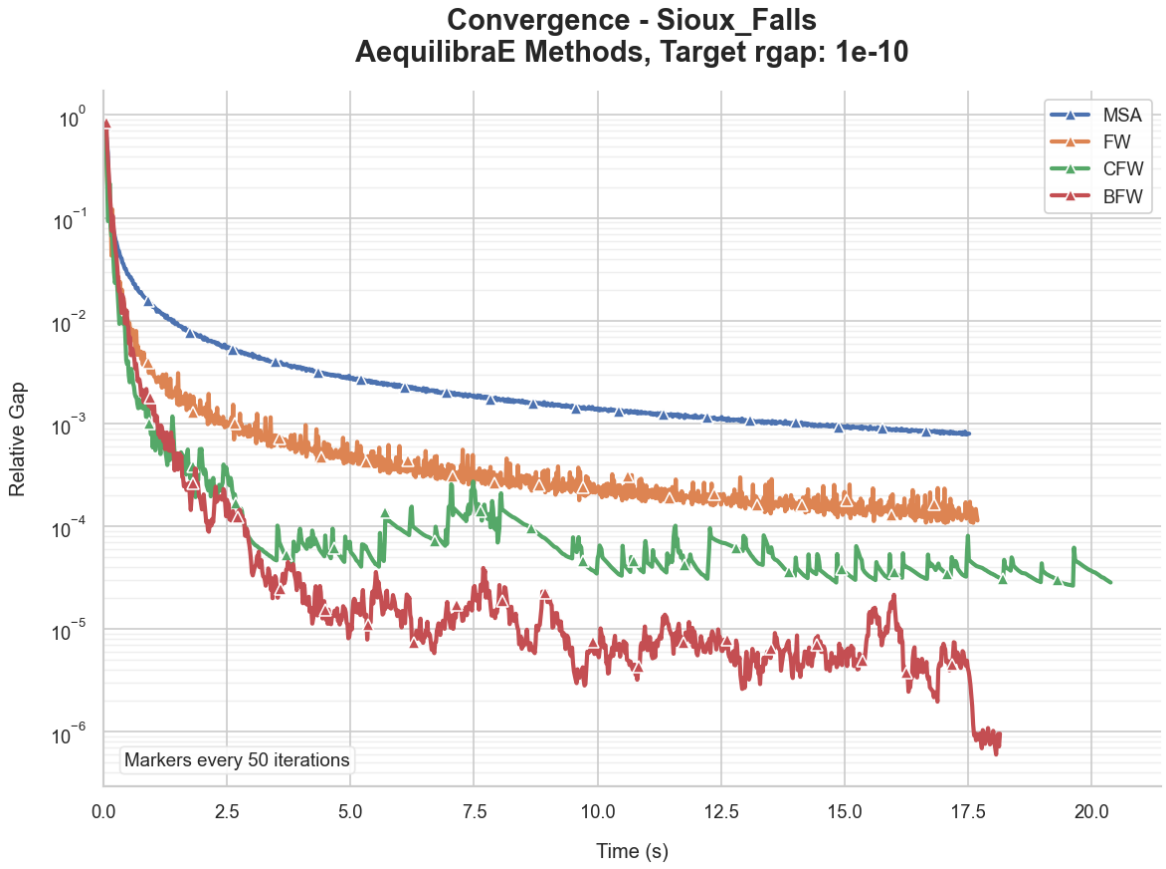
Winnipeg


Convergence - Winnipeg
AequilibraE Methods, Target rgap: 1e-10
MSA
FW
CFW
BFW
Relative Gap
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
Markers every 50 iterations
0
5
10
15
20
25
30
Time (s)

Anaheim


Convergence - Anaheim
AequilibraE Methods, Target rgap: 1e-10
MSA
FW
CFW
BFW
Relative Gap
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
Markers every 50 iterations
0
5
10
15
20
Time (s)

Sioux-Falls



As expected, Frank-Wolfe far outperforms the Method of Successive Averages for a number of iterations larger than 25 in the case of Chicago, and is capable of reaching 1.0e-04 just after 800 iterations, while MSA is still at 3.5e-4 even after 1,000 iterations for that same case.

The actual show, however, is left for the biconjugate Frank-Wolfe implementation, which delivers a relative gap of under 1.0e-04 in under 200 iterations, and a relative gap of under 1.0e-05 in just over 700 iterations.

This convergence capability, allied to its computational performance described below suggest that AequilibraE is ready to be used in large real-world applications.

7.4.2 Computational performance

All tests were run on a desktop equipped with an Ryzen 9 6900HX (**14 cores used only**) running Windows 11 Pro.

On this machine, AequilibraE performed 1,000 iterations of biconjugate Frank-Wolfe assignment on the Chicago Network in around 442 seconds, or around 0.4422s per iteration (other algorithms are as low as 0.42 seconds per iteration).

This performance is on par with that of commercial software.

Note

The biggest opportunity for performance in AequilibraE right now it to apply network contraction hierarchies to the building of the graph, but that is still a long-term goal

7.4.3 Want to run your own convergence study?

If you want to run the convergence study in your machine, with Chicago Regional instance or any other instance presented here, check out the code block below! Please make sure you have already imported TNTP files into your machine.

In the first part of the code, we'll parse TNTP instances to a format AequilibraE can understand, and then we'll perform the assignment.

7.5 Examples

7.5.1 Traffic assignment

Assigning sparse matrices

Modern Activity-Based models (and even some trip-based and tour-based ones) result on incredibly sparse demand matrices, which opens up a significant opportunity to save time during assignment by using early-exiting during the path-computation phase of assignment.

To take advantage of this, while still computing assignment skims, AequilibraE has a built-in method to skim the last iteration after the assignment is done.

Technical references

- *Traffic Assignment Procedure*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- *aequilibrae.traffic.TrafficClass()*
- *aequilibrae.traffic.TrafficAssignment()*

```
# Imports
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

from scipy.sparse import csr_array

from aequilibrae.matrix import from_dict
from aequilibrae.traffic import TrafficAssignment, TrafficClass
from aequilibrae.traffic.vdf import bpr
from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Traffic assignment

```
network = project.road.build_network()
```

Let's get the demand matrix directly from the project record, and inspect what matrices we have in the project.

```
proj_matrices = project.matrices
proj_matrices.get_available_matrices_and_metadata()
```

We get the demand matrix, and prepare it for computation

```
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])

# Copy the named matrices into sparse float64 storage. In practise these would
# be stored in the sparse OMX extension rather than as a dense matrix, then
# converted to a sparse format.
demand = from_dict(
    {name: csr_array(values, dtype="float64") for name, values in demand.items()},
    index=demand.index,
)
```

Let's perform the traffic assignment

```
# Create the assignment class
assigclass = TrafficClass(
    name="car", mode="c", cost="free_flow_time", demand=demand, skimming=[],
    graph_options={"block_centroid_flows": False},
)

# Configure the VDF, network fields, and assignment algorithm.
assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=bpr,
    name_mapping={"alpha": "b", "beta": "power"},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="bfw",
)
assig.execute(max_iterations=10, rgap=0.01)
```

After finishing the assignment, we can skim the last iteration

```
skims = assig.skim_congested(["distance"], return_matrices=True)

# Skims are returned as a dictionary, with the class names as keys
# Let's see all skims we have inside it:
print(skims["car"].field_names)
```

We can save the skims, but we need to choose to only save the final ones, as the blended were not generated

```
assig.save_skims("base_year_assignment_skims", which_ones="final", format="omx")
```

Close the project

```
project.close()
```

Traffic Assignment without an AequilibraE Model

In this example, we show how to perform Traffic Assignment in AequilibraE without a model.

We are using Sioux Falls data, from TNTP.

References

- *Static Traffic Assignment*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- *`aequilibrae.traffic.TrafficClass()`*
- *`aequilibrae.traffic.TrafficAssignment()`*
- `aequilibrae.matrix.aequilibrae_matrix()`

```
# Imports
from aequilibrae.traffic.vdf import bpr
import os
import pandas as pd
import numpy as np
from uuid import uuid4
from tempfile import gettempdir

from aequilibrae.matrix import AequilibraeMatrix
from aequilibrae.routing import Network
from aequilibrae.traffic import TrafficAssignment
from aequilibrae.traffic import TrafficClass
```

We load the example file from the GMNS GitHub repository

```
net_file = "https://raw.githubusercontent.com/bstabler/TransportationNetworks/master/
↪SiouxFalls/SiouxFalls_net.tntp"

demand_file = "https://raw.githubusercontent.com/bstabler/TransportationNetworks/
↪master/SiouxFalls/CSV-data/SiouxFalls_od.csv"

geometry_file = "https://raw.githubusercontent.com/bstabler/TransportationNetworks/
↪master/SiouxFalls/SiouxFalls_node.tntp"
```

Let's use a temporary folder to store our data

```
folder = os.path.join(gettempdir(), uuid4().hex)
```

First we load our demand file. This file has three columns: O, D, and Ton. O and D stand for origin and destination, respectively, and Ton is the demand of each OD pair.

```
dem = pd.read_csv(demand_file)
zones = int(max(dem.O.max(), dem.D.max()))
index = np.arange(zones) + 1
```

Since our OD-matrix is in a different shape than we expect (for Sioux Falls, that would be a 24x24 matrix), we must create our matrix.

```
mtx = np.zeros(shape=(zones, zones))
for element in dem.to_records(index=False):
    mtx[element[0]-1][element[1]-1] = element[2]
```

Now let's create an in-memory AequilibraE matrix with our data.

```
aem = AequilibraeMatrix(index=index, index_name="zones")
aem["matrix"] = mtx
```

Let's import information about our network. As we're loading data in TNTP format, we should do these manipulations.

```
net = pd.read_csv(net_file, skiprows=2, sep="\t", lineterminator=";", header=None)

net.columns = ["newline", "a_node", "b_node", "capacity", "length", "free_flow_time",
↪"b", "power", "speed", "toll", "link_type", "terminator"]

net.drop(columns=["newline", "terminator"], index=[76], inplace=True)
```

```
network = net[['a_node', 'b_node', "capacity", 'free_flow_time', "b", "power"]]
network = network.assign(direction=1)
network["link_id"] = network.index + 1
network = network.astype({"a_node":"int64", "b_node": "int64"})
```

Now we'll import the geometry (as lon/lat) for our network. For Haversine $A*$ path finding, pass these coordinates explicitly to the search and scale estimator. Euclidean $A*$ requires coordinates projected to a suitable local CRS instead. Geometry can otherwise be skipped.

```
geom = pd.read_csv(geometry_file, skiprows=1, sep="\t", lineterminator=";",␣
↪header=None)
geom.columns = ["newline", "lon", "lat", "terminator"]
geom.drop(columns=["newline", "terminator"], index=[24], inplace=True)
geom["node_id"] = geom.index + 1
geom = geom.astype({"node_id": "int64", "lon": "float64", "lat": "float64"}).set_
↪index("node_id")
```

Create the network snapshot from the link table. See *AequilibraE Graphs* for more information about graphs.

```
network = Network(network.assign(modes="c"), centroids=index, lonlat=geom)
```

The named matrix is ready to use in the assignment.

Let's perform our assignment. Feel free to try different algorithms, as well as change the maximum number of iterations and the gap

```
assigclass = TrafficClass(
    name="car", mode="c", cost="free_flow_time", demand=aem, skimming=["free_flow_time
↪"],
    graph_options={"block_centroid_flows": False},
)

assig = TrafficAssignment(
    network=network,
    classes=[assigclass],
    vdf=bpr,
    name_mapping={"alpha": "b", "beta": "power"},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="fw",
)
assig.execute(max_iterations=100, rgap=1e-6)
```

Now let's take a look at the Assignment results

```
assig.results()
```

And at the Assignment report

```
assig.report()
```

Traffic assignment with turn penalty and prohibition

In this example, we run static traffic assignment for Sioux Falls in two scenarios:

1. Base case (no custom turn restrictions)
2. Custom turn controls with: - a turn penalty from link 7 to link 36 - a turn prohibition from link 24 to link 19

All turn directions are 1 and node-based U-turn transitions are allowed.

References

- *Static Traffic Assignment*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.routing.Network.build_graph()*
- *aequilibrae.traffic.TrafficClass()*
- *aequilibrae.traffic.TrafficAssignment()*

Imports

```
from dataclasses import replace
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from aequilibrae.traffic import TrafficAssignment
from aequilibrae.traffic import TrafficClass
from aequilibrae.traffic.vdf import bpr
from aequilibrae.examples import create_example


def run_assignment(network, matrix):
    """Runs a single-class traffic assignment and returns link flow results."""
    traffic_class = TrafficClass(
        name="car",
        mode="c",
        cost="free_flow_time",
        demand=matrix,
        graph_options={"block_centroid_flows": False},
    )
    assignment = TrafficAssignment(
        network=network,
        classes=[traffic_class],
        time_field="free_flow_time",
        capacity_field="capacity",
        vdf=bpr,
        name_mapping={"alpha": "b", "beta": "power"},
        algorithm="bfw",
    )
    assignment.execute(max_iterations=100, rgap=1e-2)

    return assignment.results()
```

We create the Sioux Falls example project inside our temp folder.

```
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "sioux_falls")
```

Load the network and demand matrix.

```
network = replace(project.road.build_network(), allow_uturns=True)

matrix = project.matrices.get_matrix("demand.omx", subset=["matrix"])
```

Base assignment.

```
base_flows = run_assignment(network, matrix)
```

Apply turn controls in memory as node triples equivalent to: - a turn penalty from link 7 to link 36 (dir=1) - a turn prohibition from link 24 to link 19 (dir=1)

```
links = project.tables.links.data.set_index("link_id")
turn_restrictions = pd.DataFrame(
    {
        "from_node": [int(links.loc[7, "a_node"]), int(links.loc[24, "a_node"])],
        "via_node": [int(links.loc[7, "b_node"]), int(links.loc[24, "b_node"])],
        "to_node": [int(links.loc[36, "b_node"]), int(links.loc[19, "b_node"])],
        "penalty": [5.0, np.nan],  # 5.0 minutes penalty, and one prohibited turn
    }
)
# Network configuration is fixed so we will create a validated replacement with the␣
↪new turns.
network = replace(network, turn_restrictions=turn_restrictions)
```

Assignment with custom turn controls.

```
restricted_flows = run_assignment(network, matrix)
```

Plot flows for both cases and the difference.

```
links = project.tables.links.data.set_index("link_id")

plot_df = (
    links[["geometry"]]
    .join(base_flows[["matrix_tot"]].rename(columns={"matrix_tot": "base_flow"}))
    .join(restricted_flows[["matrix_tot"]].rename(columns={"matrix_tot": "restricted_
↪flow"}))
)
plot_df["difference"] = plot_df["restricted_flow"] - plot_df["base_flow"]

max_base = max(plot_df["base_flow"].max(), 1e-9)
max_restricted = max(plot_df["restricted_flow"].max(), 1e-9)
max_diff = max(np.abs(plot_df["difference"]).max(), 1e-9)

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

plot_df.plot(
    ax=axes[0],
    linewidth=4 * plot_df["base_flow"] / max_base,
    color="tab:blue",
)
axes[0].set_title("Base assignment flow")
axes[0].set_axis_off()

plot_df.plot(
    ax=axes[1],
    linewidth=4 * plot_df["restricted_flow"] / max_restricted,
    color="tab:green",
)
axes[1].set_title("With turn penalty/prohibition")
axes[1].set_axis_off()

plot_df.plot(
    ax=axes[2],
```

(continues on next page)

(continued from previous page)

```
    linewidth=3 * np.abs(plot_df["difference"]) / max_diff,
    color="lightgray",
)

plot_df.loc[plot_df["difference"] >= 0].plot(
    ax=axes[2],
    linewidth=3 * np.abs(plot_df.loc[plot_df["difference"] >= 0, "difference"]) / max_
↪diff,
    color="tab:red",
)

plot_df.loc[plot_df["difference"] < 0].plot(
    ax=axes[2],
    linewidth=3 * np.abs(plot_df.loc[plot_df["difference"] < 0, "difference"]) / max_
↪diff,
    color="tab:purple",
)

axes[2].set_title("Flow difference (restricted - base)")
axes[2].set_axis_off()

plt.tight_layout()
```

Close the project.

```
project.close()
```

Forecasting

In this example, we present a full forecasting workflow for the Sioux Falls example model.

We start creating the skim matrices, running the assignment for the base-year, and then distributing these trips into the network. Later, we estimate a set of future demand vectors which are going to be the input of a future year assignment with select link analysis.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- *`aequilibrae.traffic.TrafficClass()`*
- *`aequilibrae.traffic.TrafficAssignment()`*
- `aequilibrae.distribution.ipf()`
- `aequilibrae.distribution.gravity_calibration()`
- `aequilibrae.distribution.gravity_application()`
- `aequilibrae.distribution.synthetic_gravity_model()`

```
# Imports
from uuid import uuid4
```

(continues on next page)

(continued from previous page)

```
from os.path import join
from tempfile import gettempdir

import pandas as pd

from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
from aequilibrae.traffic.vdf import bpr
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Traffic assignment with skimming

In this step, we'll set the skims for the variable `free_flow_time`, and execute the traffic assignment for the base-year.

```
from dataclasses import replace

from aequilibrae.traffic import TrafficAssignment, TrafficClass
```

```
network = project.road.build_network()
```

Let's get the demand matrix directly from its file, and inspect what matrices we have in the project.

```
proj_matrices = project.matrices
proj_matrices.get_available_matrices_and_metadata()
```

Load the demand matrix core for computation.

```
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])
```

Let's perform the traffic assignment

```
# Create the assignment class
assigclass = TrafficClass(
    name="car",
    mode="c",
    cost="free_flow_time",
    demand=demand,
    skimming=["free_flow_time", "distance"],
    # Every node in Sioux Falls is a centroid, so paths must pass through centroids.
    graph_options={"block_centroid_flows": False},
)

# Configure the VDF, network fields, and assignment algorithm.
```

(continues on next page)

(continued from previous page)

```
assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=bpr,
    name_mapping={"alpha": "b", "beta": "power"},
    capacity_field="capacity",
    time_field="free_flow_time",
    algorithm="bfw",
)
assig.execute(max_iterations=1000, rgap=0.001)
```

After finishing the assignment, we can easily see the convergence report.

```
convergence_report = assig.report()
convergence_report.head()
```

And we can also see the results of the assignment

```
results = assig.results()
results.head()
```

We can export our results to CSV or get a Pandas DataFrame, but let's put it directly into the results database

```
assig.save_results("base_year_assignment")
```

And save the skims

```
assig.save_skims("base_year_assignment_skims", which_ones="all", format="omx")
```

Trip distribution

First, let's have a function to plot the Trip Length Frequency Distribution.

```
from math import log10, floor
import matplotlib.pyplot as plt
```

```
def plot_tlfd(demand, skim, name):
    plt.clf()
    b = floor(log10(skim.shape[0]) * 10)
    n, bins, patches = plt.hist(
        np.nan_to_num(skim.flatten(), 0),
        bins=b,
        weights=np.nan_to_num(demand.flatten()),
        density=False,
        facecolor="g",
        alpha=0.75,
    )

    plt.xlabel("Trip length")
    plt.ylabel("Probability")
    plt.title(f"Trip-length frequency distribution for {name}")
    return plt
```

Calibration

We will calibrate synthetic gravity models using the skims for `free_flow_time` that we just generated

```
import numpy as np
from aequilibrae.distribution import GravityCalibration
```

Load the demand matrix core for computation.

```
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])
```

We also need the skims we just saved into our project. We select the congested-time core.

```
imped_core = "free_flow_time_final"
imped = proj_matrices.get_matrix("base_year_assignment_skims_car.omx", subset=[imped_
↪core])

# Where ``free_flow_time_final`` is the congested time for the last iteration.
# Use 75% of the closest zone for intrazonal impedance.
impedance_values = imped[imped_core].copy()
np.fill_diagonal(impedance_values, 0)
intrazonals = np.amin(
    impedance_values, where=impedance_values > 0, initial=impedance_values.max(),␣
↪axis=1
)
intrazonals *= 0.75
np.fill_diagonal(impedance_values, intrazonals)

# Store the prepared core as a separate project matrix.
del imped[imped_core]
imped["final_time_with_intrazonals"] = impedance_values
proj_matrices.save_matrix(
    "base_year_assignment_skims_car_intrazonals.omx",
    matrix=imped,
)
```

Let's calibrate our Gravity Model

```
for function in ["power", "expo"]:
    gc = GravityCalibration(matrix=demand, impedance=imped, function=function, nan_as_
↪zero=True)
    gc.calibrate()
    model = gc.model
    # We save the model
    model.save(join(fldr, f"{function}_model.mod"))

    result_core = next(iter(gc.result_matrix))
    _ = plot_tlfd(
        gc.result_matrix[result_core], imped["final_time_with_intrazonals"], f"
↪{function} model"
    )

    # We can save the result of applying the model as well
    # We can also save the calibration report
```

(continues on next page)

(continued from previous page)

```
    with open(join(fldr, f"{function}_convergence.log"), "w") as otp:
        for r in gc.report:
            otp.write(r + "\n")
```

And let's plot a trip length frequency distribution for the demand itself

```
plt = plot_tlfd(demand["matrix"], imped["final_time_with_intrazonals"], "demand")
plt.show()
```

Forecast

We create a set of 'future' vectors using some random growth factors. We apply the model for inverse power, as the trip frequency length distribution (TFLD) seems to be a better fit for the actual one.

```
from aequilibrae.distribution import Ipf, GravityApplication, SyntheticGravityModel
```

Compute future vectors

First thing to do is to compute the future vectors from our matrix.

```
origins = np.sum(demand["matrix"], axis=1)
destinations = np.sum(demand["matrix"], axis=0)

# Then grow them with some random growth between 0 and 10%, and balance them
orig = origins * (1 + np.random.rand(origins.shape[0]) / 10)
dest = destinations * (1 + np.random.rand(origins.shape[0]) / 10)
dest *= orig.sum() / dest.sum()

vectors = pd.DataFrame({"origins":orig, "destinations":dest}, index=demand.index[:])
```

IPF for the future vectors

Let's balance the future vectors. The output of this step is going to be used later in the traffic assignment for future year.

```
args = {
    "matrix": demand,
    "vectors": vectors,
    "column_field": "destinations",
    "row_field": "origins",
    "nan_as_zero": True,
}

ipf = Ipf(project=project, **args)
ipf.fit()
```

Save the balanced matrix into the project's matrix folder

```
ipf.save_to_project(file_name="demand_ipfd.omx")
```

Impedance

Let's get the base-year assignment skim for car we created before and prepare it for computation

```
imped = proj_matrices.get_matrix(
    "base_year_assignment_skims_car_intrazonals.omx", subset=["final_time_with_
↪intrazonals"]
)
```

If we wanted the main diagonal to not be considered…

```
# np.fill_diagonal(imped["final_time_with_intrazonals"], np.nan)
```

Now we apply the Synthetic Gravity model

```
for function in ["power", "expo"]:
    model = SyntheticGravityModel()
    model.load(join(fldr, f"{function}_model.mod"))

    args = {
        "impedance": imped,
        "vectors": vectors,
        "row_field": "origins",
        "model": model,
        "column_field": "destinations",
        "nan_as_zero": True,
    }

    gravity = GravityApplication(project=project, **args)
    gravity.apply()

    # We get the output matrix and save it to OMX too,
    gravity.save_to_project(file_name=f"demand_{function}_modeled.omx")
```

We inspect the matrix files and verify that the new matrices are indeed there

```
proj_matrices.get_available_matrices_and_metadata()
```

Traffic assignment with Select Link Analysis

We'll perform traffic assignment for the future year. Let's get our future demand matrix, which corresponds to the IPF result we just saved, and see what is the core we ended up getting. It should be `matrix`.

```
demand = proj_matrices.get_matrix("demand_ipfd.omx", subset=["matrix"])
list(demand)
```

The selected matrix core is ready for computation.

The future year assignment is quite similar to the one we did for the base-year.

```
# So, let's create the assignment class
assigclass = TrafficClass(
    name="car",
    mode="c",
    cost="free_flow_time",
```

(continues on next page)

(continued from previous page)

```
    demand=demand,
    skimming=["free_flow_time", "distance"],
    # Every node in Sioux Falls is a centroid, so paths must pass through centroids.
    graph_options={"block_centroid_flows": False},
)
```

Now we select two sets of links to execute select link analysis.

```
select_links = {
    "Leaving node 1": [(1, 1), (2, 1)],
    "Random nodes": [(3, 1), (5, 1)],
}
```

Note

As we are executing the select link analysis on a particular `TrafficClass`, we should set the links we want to analyze. The input is a dictionary with string as keys and a list of tuples as values, so that each entry represents a separate set of selected links to compute.

`select_link_dict = {"set_name": [(link_id1, direction1), ..., (link_id, direction)]}`

The string name will name the set of links, and the list of tuples is the list of selected links in the form `(link_id, direction)`, as it occurs in the *Graph*.

Direction can be one of 0, 1, or -1, where 0 denotes bi-directionality.

```
# We call this command on the class we are analyzing with our dictionary of values
assigclass = replace(assigclass, select_link_sets=select_links)

# Configure the VDF, network fields, and assignment algorithm.
assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=bpr,
    name_mapping={"alpha": "b", "beta": "power"},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="bfw",
)
assig.execute(max_iterations=500, rgap=0.00001)
```

To save our select link results, all we need to do is provide it with a name. In addition to exporting the select link flows, it also exports the Select Link matrices in OMX format.

```
assig.save_select_link_results("select_link_analysis")
```

Note

Say we just want to save our select link flows, we can call: `assig.save_select_link_flows("just_flows")`

Or if we just want the select link matrices: `assig.save_select_link_matrices("just_matrices")`

Internally, the `save_select_link_results` calls both of these methods at once.

We can export the results to CSV or AequilibraE Data, but let's put it directly into the results database

```
assig.save_results("future_year_assignment")
```

And save the skims

```
assig.save_skims("future_year_assignment_skims", which_ones="all", format="omx")
```

Run convergence study

```
df = assig.report()
x = df.iteration.values
y = df.rgap.values

fig = plt.figure()
ax = fig.add_subplot(111)

plt.plot(x, y, "k--")
plt.yscale("log")
plt.grid(True, which="both")
plt.xlabel("Iterations")
plt.ylabel("Relative Gap")
plt.show()
```

Close the project

```
project.close()
```

Forecasting

In this example, we present a full forecasting workflow for the Sioux Falls example model.

We start creating the skim matrices, running the assignment for the base-year, and then distributing these trips into the network. Later, we estimate a set of future demand vectors which are going to be the input of a future year assignment with select link analysis.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- *`aequilibrae.traffic.TrafficClass()`*
- *`aequilibrae.traffic.TrafficAssignment()`*
- `aequilibrae.distribution.ipf()`
- `aequilibrae.distribution.gravity_calibration()`
- `aequilibrae.distribution.gravity_application()`
- `aequilibrae.distribution.synthetic_gravity_model()`

```
# Imports
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import pandas as pd

from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
fldr
```

```
# We'll also apply a basic logging configuration.

basic_config()

project.parameters
```

Traffic assignment with skimming

In this step, we'll set the skims for the variable `free_flow_time`, and execute the traffic assignment for the base-year.

```
from dataclasses import replace
from aequilibrae.traffic import TrafficAssignment, TrafficClass, VDF
```

```
network = project.road.build_network()
```

Let's get the demand matrix directly from its file, and inspect what matrices we have in the project.

```
proj_matrices = project.matrices
proj_matrices.get_available_matrices_and_metadata()
```

Load the demand matrix core for computation.

```
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])
```

Let's perform the traffic assignment

```
# Create the assignment class
assigclass = TrafficClass(
    name="car",
    mode="c",
    cost="free_flow_time",
    demand=demand,
    skimming=["free_flow_time", "distance"],
    # Every node in Sioux Falls is a centroid, so paths must pass through centroids.
    graph_options={"block_centroid_flows": False},
)
```

(continues on next page)

(continued from previous page)

```
from aequilibrae.routing.cython.vdf_core import akcelik, bpr, bpr2, conical, delta_
↪akcelik, delta_bpr, delta_bpr2, delta_conical, delta_inrets, inrets

bpr_spec = {
    "alpha": {"fill_NA": 0.15, "bounds": (0.0, float("inf"))},
    "beta": {"fill_NA": 4.0, "bounds": (1.0, float("inf"))},
}
vdf = VDF("bpr_example", bpr, bpr_spec, delta_bpr)

# Configure the VDF, network fields, and assignment algorithm.
assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=vdf,
    name_mapping={"alpha": "b", "beta": "power"},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="bfw",
)
assig.execute(max_iterations=1000, rgap=0.001)
```

After finishing the assignment, we can easily see the convergence report.

```
convergence_report = assig.report()
convergence_report.head()
```

And we can also see the results of the assignment

```
results = assig.results()
results.head()
```

We can export our results to CSV or get a Pandas DataFrame, but let's put it directly into the results database

```
assig.save_results("base_year_assignment")
```

And save the skims

```
assig.save_skims("base_year_assignment_skims", which_ones="all", format="omx")
```

Trip distribution

First, let's have a function to plot the Trip Length Frequency Distribution.

```
from math import floor, log10

import matplotlib.pyplot as plt
```

```
def plot_tlfd(demand, skim, name):
    plt.clf()
```

(continues on next page)

(continued from previous page)

```
    b = floor(log10(skim.shape[0]) * 10)
    n, bins, patches = plt.hist(
        np.nan_to_num(skim.flatten(), 0),
        bins=b,
        weights=np.nan_to_num(demand.flatten()),
        density=False,
        facecolor="g",
        alpha=0.75,
    )

    plt.xlabel("Trip length")
    plt.ylabel("Probability")
    plt.title(f"Trip-length frequency distribution for {name}")
    return plt
```

Calibration

We will calibrate synthetic gravity models using the skims for `free_flow_time` that we just generated

```
import numpy as np

from aequilibrae.distribution import GravityCalibration
```

Load the demand matrix core for computation.

```
demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])
```

We also need the skims we just saved into our project. We select the congested-time core.

```
imped_core = "free_flow_time_final"
imped = proj_matrices.get_matrix("base_year_assignment_skims_car.omx", subset=[imped_
↪core])

# Where ``free_flow_time_final`` is the congested time for the last iteration.
# Use 75% of the closest zone for intrazonal impedance.
impedance_values = imped[imped_core].copy()
np.fill_diagonal(impedance_values, 0)
intrazonals = np.amin(
    impedance_values, where=impedance_values > 0, initial=impedance_values.max(),␣
↪axis=1
)
intrazonals *= 0.75
np.fill_diagonal(impedance_values, intrazonals)

# Store the prepared core as a separate project matrix.
del imped[imped_core]
imped["final_time_with_intrazonals"] = impedance_values
proj_matrices.save_matrix(
    "base_year_assignment_skims_car_intrazonals.omx",
    matrix=imped,
)
```

Let's calibrate our Gravity Model

```
for function in ["power", "expo"]:
    gc = GravityCalibration(matrix=demand, impedance=imped, function=function, nan_as_
↪zero=True)
    gc.calibrate()
    model = gc.model
    # We save the model
    model.save(join(fldr, f"{function}_model.mod"))

    result_core = next(iter(gc.result_matrix))
    _ = plot_tlfd(
        gc.result_matrix[result_core], imped["final_time_with_intrazonals"], f"
↪{function} model"
    )

    # We can save the result of applying the model as well
    # We can also save the calibration report
    with open(join(fldr, f"{function}_convergence.log"), "w") as otp:
        for r in gc.report:
            otp.write(r + "\n")
```

And let's plot a trip length frequency distribution for the demand itself

```
plt = plot_tlfd(demand["matrix"], imped["final_time_with_intrazonals"], "demand")
plt.show()
```

Forecast

We create a set of 'future' vectors using some random growth factors. We apply the model for inverse power, as the trip frequency length distribution (TFLD) seems to be a better fit for the actual one.

```
from aequilibrae.distribution import GravityApplication, Ipf, SyntheticGravityModel
```

Compute future vectors

First thing to do is to compute the future vectors from our matrix.

```
origins = np.sum(demand["matrix"], axis=1)
destinations = np.sum(demand["matrix"], axis=0)

# Then grow them with some random growth between 0 and 10%, and balance them
orig = origins * (1 + np.random.rand(origins.shape[0]) / 10)
dest = destinations * (1 + np.random.rand(origins.shape[0]) / 10)
dest *= orig.sum() / dest.sum()

vectors = pd.DataFrame({"origins": orig, "destinations": dest}, index=demand.index[:])
```

IPF for the future vectors

Let's balance the future vectors. The output of this step is going to be used later in the traffic assignment for future year.

```
args = {
    "matrix": demand,
```

(continues on next page)

(continued from previous page)

```
    "vectors": vectors,
    "column_field": "destinations",
    "row_field": "origins",
    "nan_as_zero": True,
}

ipf = Ipf(project=project, **args)
ipf.fit()
```

Save the balanced matrix into the project's matrix folder

```
ipf.save_to_project(file_name="demand_ipfd.omx")
```

Impedance

Let's get the base-year assignment skim for car we created before and prepare it for computation

```
imped = proj_matrices.get_matrix(
    "base_year_assignment_skims_car_intrazonals.omx", subset=["final_time_with_
↪intrazonals"]
)
```

If we wanted the main diagonal to not be considered…

```
# np.fill_diagonal(imped["final_time_with_intrazonals"], np.nan)
```

Now we apply the Synthetic Gravity model

```
for function in ["power", "expo"]:
    model = SyntheticGravityModel()
    model.load(join(fldr, f"{function}_model.mod"))

    args = {
        "impedance": imped,
        "vectors": vectors,
        "row_field": "origins",
        "model": model,
        "column_field": "destinations",
        "nan_as_zero": True,
    }

    gravity = GravityApplication(project=project, **args)
    gravity.apply()

    # We get the output matrix and save it to OMX too,
    gravity.save_to_project(file_name=f"demand_{function}_modeled.omx")
```

We inspect the matrix files and verify that the new matrices are indeed there

```
proj_matrices.get_available_matrices_and_metadata()
```

Traffic assignment with Select Link Analysis

We'll perform traffic assignment for the future year. Let's get our future demand matrix, which corresponds to the IPF result we just saved, and see what is the core we ended up getting. It should be `matrix`.

```
demand = proj_matrices.get_matrix("demand_ipfd.omx", subset=["matrix"])
list(demand)
```

The selected matrix core is ready for computation.

The future year assignment is quite similar to the one we did for the base-year.

```
# So, let's create the assignment class
assigclass = TrafficClass(
    name="car",
    mode="c",
    cost="free_flow_time",
    demand=demand,
    skimming=["free_flow_time", "distance"],
    # Every node in Sioux Falls is a centroid, so paths must pass through centroids.
    graph_options={"block_centroid_flows": False},
)


def quadratic(congested_time, link_flows, fftime, capacity, cores, a, b):
    congested_time[:] = fftime * (a * (link_flows / capacity) ** 2 + b * (link_flows /
↪ capacity) + 1)


def quadratic_derivative(delta, link_flows, fftime, capacity, cores, a, b):
    delta[:] = fftime * (2 * a * (link_flows / capacity) + b) / capacity


vdf = VDF(
    "quadratic",
    quadratic,
    {"a": {"fill_NA": 0.15, "bounds": (0, float("inf"))}, "b": {"fill_NA": 0.1}},
    quadratic_derivative,
)
```

Now we select two sets of links to execute select link analysis.

```
select_links = {
    "Leaving node 1": [(1, 1), (2, 1)],
    "Random nodes": [(3, 1), (5, 1)],
}
```

Note

As we are executing the select link analysis on a particular `TrafficClass`, we should set the links we want to analyze. The input is a dictionary with string as keys and a list of tuples as values, so that each entry represents a separate set of selected links to compute.

```
select_link_dict = {"set_name": [(link_id1, direction1), ..., (link_id, direction)]}
```

The string name will name the set of links, and the list of tuples is the list of selected links in the form `(link_id, direction)`, as it occurs in the *Graph*.

Direction can be one of 0, 1, or -1, where 0 denotes bi-directionality.

```
# We call this command on the class we are analyzing with our dictionary of values
assigclass = replace(assigclass, select_link_sets=select_links)

# Configure the VDF, network fields, and assignment algorithm.
assig = TrafficAssignment(
    network=network,
    project=project,
    classes=[assigclass],
    vdf=vdf,
    name_mapping={"a": 0.15, "b": 1.0},
    time_field="free_flow_time",
    capacity_field="capacity",
    algorithm="bfw",
)
assig.execute(max_iterations=500, rgap=0.00001)
```

To save our select link results, all we need to do is provide it with a name. In addition to exporting the select link flows, it also exports the Select Link matrices in OMX format.

```
assig.save_select_link_results("select_link_analysis")
```

Note

Say we just want to save our select link flows, we can call: `assig.save_select_link_flows("just_flows")`

Or if we just want the select link matrices: `assig.save_select_link_matrices("just_matrices")`

Internally, the `save_select_link_results` calls both of these methods at once.

We can export the results to CSV or AequilibraE Data, but let's put it directly into the results database

```
assig.save_results("future_year_assignment")
```

And save the skims

```
assig.save_skims("future_year_assignment_skims", which_ones="all", format="omx")
```

Run convergence study

```
df = assig.report()
x = df.iteration.values
y = df.rgap.values

fig = plt.figure()
ax = fig.add_subplot(111)

plt.plot(x, y, "k--")
```

(continues on next page)

(continued from previous page)

```
plt.yscale("log")
plt.grid(True, which="both")
plt.xlabel("Iterations")
plt.ylabel("Relative Gap")
plt.show()
```

Close the project

```
project.close()
```

CHAPTER

EIGHT

PUBLIC TRANSPORT

Public transport data is a key element of transport planning in general[1]. AequilibraE is capable of importing a General Transit Feed Specification (GTFS) to its public transport database. The GTFS is a standardized data format widely used in public transport planning and operation, and was first proposed during the 2000s', for public transit agencies to describe details from their services, such as schedules, stops, fares, etc[2]. Currently, there are two types of GTFS data:

- GTFS schedule, which contains information on routes, schedules, fares, and other details;
- GTFS realtime, which contains real-time vehicle position, trip updates, and service alerts.

The GTFS protocol is being constantly updated and so are AequilibraE's capabilities of handling these changes. We strongly encourage you to take a look at the documentation provided by Mobility Data.

In this section we also present the transit assignment models, which are mathematical tools that predict how passengers behave and travel in a transit network, given some assumptions and inputs.

Transit assignment models aim to answer questions such as:

- How do transit passengers choose their routes in a complex network of lines and services?
- How can we estimate the distribution of passenger flows and the performance of transit systems?

See also

- ***Public Transport Database***
  Database structure

8.1 Transit assignment graph

In this section, we describe a graph structure for a transit network used for static, link-based, frequency-based assignment. Our focus is the classic algorithm *optimal strategies* by Spiess and Florian (1989)[1].

Let's start by giving a few definitions:

- **transit**: according to Wikipedia, it is a *"system of transport for passengers by group travel systems available for use by the general public unlike private transport, typically managed on a schedule, operated on established routes, and that charge a posted fee for each trip."*
- **transit network**: a set of transit lines and stops, where passengers can board, alight or change vehicles.

[1] Pereira, R.H.M. and Herszenhut, D. (2023) Introduction to urban accessibility: a practical guide with R. Rio de Janeiro, IPEA. Available at: https://repositorio.ipea.gov.br/bitstream/11058/12689/52/Introduction_urban_accessibility_Book.pdf

[2] Mobility Data (2024) GTFS: Making Public Transit Data Universally Accessible. Available at: https://gtfs.org/getting-started/what-is-GTFS/

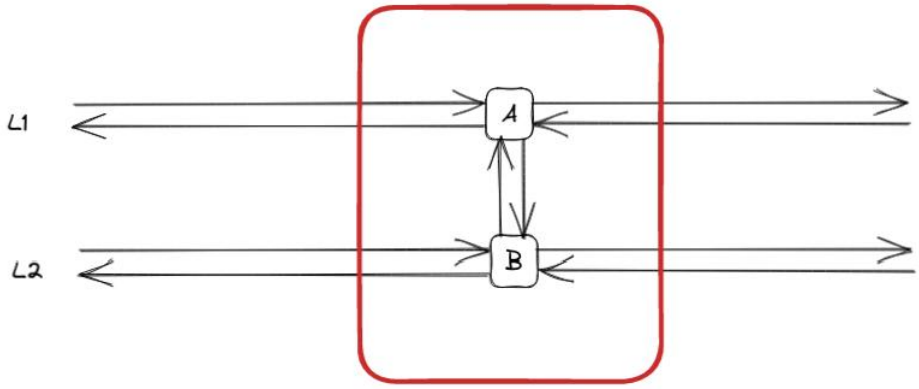
[1] Spiess, H. and Florian, M. (1989) "Optimal strategies: A new assignment model for transit networks". Transportation Research Part B: Methodological, 23(2), 83-102. Available in: https://doi.org/10.1016/0191-2615(89)90034-9

- **assignment**: distribution of the passengers (demand) on the network (supply), knowing that transit users attempt to minimize total travel time, time or distance walking, time waiting, number of transfers, fares, etc…
- **static assignment**: assignment without time evolution. Dynamic properties of the flows, such as congestion, are not well described, unlike with dynamic assignment models.
- **schedule-based approach**: in this approach, distinct vehicle trips are represented by distinct links. We can see the associated network as a time-expanded network, where the third dimension would be time.
- **frequency-based** (or **headway-based**) **approach**: unlike with the schedule-based approach, the schedules are averaged in order to get line frequencies.
- **link-based approach**: in this approach, the assignment algorithm is not evaluating paths, or any aggregated information besides attributes stored by nodes and links. In the present case, each link has an associated cost (travel-time, s) and frequency ($f = 1/s$).

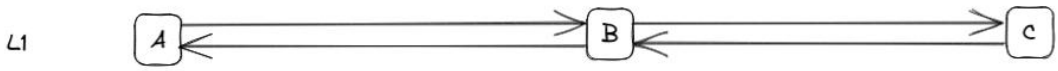
8.1.1 Elements of a transit network

These are the elements required to describe an assignment graph.

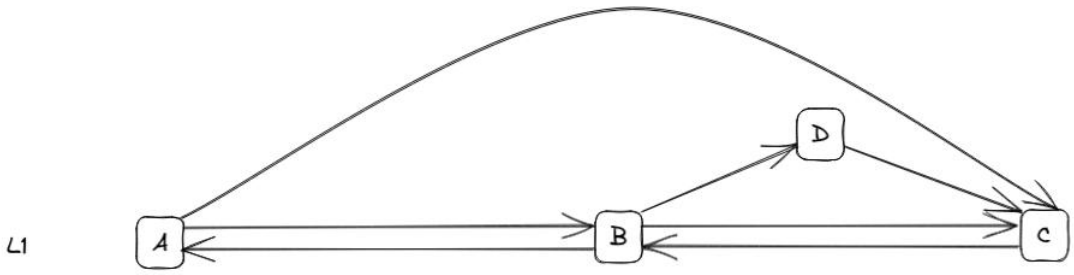
- **Transit stops and stations**: transit stops are points where passenger can board, alight or change vehicles. Also, they can be part of larger stations. In the illustratation below, two distinct stops ('A' and 'B') are highlighted, and they are both affiliated with the same station (depicted in red).



- **Transit lines**: a transit line is a set of services that may use different routes, decomposed into segments.
- **Transit routes**: a route is described by a sequence of stop nodes. We assume here the routes to be directed. For example, we can take a simple case with 3 stops. In this case, the 'L1' line is made of two different routes: 'ABC' and 'CBA'.



A route can present various configurations, such as a partial route at a given moment of the day ('AB'), a route with an additional stop ('ABDC'), a route that does not stop at a given stop ('AC').



Lines can also be decomposed into multiple sub-lines, each representing distinct routes. For the given example, we may have several sub-lines under the same commercial line (L1).

Line ID	Commercial Name	Stop Sequence	Headway (s)
L1_a1	L1	ABC	600
L1_a2	L1	ABDC	3,600
L1_a3	L1	AB	3,600
L1_a4	L1	AC	3,600
L1_b1	L1	CBA	600

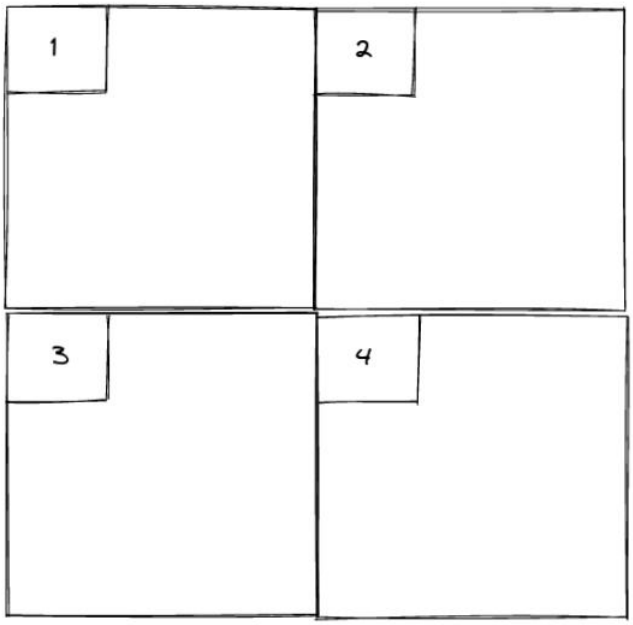
Associated with each sub-line, the headway corresponds to the mean time range between consecutive vehicles — the inverse of the line frequency used as a link attribute in the assignment algorithm.

- **Line segments**: a line segment represents a portion of a transit line between two consecutive stops. Using the example line 'L1_a1', we derive two distinct line segments:

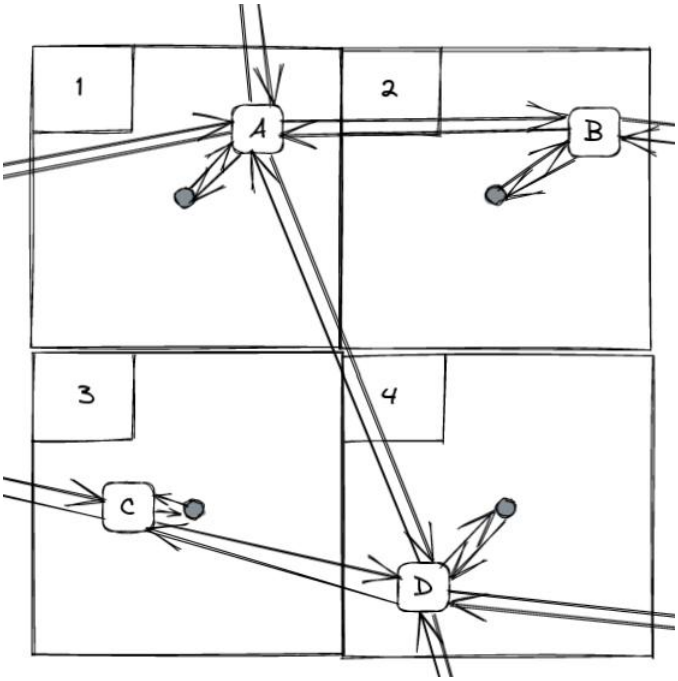
Line ID	Segment Index	Origin Stop	Destination Stop	Travel Time
L1_a1	1	A	B	300
L1_a1	2	B	C	600

Note that a travel time is included for each line segment, serving as another link attribute used by the assignment algorithm.

- **Transit Assignment Zones**: transit assignment zones correspond to the partition of the network area. The illustration below presents 4 non-overlapping zones, whose demand is expressed as a number of trips from each zone to every other zone, forming a 4 x 4 Origin-Destination (OD) matrix.



- **Connectors**: connectors are special network nodes that facilitate the connection between supply and demand.

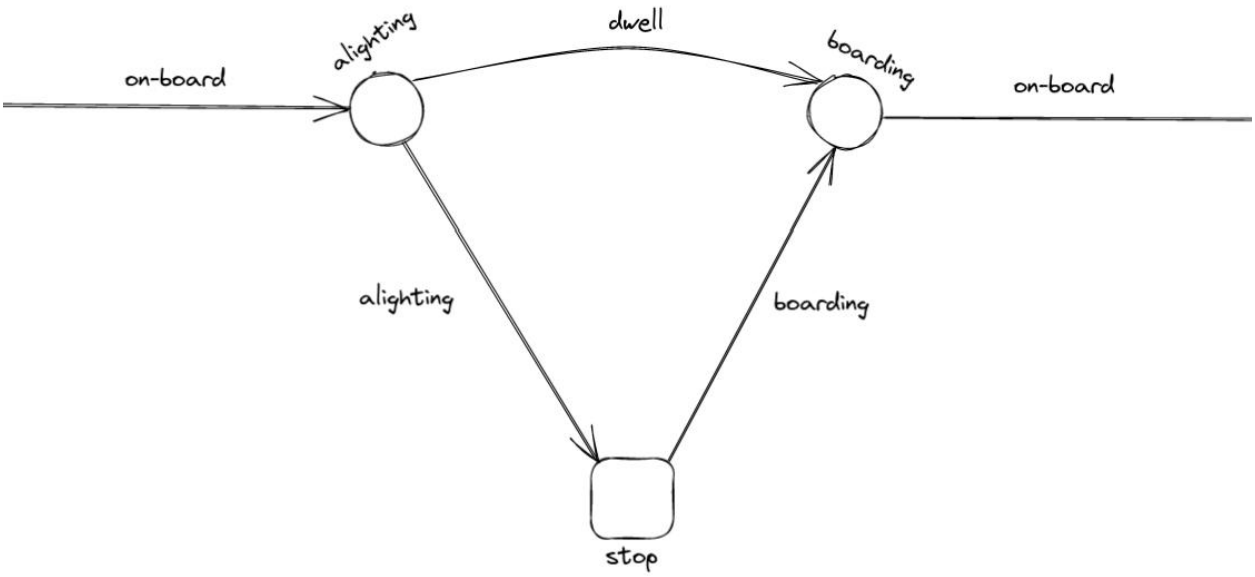


8.1.2 The assignment graph

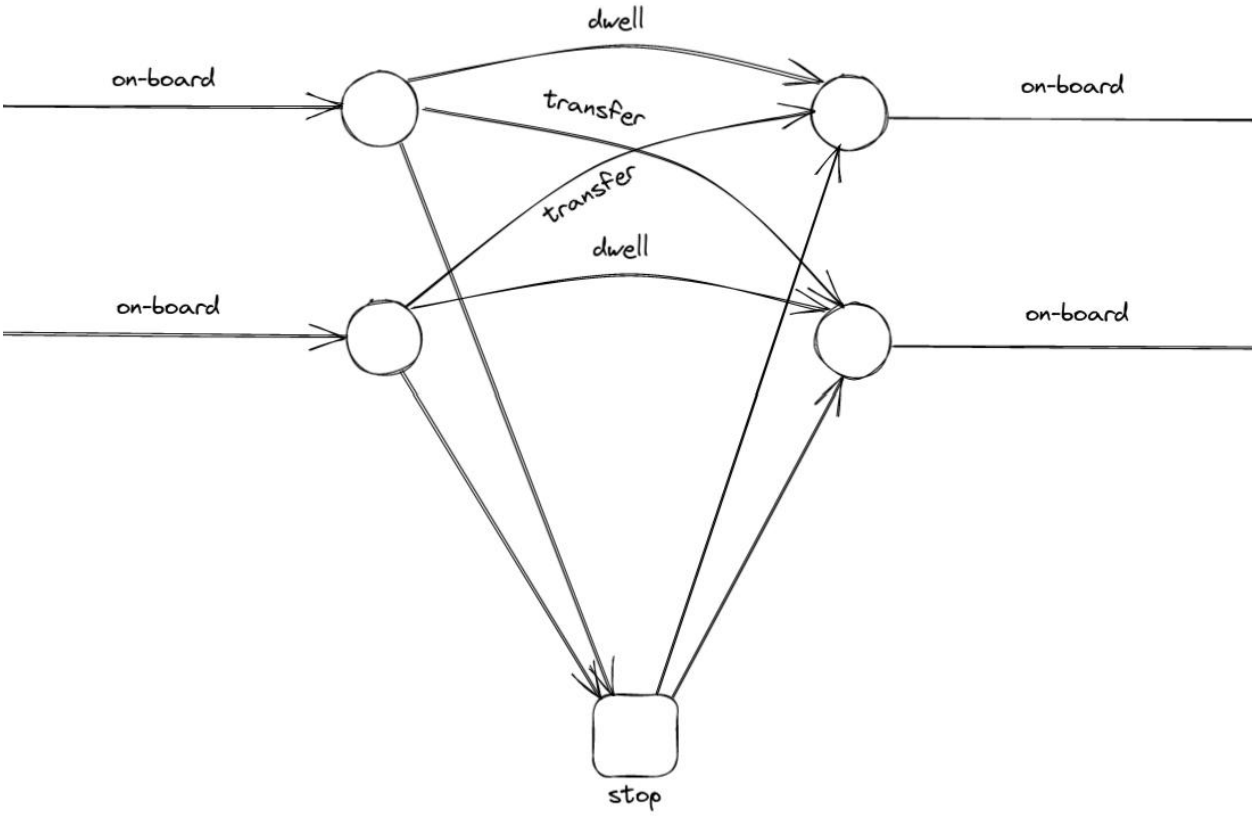
The transit network is used to generate a graph with specific nodes and links used to model the transit process. Various link types and node categories play crucial roles in this representation.

Link types	Node types
On-board	Stop
Boarding	Boarding
Alighting	Alighting
Dwell	OD
Transfer	Walking
Connector	
Walking	

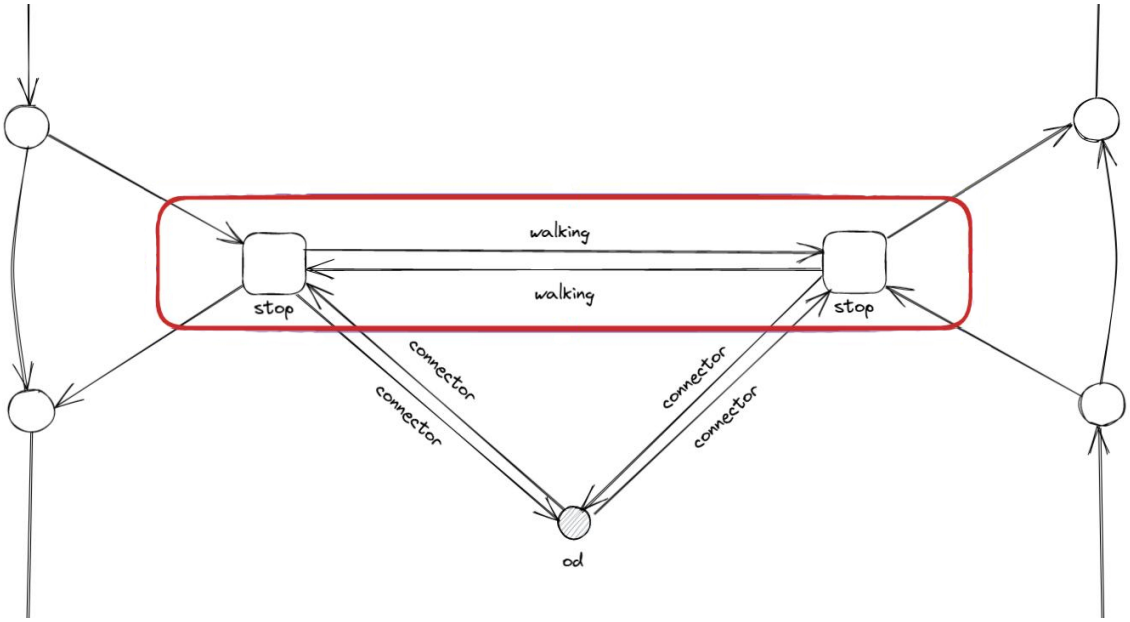
To illustrate, consider the anatomy of a simple stop (figure below). Waiting links encompass boarding and transfer links. Each line segment is associated with a boarding, an on-board and an alighting link.



Transfer links enable to compute the passenger flow count between line couples at the same stop. These links can be extended between all lines of a station if an increase in the number of links is viable.



Walking links connect *stop* nodes within a station, while *connector* links connect the zone centroids (OD nodes) to *stop* nodes. Connectors that connect OD to *stop* nodes allow passengers to access the network, while connectors in the opposite direction allow them to egress. Walking nodes/links may also be used to connect stops from distant stations.



The table below summarizes link characteristics and attributes based on link types:

Link Type	From node type	To node type	Cost	Frequency
on-board	boarding	alighting	travel time	∞
boarding	stop	boarding	constant	line frequency
alighting	alighting	stop	constant	∞
dwell	alighting	boarding	constant	∞
transfer	alighting	boarding	constant + travel time	destination line frequency
connector	OD or stop	OD or stop	travel time	∞
walking	stop or walking	stop or walking	travel time	∞

The travel time is specific to each line segment or walking time. For example, there can be 10 minutes connection between stops in a large transit station. Constant boarding and alighting times are applied uniformly across the network, and dwell links have constant cost equal to the sum of the alighting and boarding constants.

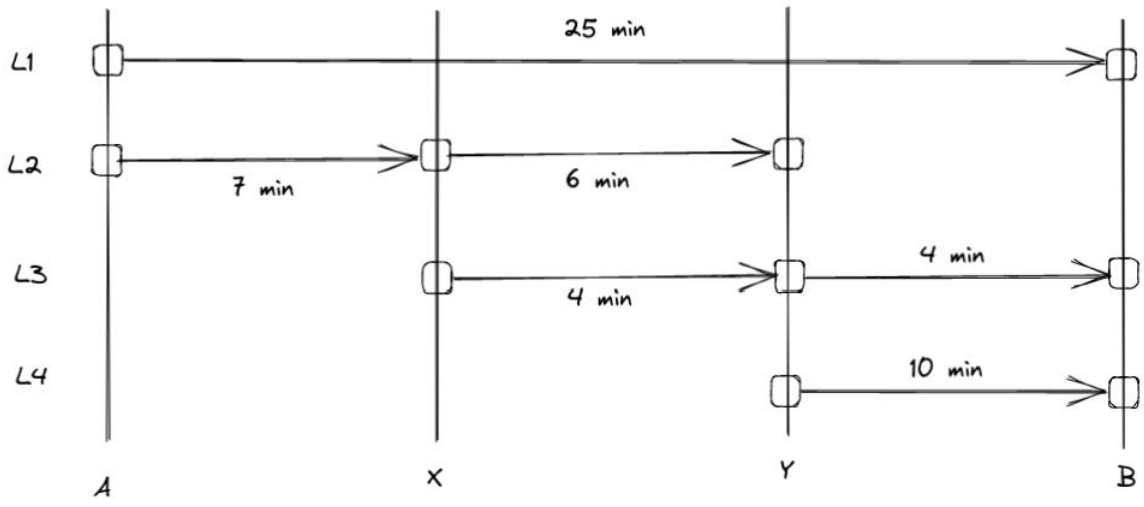
Additional attributes can be introduced for specific link types, such as:

- `line_id`: for on-board, boarding, alighting and dwell links.
- `line_seg_idx`: the line segment index for boarding, on-board and alighting links.
- `stop_id`: for alighting, dwell and boarding links. This can also apply to transfer links for inner stop transfers.
- `o_line_id`: origin line ID for transfer links.
- `d_line_id`: destination line ID for transfer links.

Assignment graph example - Based on Spiess and Florian (1989)

This illustrative example is taken from Spiess and Florian (1989)[Page 213, 1].

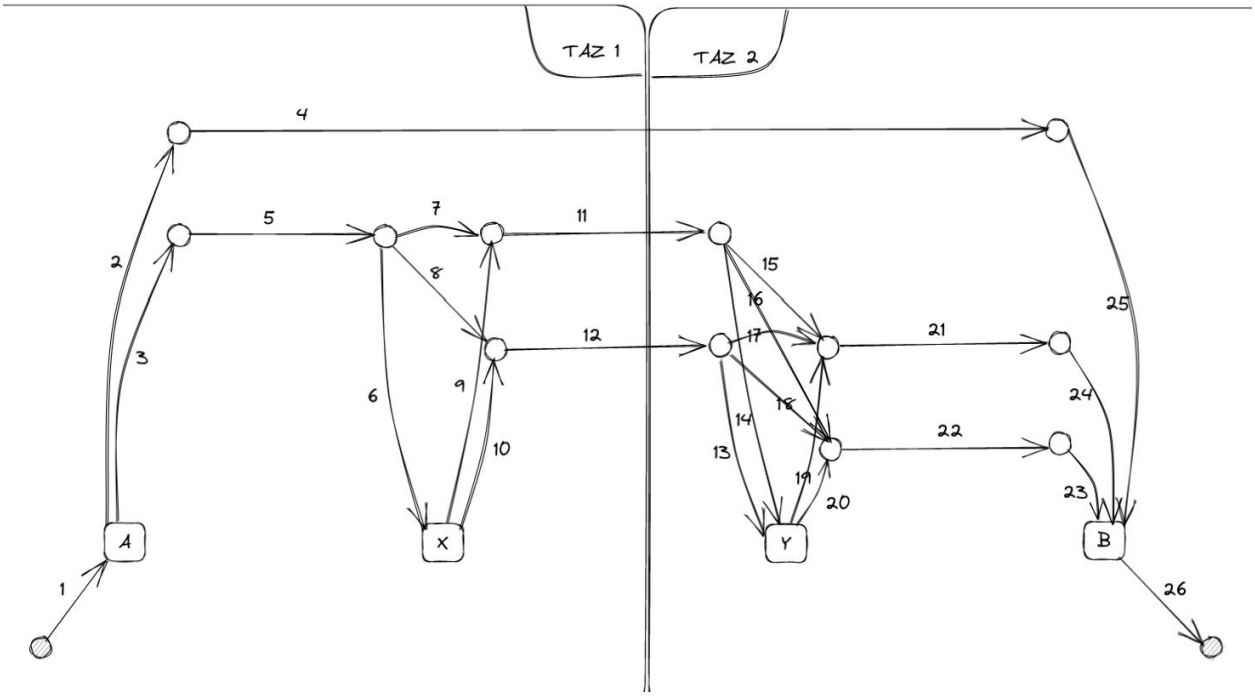
The following figure presents the travel times for each line.



We have the following four distinct line characteristics:

Line ID	Route	Headway (min)	Frequency (1/s)
L1	AB	12	0.001388889
L2	AXY	12	0.001388889
L3	XYB	30	0.000555556
L4	YB	6	0.002777778

Passengers aim to travel from A to B, prompting the division of the network area into two distinct zones: TAZ 1 and TAZ 2. The assignment graph associated with this network encompasses 26 links:



Here is a table listing all links:

Link ID	Link Type	Line ID	Cost	Frequency
1	connector		0	∞
2	boarding	L1	0	0.001388889
3	boarding	L2	0	0.001388889
4	on-board	L1	1500	∞
5	on-board	L2	420	∞
6	alighting	L2	0	∞
7	dwell	L2	0	∞
8	transfer		0	0.000555556
9	boarding	L2	0	0.001388889
10	boarding	L3	0	0.000555556
11	on-board	L2	360	∞
12	on-board	L3	240	∞
13	alighting	L3	0	∞
14	alighting	L2	0	∞
15	transfer	L3	0	0.000555556
16	transfer		0	0.002777778
17	dwell	L3	0	∞
18	transfer		0	0.002777778
19	boarding	L3	0	0.000555556
20	boarding	L4	0	0.002777778
21	on-board	L3	240	∞
22	on-board	L4	600	∞
23	alighting	L4	0	∞
24	alighting	L3	0	∞
25	alighting	L1	0	∞
26	connector		0	∞

8.1.3 Transit graph specificities in AequilibraE

The graph creation process in AequilibraE incorporates several edge types to capture the nuances of transit networks. Notable distinctions include:

- Connectors
  - access connectors: directed from od nodes to the network
  - egress connectors: directed from the network to the od nodes
- Transfer edges
  - inner transfer: connect lines within the same stop
  - outer transfer: connect lines between distinct stops within the same station
- Origin and destination nodes
  - origin nodes: represent the starting point of passenger trips
  - destination nodes: represent the end point of passenger trips

Users can customize these features using boolean parameters:

- `with_walking_edges`: create walking edges between the stops of a station
- `with_inner_stop_transfers`: create transfer edges between lines of a stop
- `with_outer_stop_transfers`: create transfer edges between lines of different stops of a station

- `blocking_centroid_flows`: duplicate OD nodes into unconnected origin and destination nodes to block centroid flows. Flows start from an origin node and end at a destination node. It is not possible to use an egress connector followed by an access connector in the middle of a trip.

Note that during the assignment, if passengers have the choice between a transfer edge or a walking edge for a line change, they will always be assigned to the transfer edge. This leads to these possible edge types:

- on-board
- boarding
- alighting
- dwell
- access_connector
- egress_connector
- inner_transfer
- outer_transfer
- walking

8.1.4 References

8.2 Hyperpath routing

Hyperpath routing is one of the basic concepts in transit assignment models, and it is a way of representing the set of optimal routes that a passenger can take from an origin to a destination, based on some criterion such as travel time or generalized cost. A hyperpath is a collection of links that form a subgraph of the transit network. Each link in the hyperpath also has a probability of being used by the passenger, which reflects the attractiveness and uncertainty of the route choice. The shortest hyperpath is optimal regarding the combination of paths weighted by the probability of being used.

Hyperpath routing can be applied to different types of transit assignment models, but here we will focus on frequency-based models. Frequency-based models assume that passengers do not have reliable information about the service schedules and arrival times, and they choose their routes based on the expected travel time or cost. This type of model is suitable for transit systems with rather frequent services.

To illustrate how hyperpath routing works in frequency-based models, we will use the algorithm by Spiess and Florian[1] implemented in AequilibraE.

For example purposes, we will use a simple grid network as an Python example to demonstrate how a hyperpath depends on link frequency for a given origin-destination pair. Note that it can be extended to other contexts such as risk-averse vehicle navigation[2].

8.2.1 Bell's network

We start by defining the directed graph $\mathcal{G} = (V, E)$, where V and E are the graph vertices and edges. The hyperpath generating algorithm requires 2 attributes for each edge $a \in V$:

- edge travel time: $u_a \geq 0$
- edge frequency: $f_a \geq 0$

[1] Spiess, H. and Florian, M. (1989) "Optimal strategies: A new assignment model for transit networks". Transportation Research Part B: Methodological, 23(2), 83-102. Available in: https://doi.org/10.1016/0191-2615(89)90034-9

[2] Ma, J., Fukuda, D. and Schmöcker, J.D. (2012) "Faster hyperpath generating algorithms for vehicle navigation", Transportmetrica A: Transport Science, 9(10), 925–948. Available in: https://doi.org/10.1080/18128602.2012.719165

The edge frequency is inversely related to the exposure to delay. For example, in a transit network, a boarding edge has a frequency that is the inverse of the headway (or half the headway, depending on the model assumptions). A walking edge has no exposure to delay, so its frequency is assumed to be infinite.

Bell's network is a synthetic network: it is a n-by-n grid bi-directional network[Page 222, 23]. The edge travel time is taken as random number following a uniform distribution:

$$u_a \sim \mathbf{U}[0, 1)$$

To demonstrate how the hyperpath depends on the exposure to delay, we will use a positive constant (α) and a base delay (d_a) for each edge that follows a uniform distribution:

$$d_a \sim \mathbf{U}[0, 1)$$

The constant $\alpha \geq 0$ allows us to adjust the edge frequency as follows:

$$f_a = \begin{cases} 1/\left(\alpha\ d_a\right) & \text{if } \alpha\ d_a \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

Notice that a smaller α value implies higher edge frequencies, and vice versa.

8.2.2 Hyperpath computation

Let's create a function that:

- creates the network,
- computes the edge frequency given an input value for α,
- computes the shortest hyperpath,
- and plots the network and hyperpath.

We start with $\alpha = 0$. This implies that there is no delay over all the network. The resulting hyperpath corresponds to the same shortest path that Dijkstra's algorithm would have computed. You can call NetworkX's method `nx.dijkstra_path` to compute the shortest path.

To introduce some delay in the network, we can increase the value of α. We notice that the shortest path is no longer unique and multiple routes are suggested. The link usage probability is reflected by the line width. The majority of the flow still follows the shortest path, but some of it is distributed among different alternative paths. This becomes more apparent as we further increase α.

The code below allows you to reproduce the same experiment that resulted in the previous figures.

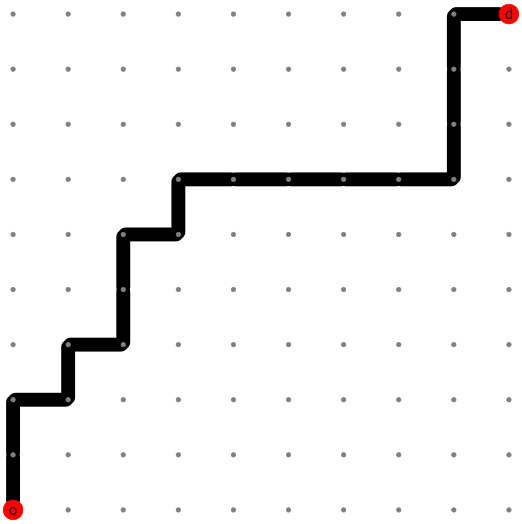
Listing 0: Hyperpath computation

```
# Let's import some packages
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd

from aequilibrae.matrix import from_dict
from aequilibrae.routing.cython.demand import assignment_demand
from aequilibrae.routing.cython.public_transport import HyperpathGenerating
from numba import jit
```

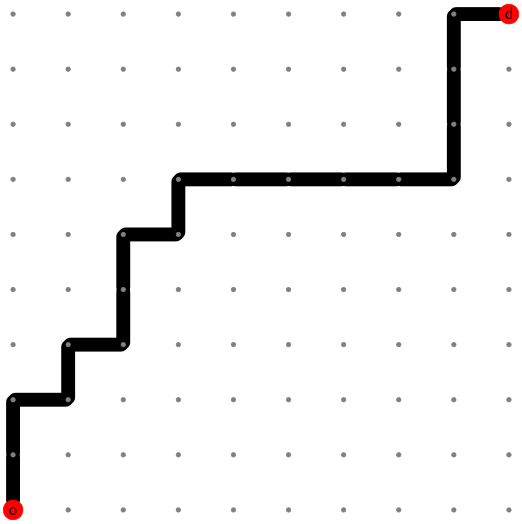
(continues on next page)

[3] Bell, M.G.H. (2009) "Hyperstar: A multi-path Astar algorithm for risk averse vehicle navigation", Transportation Research Part B: Methodological, 43(1), 97-107. Available in: https://doi.org/10.1016/j.trb.2008.05.010.

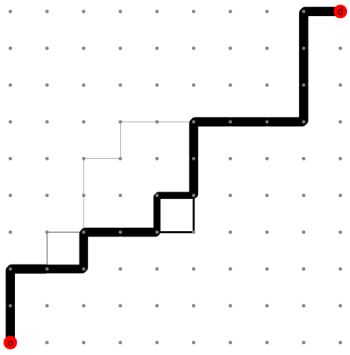
Shortest hyperpath - Bell's network α=0.0



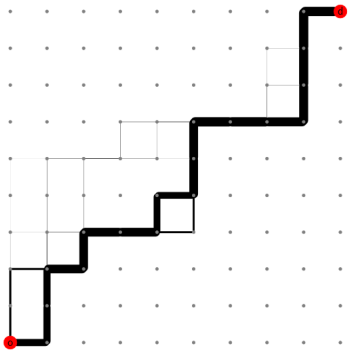
Shortest path - Bell's network



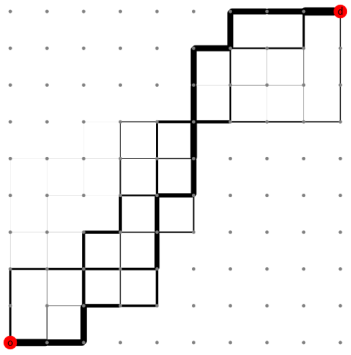
Shortest hyperpath - Bell's network α=0.5



Shortest hyperpath - Bell's network α=1.0



Shortest hyperpath - Bell's network α=100.0



(continued from previous page)

```
RANDOM_SEED = 124  # random seed
FIGURE_SIZE = (6, 6)  # figure size


def create_vertices(n):
    x = np.linspace(0, 1, n)
    y = np.linspace(0, 1, n)
    xv, yv = np.meshgrid(x, y, indexing="xy")
    vertices = pd.DataFrame()
    vertices["x"] = xv.ravel()
    vertices["y"] = yv.ravel()
    return vertices

@jit
def create_edges_numba(n):
    m = 2 * n * (n - 1)
    tail = np.zeros(m, dtype=np.uint32)
    head = np.zeros(m, dtype=np.uint32)
    k = 0
    for i in range(n - 1):
        for j in range(n):
            tail[k] = i + j * n
            head[k] = i + 1 + j * n
            k += 1
            tail[k] = j + i * n
            head[k] = j + (i + 1) * n
            k += 1
    return tail, head

def create_edges(n, seed=124):
    tail, head = create_edges_numba(n)
    edges = pd.DataFrame()
    edges["tail"] = tail
    edges["head"] = head
    m = len(edges)
    rng = np.random.default_rng(seed=seed)
    edges["trav_time"] = rng.uniform(0.0, 1.0, m)
    edges["delay_base"] = rng.uniform(0.0, 1.0, m)
    return edges

def generate_hyperpath(n, alpha):
    edges = create_edges(n, seed=RANDOM_SEED)
    delay_base = edges.delay_base.values
    indices = np.where(delay_base == 0.0)
    delay_base[indices] = 1.0
    freq_base = 1.0 / delay_base
    freq_base[indices] = np.inf

    edges["freq_base"] = freq_base
    if alpha == 0.0:
        edges["freq"] = np.inf
```

(continues on next page)

(continued from previous page)

```
    else:
        edges["freq"] = edges.freq_base / alpha

    # Spiess & Florian
    centroids = np.array([0, n * n - 1], dtype=np.int64)
    sf = HyperpathGenerating(
        edges,
        tail="tail",
        head="head",
        trav_time="trav_time",
        freq="freq",
        o_vert_ids=centroids,
        d_vert_ids=centroids,
        nodes_to_indices=np.arange(n * n, dtype=np.int64),
    )
    demand = from_dict({"passengers": np.array([[0.0, 1.0], [0.0, 0.0]])},␣
↪index=centroids)
    sf.assign(assignment_demand(demand, centroids, transpose=True), threads=1)

    return sf

def plot_shortest_hyperpath(n=10, alpha=10.0, is_dijkstra=False, figsize=FIGURE_SIZE,
↪ title=""):
    vertices = create_vertices(n)
    n_vertices = n * n
    sf = generate_hyperpath(n, alpha)

    attr = "trav_time" if is_dijkstra else "volume"

    # NetworkX
    G = nx.from_pandas_edgelist(
        sf._edges,
        source="tail",
        target="head",
        edge_attr=attr,
        create_using=nx.DiGraph,
    )

    if is_dijkstra:
        nodes = nx.dijkstra_path(G, 0, n*n-1, weight='trav_time')
        edges = list(nx.utils.pairwise(nodes))
        widths = 1e2 * np.array([1 if (u,v) in edges else 0 for u, v in G.edges()]) /
↪ n
    else:
        widths = 1e2 * np.array([G[u][v]["volume"] for u, v in G.edges()]) / n
    pos = vertices[["x", "y"]].values

    _ = plt.figure(figsize=figsize)
    node_colors = n_vertices * ["gray"]
    node_colors[0] = "r"
    node_colors[-1] = "r"
    ns = 100 / n
```

(continues on next page)

(continued from previous page)

```
    node_size = n_vertices * [ns]
    node_size[0] = 20 * ns
    node_size[-1] = 20 * ns
    labeldict = {}
    labeldict[0] = "O"
    labeldict[n * n - 1] = "D"
    nx.draw(
        G,
        pos=pos,
        width=widths,
        node_size=node_size,
        node_color=node_colors,
        arrowstyle="-",
        labels=labeldict,
        with_labels=True,
    )
    ax = plt.gca()
    _ = ax.set_title(title, color="k")

plot_shortest_hyperpath(n=10, alpha=0.0, title="Shortest hyperpath - Bell's Network
↪$\\alpha$=0.0")
plot_shortest_hyperpath(n=10, alpha=0.0, is_dijkstra=True, title="Shortest path -␣
↪Dijkstra's Algorithm")
plot_shortest_hyperpath(n=10, alpha=0.5, title="Shortest hyperpath - Bell's Network
↪$\\alpha$=0.5")
plot_shortest_hyperpath(n=10, alpha=1.0, title="Shortest hyperpath - Bell's Network
↪$\\alpha$=1.0")
plot_shortest_hyperpath(n=10, alpha=100.0, title="Shortest hyperpath - Bell's␣
↪Network $\\alpha$=100.0")
```

8.2.3 References

8.3 Transit skimming

Transit skimming in AequilibraE is incredibly flexible, but more sophisticated use requires a good understanding of the structure of the *Transit assignment graph*, so we recommend reading that section first.

For typical use cases, the method *set_skimming_fields* accepts a set of predefined fields which are defined based on the auto-generated link types. These include:

- discrete: *'boardings'*, *'alightings'*, *'inner_transfers'*, *'outer_transfers'*, and *'transfers'*.
- continuous: *'trav_time'*, *'on_board_trav_time'*, *'dwelling_time'*, *'egress_trav_time'*, *'access_trav_time'*, *'walking_trav_time'*, *'transfer_time'*, *'in_vehicle_trav_time'*, and *'waiting_time'*.

Each transit class accepts either an `AequilibraeMatrix` or an `AequilibraeSparseMatrix` containing exactly one demand matrix, with finite, non-negative float64 values. Use `matrix.subset(["pt"])` to select a matrix before constructing the class if needed. Skims are always dense `AequilibraeMatrix` objects using the demand's zone index.

```
>>> import numpy as np
>>> from aequilibrae.matrix import AequilibraeMatrix
>>> from aequilibrae.transit import TransitAssignment, TransitClass
>>> from aequilibrae.examples import create_example
```

(continues on next page)

(continued from previous page)

```
>>> project = create_example(project_path, "coquimbo")
>>> data = project.transit

>>> graph = data.create_graph(
...     with_outer_stop_transfers=False,
...     with_walking_edges=False,
...     blocking_centroid_flows=False,
...     connector_method="overlapping_regions",
... )
>>> network = project.road.build_network()

>>> graph.create_line_geometry(method="direct", graph="c")

>>> transit_graph = graph.to_transit_graph()

>>> # We mock a demand matrix
>>> num_zones = len(transit_graph.centroids)

>>> mat = AequilibraeMatrix(index=transit_graph.centroids, index_name="zones")
>>> mat["pt"] = np.full((num_zones, num_zones), 1.0)

>>> # We can now execute the assignment, and we will use some of the default skimming␣
↪fields
>>> skim_cols = ["trav_time", "boardings", "in_vehicle_trav_time", "egress_trav_time",
↪ "access_trav_time"]

>>> assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)

>>> assig = TransitAssignment()
>>> assig.add_class(assigclass)
>>> assig.set_time_field("trav_time")
>>> assig.set_frequency_field("freq")

>>> assig.set_skimming_fields(skim_cols) # Skimming must be set after a transit␣
↪assignment class is added

>>> assig.set_algorithm("os")

>>> assig.execute()

>>> project.close()
```

More sophisticated skimming is also possible, such as skimming related to specific routes and/or modes. As it is the case with traffic graphs, this type of exercise consists basically of defining fields in the graph that represent the desired skimming metrics.

One example is skimming travel time in rail only.

```
>>> from aequilibrae.transit import TransitAssignment, TransitClass

>>> project = create_example(f"{project_path}v2", "coquimbo")
>>> data = project.transit
```

(continues on next page)

(continued from previous page)

```
>>> graph = data.create_graph(
...     with_outer_stop_transfers=False,
...     with_walking_edges=False,
...     blocking_centroid_flows=False,
...     connector_method="overlapping_regions",
... )
>>> network = project.road.build_network()

>>> graph.create_line_geometry(method="direct", graph="c")

>>> transit_graph = graph.to_transit_graph()

>>> # We now define a new field in the graph that will be used for skimming
>>> transit_graph.graph["rail_trav_time"] = np.where(
...      transit_graph.graph["link_type"].isin(["on-board", "dwell"]), 0, transit_
↪graph.graph["trav_time"]
... )

>>> all_routes = transit.get_table("routes")
>>> rail_ids = all_routes.query("route_type in [1, 2]").route_id.to_numpy()

# Assign zero travel time to all non-rail links
>>> transit_graph.graph.loc[~transit_graph.graph.line_id.isin(rail_ids),"rail_trav_
↪time"] =0

>>> # We mock a demand matrix
>>> num_zones = len(transit_graph.centroids)

>>> mat = AequilibraeMatrix(index=transit_graph.centroids, index_name="zones")
>>> mat["pt"] = np.full((num_zones, num_zones), 1.0)

>>> # We can now execute the assignment, and we will use some of the default skimming␣
↪fields
>>> skim_cols = ["trav_time", "boardings", "in_vehicle_trav_time", "egress_trav_time",
↪ "access_trav_time"]

>>> assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)

>>> assig = TransitAssignment()
>>> assig.add_class(assigclass)
>>> assig.set_time_field("trav_time")
>>> assig.set_frequency_field("freq")

>>> # Skimming must be set after a transit assignment class is added
>>> assig.set_skimming_fields(["rail_trav_time"])

>>> assig.set_algorithm("os")

>>> assig.execute()

>>> project.close()
```

8.4 Examples

8.4.1 Public Transport

Import GTFS

In this example, we import a GTFS feed to our model and perform map matching.

We use data from Coquimbo, a city in La Serena Metropolitan Area in Chile.

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.transit*
- project.transit.new_gtfs_builder()

```
# Imports
from uuid import uuid4
from os import remove
from os.path import join
from tempfile import gettempdir

import folium
import geopandas as gpd
import pandas as pd

from aequilibrae import Project
from aequilibrae.examples import create_example
```

```
# Let's create an empty project on an arbitrary folder.
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

As the Coquimbo example already has a complete GTFS model, we shall remove its public transport database for the sake of this example. Close its open connection before removing the database, then reopen the project and create a fresh transit database.

```
project.close()
remove(join(fldr, "public_transport.sqlite"))
project = Project(fldr)
```

Let's import the GTFS feed.

```
dest_path = join(fldr, "gtfs_coquimbo.zip")
```

Now we create our Transit object and import the GTFS feed into our model. This will automatically create a new public transport database.

```
data = project.scenario.create_transit_database()

transit = data.new_gtfs_builder(agency="Lisanco", file_path=dest_path)
```

To load the data, we must choose one date. We're going to continue with 2016-04-13 but feel free to experiment with any other available dates. Transit class has a function allowing you to check dates for the GTFS feed. It should take approximately 2 minutes to load the data.

```
transit.load_date("2016-04-13")

# Now we execute the map matching to find the real paths.
# Depending on the GTFS size, this process can be really time-consuming.

# transit.set_allow_map_match(True)
# transit.map_match()

# Finally, we save our GTFS into our model.
transit.save_to_disk()
```

Now we will plot one of the route's patterns we just imported

```
with project.transit_connection as conn:
    patterns = pd.read_sql("SELECT pattern_id, ST_AsText(geometry) geom FROM routes;",
↪ con=conn)
    stops = pd.read_sql("""SELECT stop_id, ST_X(geometry) X, ST_Y(geometry) Y FROM␣
↪stops""", con=conn)
```

We turn the patterns and stops DataFrames into GeoDataFrames so we can plot them more easily.

```
patterns = gpd.GeoDataFrame(patterns, geometry=gpd.GeoSeries.from_wkt(patterns["geom
↪"]), crs=4326)
stops = gpd.GeoDataFrame(stops, geometry=gpd.GeoSeries.from_xy(stops["X"], stops["Y
↪"]), crs=4326)
```

And plot out data!

```
map = patterns.explore(color=["#146DB3", "#EB9719"], style_kwds={"weight": 4}, name=
↪"links")
map = stops.explore(m=map, color="black", style_kwds={"radius": 2, "fillOpacity": 1.0}
↪, name="stops")

folium.LayerControl().add_to(map)
map
```

```
project.close()
```

Public transport assignment with Optimal Strategies

In this example, perform a Spiess & Florian assignment. Click here to check out the paper.

We use data from Coquimbo, a city in La Serena Metropolitan Area in Chile.

References

- *Public Transport*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.transit*
- project.transit.create_graph()
- *aequilibrae.transit.TransitClass*
- *aequilibrae.transit.TransitAssignment*
- *aequilibrae.matrix.AequilibraeMatrix*

```
# Imports for example construction
from uuid import uuid4
from os.path import join
from tempfile import gettempdir

from aequilibrae.examples import create_example
```

Let's create an empty project on an arbitrary folder.

```
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

Let's access our project's transit gateway.

```
data = project.transit
```

Graph building

Let's build the transit network. We'll disable `outer_stop_transfers` and `walking_edges` because Coquimbo doesn't have any parent stations.

For the OD connections we'll use the `overlapping_regions` method and create some accurate line geometry later. Creating the graph should only take a moment. By default zoning information is pulled from the project network. If you have your own zoning information add it using `graph.add_zones(zones)` then `graph.create_graph()`.

```
graph = data.create_graph(with_outer_stop_transfers=False, with_walking_edges=False,␣
↪blocking_centroid_flows=False, connector_method="overlapping_regions")

# We drop geometry here for the sake of display.
graph.vertices.drop(columns="geometry")
```

```
graph.edges
```

The graphs are also stored in the `project.transit.graphs` dictionary. They are keyed by the 'period_id' they were created for. A graph for a different 'period_id' can be created by providing `period_id=` in the `project.transit.create_graph` call. You can view previously created periods with the `Periods` object.

```
periods = project.tables.periods
periods.data
```

Connector project matching

Now we'll create the line strings for the access connectors, this step is optional but provides more accurate distance estimations and better looking geometry.

Because Coquimbo doesn't have many walking edges we'll match onto the `"c"` graph.

```
graph.create_line_geometry(method="connector project match", graph="c")
```

Saving and reloading

Lets save all graphs to the 'public_transport.sqlite' database.

```
data.save_graphs()
```

We can reload the saved graphs with `data.load`. This will create new `TransitGraphBuilder`'s based on the 'period_id' of the saved graphs. The graph configuration is stored in the 'transit_graph_config' table in 'project_database.sqlite' as serialised JSON.

```
data.load()
```

We can also remove the previously saved graphs.

```
# data.remove_graphs()
```

Links and nodes are stored in a similar manner to the 'project_database.sqlite' database.

Reading back into AequilibraE

You can retrieve a reloaded graph via its 'period_id'.

```
period_id = periods.default_period.period_id
```

```
graph_db = data.graphs[period_id]
graph_db.vertices.drop(columns="geometry")
```

```
graph_db.edges
```

Converting to a AequilibraE graph object

To perform an assignment we need to convert the graph builder into a graph.

```
transit_graph = graph.to_transit_graph()
```

Mock demand matrix

We'll create a mock demand matrix with demand 1 for every zone. We'll also need to convert from `zone_id`'s to `node_id`'s.

```
import numpy as np
from aequilibrae.matrix import AequilibraeMatrix
```

```
zones_in_the_model = len(transit_graph.centroids)

mat = AequilibraeMatrix(index=transit_graph.centroids, index_name="zones")
mat["pt"] = np.full((zones_in_the_model, zones_in_the_model), 1.0)
```

Hyperpath generation/assignment

We'll create a `TransitAssignment` object as well as a `TransitClass`

```
from aequilibrae.transit import TransitAssignment, TransitClass
```

Create the assignment class

```
assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)

assig = TransitAssignment(project=project)

assig.add_class(assigclass)

# We need to tell AequilbraE where to find the appropriate fields we want to use,
# as well as the assignment algorithm to use.
assig.set_time_field("trav_time")
assig.set_frequency_field("freq")

assig.set_algorithm("os")
```

Each transit class requires exactly one demand matrix. Use mat.subset(["pt"]) if the input contains more.

Let's perform the assignment for the transit classes added

```
assig.execute()
```

View the results

```
assig.results()
```

Saving results

We'll be saving the results to another sqlite db called 'results_database.sqlite'. The 'results' table with 'project_database.sqlite' contains some metadata about each table in 'results_database.sqlite'.

```
assig.save_results(table_name='hyperpath example')
```

Wrapping up

```
project.close()
```

Public transport assignment with skimming

In this example, we build on the transit assignment example and add skimming to it.

We use data from Coquimbo, a city in La Serena Metropolitan Area in Chile.

References

WE HIGHLY RECOMMEND YOU READ THE DOCUMENTATION ON SKIMMING BEFORE PROCEEDING

- *Public Transport*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.Project.transit*
- project.transit.create_graph()
- *aequilibrae.transit.TransitClass*
- *aequilibrae.transit.TransitAssignment*
- *aequilibrae.matrix.AequilibraeMatrix*

Imports for example construction

```
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import numpy as np

from aequilibrae.matrix import AequilibraeMatrix
from aequilibrae.transit import TransitAssignment, TransitClass
from aequilibrae.examples import create_example
```

```
# Let's create an empty project on an arbitrary folder.
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

Let's access our project's transit gateway.

```
data = project.transit
```

Graph building

Let's build the transit network. We'll disable `outer_stop_transfers` and `walking_edges` because Coquimbo doesn't have any parent stations.

For the OD connections we'll use the `overlapping_regions` method and create some accurate line geometry later. Creating the graph should only take a moment. By default zoning information is pulled from the project network. If you have your own zoning information add it using `graph.add_zones(zones)` then `graph.create_graph()`.

```
graph = data.create_graph(
    with_outer_stop_transfers=False,
    with_walking_edges=False,
```

(continues on next page)

(continued from previous page)

```
    blocking_centroid_flows=False,
    connector_method="overlapping_regions"
)
```

Connector project matching

```
graph.create_line_geometry(method="connector project match", graph="c")
data.save_graphs()
data.load()

# Reading back into AequilibraE
graph_db = data.graphs[project.tables.periods.default_period.period_id]
graph_db.vertices.drop(columns="geometry")

# To perform an assignment we need to convert the graph builder into a graph.
transit_graph = graph_db.to_transit_graph()
transit_graph.set_skimming_fields(["trav_time", "boardings", "freq"])
```

```
# Mock demand matrix
zones = len(transit_graph.centroids)
mat = AequilibraeMatrix(index=transit_graph.centroids, index_name="zones")
mat["pt"] = np.full((zones, zones), 1.0)
```

Hyperpath generation/assignment

We'll create a `TransitAssignment` object as well as a `TransitClass`.

```
# Create the assignment class
assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)

assig = TransitAssignment(project=project)

assig.add_class(assigclass)

# Set assignment
assig.set_time_field("trav_time")
assig.set_frequency_field("freq")
assig.set_algorithm("os")

# Perform the assignment for the transit classes added
assig.execute()

# We can use the get_skim_results() method to retrieve the skims
assig.get_skim_results()["pt"]["boardings"].sum()
```

Saving results

We'll save the assignment's link-volume results to results_database.sqlite. The skim matrices returned above remain in memory; this call does not save them.

```
assig.save_results(table_name='hyperpath example')
```

Wrapping up

```
project.close()
```

8.5 References

CHAPTER

NINE

ROUTE CHOICE

The route choice problem does not have a closed solution, and the selection of one of the many existing frameworks for solution depends on many factors[1][2]. A common modelling framework in practice consists of two steps: choice set generation and the choice selection process.

AequilibraE is the first modeling package with full support for route choice, from the creation of choice sets through multiple algorithms to the assignment of trips to the network using the traditional path-size logit.

9.1 Choice set generation

Consistent with AequilibraE's software architecture, the route choice set generation is implemented as a separate Cython module that integrates into existing AequilibraE infrastructure; this allows it to benefit from established optimisations such as graph compression and high-performance data structures.

A key point of difference in AequilibraE's implementation comes from its flexibility in allowing us to reconstruct a compressed graph for computation between any two points in the network. This is a significant advantage when preparing datasets for model estimation, as it is possible to generate choice sets between exact network positions collected from observed data (e.g. vehicle GPS data, location-based services, etc.), which is especially relevant in the context of micro-mobility and active modes.

There are two different route choice set generation algorithms available in AequilibraE: Link Penalisation (LP), and Breadth-First Search with Link-Elimination (BFS-LE). The underlying implementation relies on the use of several specialized data structures to minimise the overhead of route set generation and storage, as both methods were implemented in Cython for easy access to existing AequilibraE methods and standard C++ data structures.

The process is designed to run multiple calculations simultaneously across the origin-destination pairs, utilising multi-core processors and improving computational performance. As Rieser-Schüssler *et al.* (2012)[1]_ noted, pathfinding is the most time-consuming stage in generating a set of route choices. Despite the optimisations implemented to reduce the computational load of maintaining the route set generation overhead, computational time is still not trivial, as pathfinding remains the dominant factor in determining runtime.

9.1.1 A* searches

Both methods use Dijkstra by default. To use A*, supply coordinates when constructing `RouteChoice` and select the search options through `set_choice_set_generation`:

```
from aequilibrae.route_choice import RouteChoice
from aequilibrae.routing import estimate_heuristic_scale

# coordinates is a DataFrame with planar x/y columns indexed by external node ID.
```

(continues on next page)

[1] Rieser-Schüssler, N., Balmer, M., and Axhausen, K.W. (2012). Route choice sets for very high-resolution data. Transportmetrica A: Transport Science, 9(9), 825–845. https://doi.org/10.1080/18128602.2012.671383

[2] Zill, J.C. and Camargo, P.V. (2024) State-Wide Route Choice Models. Presented at the ATRF, Melbourne, Australia.

(continued from previous page)

```
scale = estimate_heuristic_scale(graph, coordinates)
choice = RouteChoice(graph, coordinates=coordinates)
choice.set_choice_set_generation(
    "bfsle", max_routes=5, a_star=True, heuristic="euclidean", heuristic_scale=scale
)
```

Euclidean A* requires coordinates for every graph node in a planar coordinate system. For Haversine A*, use `heuristic="haversine"` and explicitly supply a DataFrame with `lon` and `lat` columns in degrees to both `estimate_heuristic_scale()` and `RouteChoice()`. Graphs do not retain coordinates.

A* requires an explicit finite, nonnegative scale to convert distance to cost units. A scale above a consistent bound can give non-shortest paths and change the route sets. A consistent bound remains valid when links are removed or their costs increase. Both heuristics support turn penalties and prohibited turns. Equal-cost paths may be selected differently from Dijkstra, which can also change later routes in the set.

9.1.2 Link-Penalisation

The link penalisation (LP) method is one of the most traditional approaches for generating route choice sets. It consists of an iterative approach where, in each iteration, the shortest path between the origin and the destination in question is computed. After each iteration, however, a pre-defined penalty factor is applied to all links that are part of the path found, essentially modifying the graph to make the previously found path less attractive.

The LP method is a simple and effective way to generate route choice sets, but it is sensitive to the penalty factor, which can significantly affect the quality of the generated choice sets, requiring experimentation during the model development/estimation stage. For both LP and BFS-LE, the factor must be finite and at least 1. A factor of 1 disables penalisation.

The overhead of the LP method is negligible due to AequilibraE's internal data structures that allow for easy data manipulation of the graph in memory.

9.1.3 BFS-LE

At a high level, BFS-LE operates on a graph of graphs, exploring unique graphs linked by a single removed edge. Each graph can be uniquely categorised by a set of removed links from a common base graph, allowing us to avoid explicitly maintaining the graph of graphs. Instead, generating and storing that graph's set of removed links in the breadth-first search (BFS) order.

To efficiently store and determine the uniqueness of a new route or removed link sets, we used modified hash functions with properties that allowed us to store and nest them within standard C++ data structures. We used a commutative hash function for the removed link sets to allow for amortised $O(1)$ order-independent uniqueness testing. While the removed link sets are always constructed incrementally, we did not opt for an incremental hash function as we did not deem this a worthwhile optimisation. The removed link sets rarely grew larger than double digits, even on a network with over 600,000 directed links. This may be an area worth exploring for networks with a significantly larger number of desired routes than links between ODs.

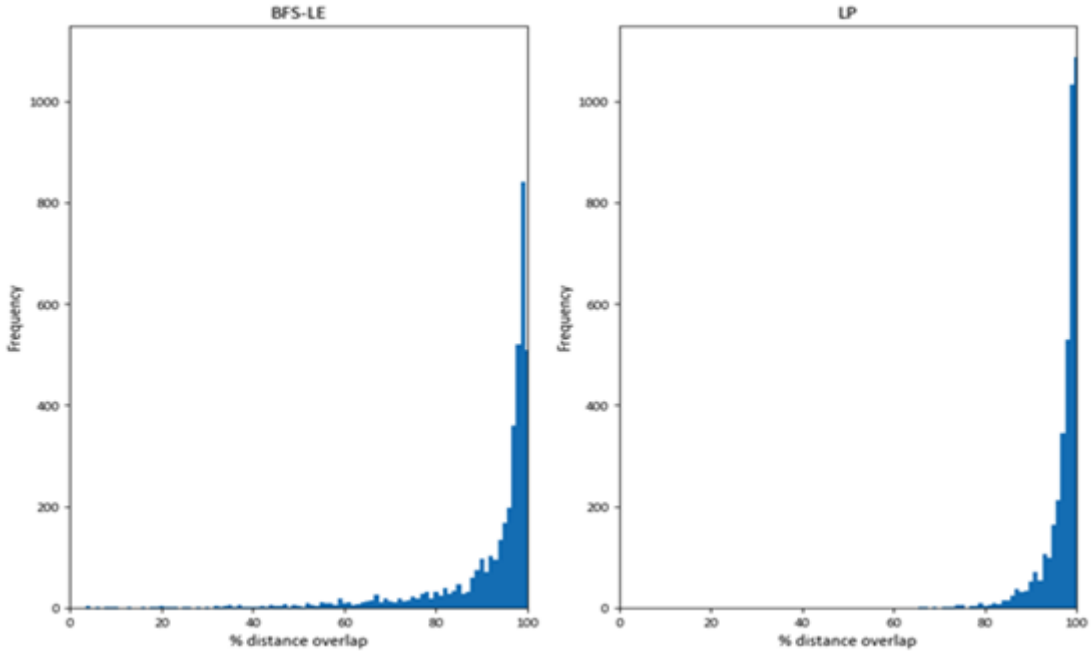
For uniqueness testing of discovered routes, AequilibraE implements a traditional, non-commutative hash function. Since cryptographic security was not a requirement for our purposes, we use a fast general-purpose integer hash function. Further research could explore the use of specialised integer vector hash functions. As we did not find the hashing had a non-negligible influence on the runtime performance, this optimisation was not tested.

AequilibraE also implements a combination of LP and BFS-LP as an optional feature to the latter algorithm, as recommended by Rieser-Schüssler *et al.* (2012)[1], which is also a reference for further details on the BFS-LE algorithm.

[1] Rieser-Schüssler, N., Balmer, M., and Axhausen, K.W. (2012). Route choice sets for very high-resolution data. Transportmetrica A: Transport Science, 9(9), 825–845. https://doi.org/10.1080/18128602.2012.671383

9.1.4 Comparative experiment

In an experiment with nearly 9,000 observed vehicle GPS routes covering a large Australian State, we found that all three algorithms (LP, BFS-LE, and BFS-LE+LP) had excellent performance in reproducing the observed routes. However, the computational overhead of BFS-LE is substantial enough to recommend always verifying if LP is fit-for-purpose.



9.1.5 References

9.2 Path-size logit (PSL)

Path-size logit is based on the multinomial logit (MNL) model, which is one of the most used models in the transportation field in general[1]. It can be derived from random utility-maximizing principles with certain assumptions on the distribution of the random part of the utility. To account for the correlation of alternatives, Ramming (2002)[2] introduced a correction factor that measures the overlap of each route with all other routes in a choice set based on shared link attributes, which gives rise to the PSL model. The PSL is currently the most used route choice model in practice, hence its choice as the first algorithm to be implemented in AequilibraE.

The PSL model's utility function is defined by:

$$U_i = V_i + \beta_{PSL} \times \log \gamma_i + \varepsilon_i$$

with path overlap correction factor:

$$\gamma_i = \sum_{a \in A_i} \frac{l_a}{L_i} \times \frac{1}{\sum_{k \in R} \delta_{a,k}}$$

Here, U_i is the total utility of alternative i, V_i is the observed utility, ε_i is an identical and independently distributed random variable with a Gumbel distribution, $\delta_{a,k}$ is the Kronecker delta, l_a is cost of link a, L_i is total cost of route i, A_i is the link set and R is the route choice set for individual j (index j suppressed for readability). The path overlap correction factor γ can be theoretically derived by aggregation of alternatives under certain assumptions, see[3] and references therein.

[1] Ben-Akiva, M., and Lerman, S. (1985) Discrete Choice Analysis. The MIT Press.

[2] Ramming, M.S. (2002) Network Knowledge and Route Choice. Massachusetts Institute of Technology. Available at: https://dspace.mit.edu/bitstream/handle/1721.1/49797/50436022-MIT.pdf?sequence=2&isAllowed=y

[3] Frejinger, E. (2008) Route Choice Analysis: Data, Models, Algorithms and Applications. Available at: https://infoscience.epfl.ch/server/api/core/bitstreams/6d43511f-e9c4-4fb4-b5c9-83a4515154b8/content

Notice that AequilibraE's path computation procedures require all link costs to be positive. For that reason, link utilities (or disutilities) must be positive, while its obvious minus sign is handled internally. This mechanism prevents the possibility of links with actual positive utility, but those cases are arguably not reasonable to exist in practice.

Important

AequilibraE uses cost to compute path overlaps rather than distance. Finite turn penalties are included in route cost and path overlap.

9.2.1 Route cost cutoff

A cost cutoff is available to mask unfavourable routes before applying the path-size logit assignment. For each OD pair, it masks routes whose cost exceeds `disutility_cutoff_constant + disutility_cutoff_coefficient * cheapest_route_cost`. Set both parameters to apply the cutoff, or omit both for no cutoff. See ../route_choice_transit_api_migration if you previously used `cutoff_prob`.

9.2.2 Full process overview

The estimation of route choice models based on vehicle GPS data can be explored on a family of papers scheduled to be presented at the ATRF 2024[4][5][6].

9.2.3 References

9.3 Examples

9.3.1 Route Choice

Route Choice with Turn Penalties

In this example, we show how route choice accounts for turn penalties during choice set generation and path-size logit calculations.

References

- *Route Choice*
- *Turn Restrictions*

See also

Several functions, methods, classes and modules are used in this example:

- *aequilibrae.routing.Network*
- *aequilibrae.route_choice.RouteChoice()*

Imports

[4] Zill, J.C. and Camargo, P.V. (2024) State-Wide Route Choice Models. Presented at the ATRF, Melbourne, Australia.

[5] Camargo, P.V. and Imai, R. (2024) Map-Matching Large Streams of Vehicle GPS Data into Bespoke Networks. Presented at the ATRF, Melbourne.

[6] Moss, J., Camargo, P.V., de Freitas, C. and Imai, R. (2024) High-Performance Route Choice Set Generation on Large Networks. Presented at the ATRF, Melbourne.

```
import numpy as np
import pandas as pd

from aequilibrae.routing import Network
from aequilibrae.route_choice import RouteChoice
```

Create a small network with two routes from node 10 to node 40.

```
links = pd.DataFrame(
    {
        "link_id": [71, 12, 55, 24],
        "a_node": [10, 20, 10, 30],
        "b_node": [20, 40, 30, 40],
        "direction": [1, 1, 1, 1],
        "time": [1.0, 1.0, 2.0, 2.0],
    }
)
```

Add a five-unit turn penalty to the direct route before building the graph.

```
turns = pd.DataFrame(
    {
        "from_node": [10],
        "via_node": [20],
        "to_node": [40],
        "penalty": [5.0],
    }
)
network = Network(links, centroids=np.array([10, 20, 30, 40]), turn_
↪restrictions=turns)
graph = network.build_graph("c", cost="time", compact=True, block_centroid_
↪flows=False)
```

Generate the route choice set. The direct route has a cost of 2 plus the turn penalty, so the alternative route with a cost of 4 is more likely.

```
rc = RouteChoice(graph)
rc.set_choice_set_generation("bfsle", max_routes=2, max_depth=5)
rc.execute_single(10, 40, demand=1.0)

route_choice_results = rc.get_results()
print(route_choice_results)
```

We can also recalculate path-size logit results for routes imported from another source. Route link IDs use a positive sign for the AB direction and a negative sign for the BA direction.

```
imported_routes = pd.DataFrame(
    {
        "origin id": [10, 10],
        "destination id": [40, 40],
        "route set": [[71, 12], [55, 24]],
        "route directions": [[1, 1], [1, 1]],
    }
)
```

`recompute_psl` validates origin and destination endpoints, link connectivity and turn restrictions before calculating costs, overlap and probabilities.

```
imported_results = rc.recompute_psl(imported_routes)
print(imported_results)
```

Route Choice set generation

In this example, we show how to generate route choice sets for estimation of route choice models, using a a city in La Serena Metropolitan Area in Chile.

References

- *Route Choice*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.route_choice.route_choice()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join

import folium
import numpy as np
from aequilibrae.examples import create_example
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

Model parameters

Let's select a set of nodes of interest

```
od_pairs_of_interest = [(71645, 79385), (77011, 74089)]
nodes_of_interest = (71645, 74089, 77011, 79385)
```

Load the network snapshot

```
network = project.road.build_network()
# Several fields in the project are filled with NaNs.
# This is true, but we won't use those fields.
```

Build the car graph with distance as the cost field.

```
graph = network.build_graph(mode="c", cost="distance", centroids=np.array(nodes_of_
↪interest), compact=True)

# Available network modes:
network.modes

# We set the nodes of interest as centroids to make sure they are not simplified away␣
↪when we create the network
graph.centroids
```

Route Choice class

Here we'll construct and use the Route Choice class to generate our route sets

```
from aequilibrae.route_choice import RouteChoice
```

Graph construction has already prepared the compressed-link to network-link mapping. RouteChoice reuses that mapping and snapshots the routing costs.

```
rc = RouteChoice(graph)
```

It is highly recommended to set either `max_routes` or `max_depth` to prevent runaway results.

We'll also set a 5% penalty (`penalty=1.05`), which is likely a little too large, but it creates routes that are distinct enough to make this simple example more interesting.

```
rc.set_choice_set_generation("bfsle", max_routes=5, penalty=1.05)
rc.prepare(od_pairs_of_interest)
rc.execute(perform_assignment=True)

choice_set = rc.get_results()
```

If we were interested in storing the route choice result, we could also write them to disk using the `save_path_files` method.

```
# rc.save_path_files(path)
```

From those path files we could also perform a full assignment or select link analysis by using the `execute_from_path_files` method.

```
# rc.execute_from_path_files(path)
```

Or if we had externally computed route choice sets, we can use AequilibraEs assignment procedures by loading them with the `execute_from_pandas` method.

```
# rc.execute_from_pandas(path_files_df)
```

Plotting choice sets

Now we will plot the paths we just created for the second OD pair

```
# We get the data we will use for the plot: links, nodes and the route choice set
plot_routes = choice_set[(choice_set["origin id"] == 77011)]["route set"].values
```

(continues on next page)

(continued from previous page)

```
links = project.tables.links.data

# For ease of plot, we create a GeoDataFrame for each route in the choice set
route_1 = links[links.link_id.isin(np.absolute(plot_routes[0]))]
route_2 = links[links.link_id.isin(np.absolute(plot_routes[1]))]
route_3 = links[links.link_id.isin(np.absolute(plot_routes[2]))]
route_4 = links[links.link_id.isin(np.absolute(plot_routes[3]))]
route_5 = links[links.link_id.isin(np.absolute(plot_routes[4]))]

nodes = project.tables.nodes.data
nodes = nodes[nodes["node_id"].isin([77011, 74089])]
```

```
map = route_1.explore(color="red", style_kwds={"weight": 3}, name="route_1")
map = route_2.explore(m=map, color="blue", style_kwds={"weight": 3}, name="route_2")
map = route_3.explore(m=map, color="green", style_kwds={"weight": 3}, name="route_3")
map = route_4.explore(m=map, color="purple", style_kwds={"weight": 3}, name="route_4")
map = route_5.explore(m=map, color="orange", style_kwds={"weight": 3}, name="route_5")

map = nodes.explore(m=map, color="black", style_kwds={"radius": 5, "fillOpacity": 1.0}
↪, name="network_nodes")

folium.LayerControl().add_to(map)
map
```

```
project.close()
```

Route Choice

In this example, we show how to perform route choice set generation using BFSLE and Link penalisation, for a city in La Serena Metropolitan Area in Chile.

References

- *Route Choice*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- `aequilibrae.route_choice.route_choice()`
- `aequilibrae.matrix.aequilibrae_matrix()`

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
```

(continues on next page)

(continued from previous page)

```
from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Model parameters

```
import numpy as np
```

We'll set the parameters for our route choice model. These are the parameters that will be used to calculate the utility of each path. In our example, the utility is equal to $distance * theta$, and the path overlap factor (PSL) is equal to $beta$.

```
# Distance factor
theta = 0.00011

# PSL parameter
beta = 1.1
```

Let's select a set of nodes of interest

```
nodes_of_interest = (71645, 74089, 77011, 79385)
```

Let's load the source network snapshot

```
network = project.road.build_network()
# Several fields in the project are filled with NaNs.
# This is true, but we won't use those fields.
```

We also see what modes are available

```
network.modes
```

Add the utility field to the source network before graph preparation.

```
network.links["utility"] = network.links.distance * theta
graph = network.build_graph(
    mode="c", cost="utility", centroids=np.array(nodes_of_interest), compact=True
)
```

Mock demand matrix

We'll create a mock demand matrix with demand 1 for every zone and prepare it for computation.

```
from aequilibrae.matrix import AequilibraeMatrix

mat = AequilibraeMatrix(index=graph.centroids, index_name="zones")
mat["demand"] = np.full((graph.num_zones, graph.num_zones), 10.0)
mat["5x demand"] = np.full((graph.num_zones, graph.num_zones), 50.0)
```

Create plot function

Before dive into the Route Choice class, let's define a function to plot assignment results.

```
import folium
```

```
def plot_results(link_loads):

    link_loads = link_loads[link_loads["demand_tot"] > 0]
    max_load = link_loads["demand_tot"].max()
    links = project.tables.links.data
    loaded_links = links.merge(link_loads, on="link_id", how="inner")

    loads_lyr = folium.FeatureGroup("link_loads")

    # Maximum thickness we would like is probably a 10, so let's make sure we don't␣
↪go over that
    factor = 10 / max_load

    return loaded_links.explore(
        color="red",
        style_kwds={
            "style_function": lambda x: {
                "weight": x["properties"]["demand_tot"] * factor,
            }
        },
    )
```

Route Choice class

Here we'll construct and use the Route Choice class to generate our route sets

```
from aequilibrae.route_choice import RouteChoice
```

Graph construction has already prepared the compressed-link to network-link mapping. RouteChoice reuses that mapping and snapshots the routing costs.

```
rc = RouteChoice(graph)

# Let's check the default parameters for the Route Choice class
print(rc.default_parameters)
```

Let's add the demand. If it's not provided, link loading cannot be performed.

```
rc.set_demand(mat)
```

It is highly recommended to set either `max_routes` or `max_depth` to prevent runaway results.

```
rc.set_choice_set_generation("bfsle", max_routes=5)
```

We can now perform a computation for single OD pair if we'd like. Here we do one between the first and last centroid as well as an assignment.

```
results = rc.execute_single(77011, 74089, demand=1.0)
print(results[0])
```

Because we asked it to also perform an assignment we can access the various results from that.

```
res = rc.get_results()
res.head()
```

```
plot_results(rc.get_load_results())
```

Batch operations

To perform a batch operation we need to prepare the object first. We can either provide a list of tuple of the OD pairs we'd like to use, or we can provided a 1D list and the generation will be run on all permutations.

```
rc.prepare()
```

Now we can perform a batch computation with an assignment

```
rc.execute(perform_assignment=True)
res = rc.get_results()
res.head()
```

Since we provided a matrix initially we can also perform link loading based on our assignment results.

```
rc.get_load_results()
```

We can plot these as well

```
plot_results(rc.get_load_results())
```

Select link analysis

We can also enable select link analysis by providing the links and the directions that we are interested in. Here we set the select link to trigger when (7369, 1) and (20983, 1) is utilised in "sl1" and "sl2" when (7369, 1) is utilised.

```
rc.set_select_links({"sl1": [[(7369, 1), (20983, 1)]], "sl2": [[(7369, 1)]]})
rc.execute(perform_assignment=True)
```

We can get then the results in a Pandas DataFrame for both the network.

```
sl = rc.get_select_link_loading_results()
sl
```

We can also access the OD matrices for this link loading. These labelled matrices contain SciPy CSR arrays for ease of use. They're stored in a dictionary keyed by select link set name, with demand fields inside each matrix.

```
rc.get_select_link_od_matrix_results()
```

```
od_matrix = rc.get_select_link_od_matrix_results()["sl1"]["demand"]
od_matrix.toarray()
```

```
project.close()
```

Route Choice with sub-area analysis

In this example, we show how to perform sub-area analysis using route choice assignment, for a city in La Serena Metropolitan Area in Chile.

References

- *Route Choice*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.routing.graph()`
- `aequilibrae.route_choice.route_choice()`
- `aequilibrae.route_choice.sub_area()`
- `aequilibrae.matrix.aequilibrae_matrix()`

```
# Imports
import itertools
from os.path import join
from tempfile import gettempdir
from uuid import uuid4

import folium
import numpy as np
import pandas as pd

from aequilibrae.examples import create_example
from aequilibrae.utils import basic_config
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
# We'll also apply a basic logging configuration.

basic_config()
```

Model parameters

We'll set the parameters for our route choice model. These are the parameters that will be used to calculate the utility of each path. In our example, the utility is equal to $distance * theta$, and the path overlap factor (PSL) is equal to $beta$.

```
theta = 0.011  # Distance factor

beta = 1.1  # PSL parameter
```

Let's build all graphs

```
network = project.road.build_network()
# We get warnings that several fields in the project are filled with NaNs.
# This is true, but we won't use those fields.
```

We grab the graph for cars

```
network.modes
```

Let's say that utility is just a function of distance. So we build our *utility* field as the $distance * theta$.

```
network.links["utility"] = network.links.distance * theta
```

Prepare a compact graph for the project centroids.

```
graph = network.build_graph(mode="c", cost="utility", compact=True)
```

Mock demand matrix

We'll create a mock demand matrix with demand *10* for every zone and prepare it for computation.

```
from aequilibrae.matrix import AequilibraeMatrix, from_df

mat = AequilibraeMatrix(index=graph.centroids, index_name="zones")
mat["demand"] = np.full((graph.num_zones, graph.num_zones), 10.0)
```

Sub-area preparation

We need to define some polygon for out sub-area analysis, here we'll use a section of zones and create out polygon as the union of their geometry. It's best to choose a polygon that avoids any unnecessary intersections with links as the resource requirements of this approach grow quadratically with the number of links cut.

```
zones_of_interest = [29, 30, 31, 32, 33, 34, 37, 38, 39, 40, 49, 50, 51, 52, 57, 58,␣
↪59, 60]
zones = project.tables.zones.data.set_index("zone_id")
zones = zones.loc[zones_of_interest]
zones.head()
```

Sub-area analysis

From here there are two main paths to conduct a sub-area analysis, manual or automated. AequilibraE ships with a small class that handle most of the details regarding the implementation and extract of the relevant data. It also exposes all the tools necessary to conduct this analysis yourself if you need fine grained control.

Automated sub-area analysis

We first construct out `SubAreaAnalysis` object from the graph, zones, and matrix we previously constructed, then configure the route choice assignment and execute it. From there the `post_process` method is able to use the route choice assignment results to construct the desired demand matrix as a DataFrame. If we were interested in the original origin and destination IDs for each entry we could use *subarea.post_process(keep_original_ods=True)* instead. This will attach the true ODs from the select link OD matrix as part of the index. However, this will create a significantly larger, but more flexible matrix.

```
from aequilibrae.route_choice import SubAreaAnalysis

subarea = SubAreaAnalysis(graph, zones, mat, project=project)
subarea.rc.set_choice_set_generation("lp", max_routes=3, penalty=1.02, store_
↪results=False)
subarea.rc.execute(perform_assignment=True)
demand = subarea.post_process()
demand
```

We'll re-prepare our graph but with our new "external" ODs.

```
new_centroids = np.unique(demand.reset_index()[["origin id", "destination id"]].to_
↪numpy().reshape(-1))
graph = network.build_graph(mode="c", cost="utility", centroids=new_centroids,␣
↪compact=True)
new_centroids
```

We can then perform an assignment using our new demand matrix on the limited graph

```
from aequilibrae.route_choice import RouteChoice

demand_matrix = from_df(
    demand.reset_index(),
    row="origin id",
    col="destination id",
    subset=list(demand.columns),
    index=new_centroids,
    index_name="zones",
    sparse=True,
)

rc = RouteChoice(graph)
rc.set_demand(demand_matrix)
rc.set_choice_set_generation("lp", max_routes=3, penalty=1.02, store_results=False,␣
↪seed=123)
rc.execute(perform_assignment=True)
```

Let's take the union of the zones GeoDataFrame as a polygon

```
poly = zones.union_all()
poly
```

And prepare the sub-area to plot.

```
subarea_zone = folium.Polygon(
    locations=[(x[1], x[0]) for x in poly.boundary.coords],
```

(continues on next page)

(continued from previous page)

```
    fill_color="blue",
    fill_opacity=0.1,
    fill=True,
    weight=1,
)
```

We create a function to plot out link loads data more easily

```
def plot_results(link_loads):
    link_loads = link_loads[link_loads["demand_tot"] > 0]
    max_load = link_loads[["demand_tot"]].max()
    links = project.tables.links.data
    loaded_links = links.merge(link_loads, on="link_id", how="inner")
    factor = 10 / max_load

    return loaded_links.explore(
        color="red",
        style_kwds={
            "style_function": lambda x: {
                "weight": x["properties"]["demand_tot"] * factor,
            }
        },
    )
```

And plot our data!

```
map = plot_results(rc.get_load_results())
subarea_zone.add_to(map)
map
```

Sub-area further preparation

It's useful later on to know which links from the network cross our polygon.

```
links = project.tables.links.data
inner_links = links[links.crosses(poly.boundary)].sort_index()
inner_links.head()
```

As well as which nodes are interior.

```
nodes = project.tables.nodes.data.set_index("node_id")
inside_nodes = nodes.sjoin(zones, how="inner").sort_index()
inside_nodes.head()
```

Let's filter those network links to links retained by the graph, including links inside contracted chains, and create a *link_id*, *direction* multi-index.

```
mapping = graph.mapping
source = mapping.network
retained = mapping.network_link_to_graph >= 0
g = pd.DataFrame(
    {
```

(continues on next page)

(continued from previous page)

```
        "link_id": source.link_ids[retained],
        "direction": source.directions[retained],
        "a_node": source.node_ids[source.link_tails[retained]],
        "b_node": source.node_ids[source.link_heads[retained]],
    }
)
g = g.loc[g.link_id.isin(inner_links.link_id)].set_index(["link_id", "direction"])
g.head()
```

Here we'll quickly visualise what our sub-area is looking like. We'll plot the polygon from our zoning system and the links that it cuts.

```
map = inner_links.explore(color="red", style_kwds={"weight": 4})
subarea_zone.add_to(map)
map
```

Manual sub-area analysis

Here we'll construct and use the Route Choice class to generate our route sets,

In order to perform out analysis we need to know what OD pairs have flow that enters and/or exists our polygon. To do so we perform a select link analysis on all links and pairs of links that cross the boundary. We create them as tuples of tuples to make represent the select link AND sets.

```
entering = g.loc[~g.a_node.isin(inside_nodes.index) & g.b_node.isin(inside_nodes.
↪index)].index
exiting = g.loc[g.a_node.isin(inside_nodes.index) & ~g.b_node.isin(inside_nodes.
↪index)].index
edge_pairs = {pair: (pair,) for pair in itertools.product(entering, exiting)}
single_edges = {x: ((x,),) for x in g.index}
f"Created: {len(edge_pairs)} edge pairs from {len(single_edges)} edges"
```

Let's prepare our graph once again

```
graph = network.build_graph(mode="c", cost="utility", compact=True, block_centroid_
↪flows=False)
```

Graph construction has already prepared the compressed-link to network-link mapping. We need to supply a Graph and an AequilibraeMatrix or AequilibraeSparseMatrix via the `set_demand` method, if demand is not provided link loading cannot be performed.

```
rc = RouteChoice(graph)
rc.set_demand(mat)
```

Here we add the union of edges as select link sets.

```
rc.set_select_links(single_edges | edge_pairs)
```

For the sake of demonstration we limit out demand matrix to a few OD pairs.

```
ods_pairs_of_interest = [
    (4, 39),
    (92, 37),
```

(continues on next page)

(continued from previous page)

```
    (31, 58),
    (4, 19),
    (39, 34),
]
ods_pairs_of_interest = ods_pairs_of_interest + [(x[1], x[0]) for x in ods_pairs_of_
↪interest]

# Construct a sparse Aeq matrix for our sub-selection
demand = pd.DataFrame(
    {name: [mat.mat[name, o, d] for o, d in ods_pairs_of_interest] for name in mat},
    index=pd.MultiIndex.from_tuples(ods_pairs_of_interest, names=["origin id",
↪"destination id"]),
).sort_index()
demand_matrix = from_df(demand.reset_index(), "origin id", "destination id",␣
↪index=mat.index, sparse=True)

rc = RouteChoice(graph)
rc.set_demand(demand_matrix)
rc.set_select_links(single_edges | edge_pairs)
rc.demand
```

Perform the assignment

```
rc.set_choice_set_generation("lp", max_routes=3, penalty=1.02, store_results=False,␣
↪seed=123)
rc.execute(perform_assignment=True)
```

We can visualise the current links loads

```
map = plot_results(rc.get_load_results())
subarea_zone.add_to(map)
map
```

We'll pull out just OD matrix results as well we need it for the post-processing, we'll also use the returned SciPy CSR arrays directly.

```
sl_od = rc.get_select_link_od_matrix_results()
od_index = demand_matrix.index
edge_totals = {k: sl_od[k]["demand"] for k in single_edges}
edge_pair_values = {k: sl_od[k]["demand"] for k in edge_pairs}
```

For the post processing, we are interested in the demand of OD pairs that enter or exit the sub-area, or do both. For the single enters and exists we can extract that information from the single link select link results. We also need to map the links that cross the boundary to the origin/destination node and the node that appears on the outside of the sub-area.

```
from collections import defaultdict

entered = defaultdict(float)
exited = defaultdict(float)
for (link_id, dir), v in edge_totals.items():
    link = g.loc[link_id, dir]
    for (o, d), load in v.todok().items():
        o = od_index[o]
```

(continues on next page)

(continued from previous page)

```
        d = od_index[d]

        o_inside = o in inside_nodes.index
        d_inside = d in inside_nodes.index

        if o_inside and not d_inside:
            exited[o, link.b_node] += load
        elif not o_inside and d_inside:
            entered[link.a_node, d] += load
        elif not o_inside and not d_inside:
            pass
```

Here he have the load that entered the sub-area

```
entered
```

and the load that exited the sub-area

```
exited
```

To find the load that both entered and exited we can look at the edge pair select link results.

```
through = defaultdict(float)
for (l1, l2), v in edge_pair_values.items():
    link1 = g.loc[l1]
    link2 = g.loc[l2]

    for (o, d), load in v.todok().items():
        o = od_index[o]
        d = od_index[d]
        o_inside = o in inside_nodes.index
        d_inside = d in inside_nodes.index

        if not o_inside and not d_inside:
            through[link1.a_node, link2.b_node] += load

through
```

With these results we can construct a new demand matrix. Usually this would be now transplanted onto another network, however for demonstration purposes we'll reuse the same network.

```
manual_od_pairs = list(entered.keys()) + list(exited.keys()) + list(through.keys())
manual_od_pairs = np.asarray(manual_od_pairs, dtype=np.uint64)
demand = pd.DataFrame(
    list(entered.values()) + list(exited.values()) + list(through.values()),
    index=pd.MultiIndex.from_arrays(
        [manual_od_pairs[:, 0], manual_od_pairs[:, 1]], names=["origin id",
↪"destination id"]
    ),
    columns=["demand"],
).sort_index()
demand.index = demand.index.set_levels([level.astype(np.int64) for level in demand.
```

(continues on next page)

(continued from previous page)

```
↪index.levels])
demand.head()
```

We'll re-prepare our graph but with our new "external" ODs.

```
new_centroids = np.unique(demand.reset_index()[["origin id", "destination id"]].to_
↪numpy(dtype=np.int64).reshape(-1))
graph = network.build_graph(mode="c", cost="utility", centroids=new_centroids,␣
↪compact=True)
new_centroids
```

Re-perform our assignment

```
rc = RouteChoice(graph)
rc.set_demand(from_df(demand.reset_index(), "origin id", "destination id", index=new_
↪centroids, sparse=True))
rc.set_choice_set_generation("lp", max_routes=3, penalty=1.02, store_results=False,␣
↪seed=123)
rc.execute(perform_assignment=True)
```

And plot the link loads for easy viewing

```
map = plot_results(rc.get_load_results())
subarea_zone.add_to(map)
map
```

```
project.close()
```

9.4 References

CHAPTER

TEN

OTHER APPLICATIONS

In this section, we bring some of AequilibraE's applications that do not match a specific subject.

10.1 SimWrapper Extension

SimWrapper is a browser-based, client-side tool for exploring transport simulation outputs. It accepts simple YAML configuration files describing dashboards of maps, tables and charts. Nothing is uploaded to the internet, everything is kept local. The application is open-source and designed so that non-coding stakeholders can open a dashboard in their browser and interact with results.

The SimWrapper extension generates portable dashboard configuration for visualising an AequilibraE project with the simwrapper.app or explore.outerloop.io. At the time of writing Outer Loop is hosting a preview version of SimWrapper while the required changes are being unstreamed.

You can view an example dashboard at explore.outerloop.io by selecting "Chicago AequilibraE Example" from the Example Dashboards section.

10.1.1 Usage

- The generator is implemented as `aequilibrae.utils.simwrapper.generate_simwrapper_config.SimwrapperConfigGenerator`.
- A CLI entry point is installed as the `aeq-sim` script (registered in the package entry points).
- Output: a `dashboard.yaml` file and data files (CSV and Vega-Lite JSON) written into the chosen `<output_dir>`. A data subfolder named `simwrapper_data/` is created under the output directory for CSV and Vega-Lite spec files.

Quickstart — CLI

```
$ aeq-sim --project /path/to/project --output-dir simwrapper
```

If you omit `--project`, the CLI defaults to the current directory (`.`). If you omit `--output-dir`, the CLI will write to a *simwrapper* folder by default.

CLI options

- `--project` / `-p`: project root (folder containing `project_database.sqlite`).
- `--output-dir` / `-o`: output directory (created inside the project).
- `--max-results-tables`: limits the number of results scenarios included. When not provided, the generator defaults to three results.
- `--results-tables`: explicit list of results table names to include (space-separated, e.g. `--results-tables table_1 table_2`). When not provided, defaults to the first *max-results-tables* tables.

- `--centroid-link-types`: explicit link-type names considered centroid connectors (space-separated, e.g. `--centroid-link-types centroid connector`).
- `--quiet` / `-q`: suppress informational output.

Note

`output_dir` must reside inside the project directory. Absolute paths outside the project are rejected.

Quickstart — Python API

```
from aequilibrae import Project
from aequilibrae.utils.simwrapper.generate_simwrapper_config import␣
↪SimwrapperConfigGenerator

prj = Project('/path/to/project')

gen = SimwrapperConfigGenerator(prj, output_dir='simwrapper', max_results_tables=3)
gen.write_yamls()
```

The constructor accepts the same configuration knobs available in the CLI (`output_dir`, `max_results_tables`, `results_tables`, `centroid_link_types`). `output_dir` must be located inside the project directory; absolute paths outside the project will raise a `ValueError`.

What the generator writes

All generated files are written into the chosen output directory (`<output_dir>`).

- `<output_dir>/dashboard.yaml` — the SimWrapper dashboard configuration.
- Data files referenced by the dashboard (CSV and Vega-Lite JSON specs), for example assignment convergence CSVs and `assignment_convergence.vega.json`.

The generator instance exposes a `generated_files` mapping after writing the outputs. This dict-like object maps descriptive keys (for example `"dashboard"` or `"assignment_convergence"`) to `pathlib.Path` objects pointing to the written files. It is useful for programmatic inspection or other downstream processing.

Viewing the dashboard

Warning

SimWrapper requires a Chromium-based browser (such as Google Chrome or Microsoft Edge) to access local files.

1. Open simwrapper.app or explore.outerloop.io in your browser.
2. Select “view local files” then select the project folder (the one containing `project_database.sqlite`)

10.1.2 Notes

- The YAML dashboard references data files using a relative path, so the dashboard may be shared by sharing the entire project.
- If your project contains many results scenarios, use `--max-results-tables` or `results_tables` to control which scenarios are included; the generator attempts to pick the most recent scenarios automatically when not specified. Selection is based on the `timestamp` field when present (most recent first), with a fallback ordering by

`table_name` when timestamps are not available. When centroid link types are not specified the generator attempts to infer them from the project's link types (names containing "centroid" or "connector") or from the links table.

- If you do not want any results scenarios included, pass `--max-results-tables 0` to the CLI or `max_results_tables=0` to the Python API; the generator will omit results panels from the dashboard in that case.

10.2 Examples

10.2.1 Other applications

Creating Delaunay Lines

In this example, we show how to create AequilibraE's famous Delaunay Lines, but in Python.

For more on this topic, see its first publication.

We use the Sioux Falls example once again.

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.utils.create_delaunay_network.DelaunayAnalysis()`

```
# Imports
import pandas as pd
from uuid import uuid4
from os.path import join
import sqlite3
from tempfile import gettempdir
from geopandas import read_postgis

from aequilibrae.examples import create_example
from aequilibrae.utils.create_delaunay_network import DelaunayAnalysis
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
```

Get the Delaunay Lines generation class

```
da = DelaunayAnalysis(project)

# Let's create the triangulation based on the zones, but we could create based on the␣
↪network (centroids) too
da.create_network("zones")
```

Now we get the matrix we want and create the Delaunay Lines

```
demand = project.matrices.get_matrix("demand.omx", subset=["matrix"])
```

And we will call it 'delaunay_test'./ It will also be saved in the results_database.sqlite

```
da.assign_matrix(demand, "delaunay_test")
```

we retrieve the results

```
results = project.results.get_results("delaunay_test").set_index("link_id")
```

Now we get the matrix we want and create the Delaunay Lines

```
with project.db_connection as conn:
    links = read_postgis(
        "Select link_id, st_asBinary(geometry) geometry from delaunay_network",
        conn,
        geom_col="geometry",
        crs=4326
    )
    links.set_index("link_id", inplace=True)

df = links.join(results)

max_vol = df.matrix_tot.max()

df.plot(linewidth=4 * df["matrix_tot"] / max_vol, color="blue")
```

Close the project

```
project.close()
```

Create a zone system based on Hex Bins

In this example, we show how to create hex bin zones covering an arbitrary area.

We also add centroid connectors and a special generator zone to our network to make it a pretty complete example.

We use the Nauru example to create roughly 100 zones covering the whole modeling area as delimited by the entire network.

You are obviously welcome to create whatever zone system you would like, as long as you have the geometries for them. In that case, you can just skip the hex bin computation part of this notebook.

References

- *Accessing project zones*

See also

Several functions, methods, classes and modules are used in this example:

- `aequilibrae.project.network.zones()`
- `aequilibrae.project.network.nodes()`

```
# Imports
from uuid import uuid4
```

(continues on next page)

(continued from previous page)

```
from tempfile import gettempdir
from os.path import join
from math import sqrt
from shapely.geometry import Point
import shapely.wkb

from aequilibrae.examples import create_example, list_examples
from aequilibrae.utils.aeq_signal import simple_progress, SIGNAL
s = SIGNAL(object)
```

Let's print the list of examples that ship with AequilibraE

```
print(list_examples())
```

```
# We create an empty project on an arbitrary folder
fldr = join(gettempdir(), uuid4().hex)

# Let's use the Nauru example project for display
project = create_example(fldr, "nauru")
```

We said we wanted 100 zones

```
zones = 100
```

Hex Bins using Spatialite

Spatialite requires a few things to compute hex bins. One of them is the area you want to cover.

```
network = project.tables
```

So we use the convenient network method `convex_hull()` (it may take some time for very large networks)

```
geo = network.convex_hull()
```

The second thing is the side of the hex bin, which we can compute from its area. The approximate area of the desired hex bin is

```
zone_area = geo.area / zones
```

Since the area of the hexagon is $\frac{3\sqrt{3}}{2} * side^2$ the side is equal to $\sqrt{\frac{2\sqrt{3}*area}{9}}$

```
zone_side = sqrt(2 * sqrt(3) * zone_area / 9)
```

Now we can run an SQL query to compute the hexagonal grid. There are many ways to create hex bins (including with a GUI on QGIS), but we find that using SpatiaLite is a pretty neat solution, for which we will use the entire network bounding box to make sure we cover everything.

```
extent = network.extent()
```

```
b = extent.bounds
sql = "select st_asbinary(HexagonalGrid(GeomFromWKB(?), ?, 0, GeomFromWKB(?)))"
with project.db_connection as conn:
```

(continues on next page)

(continued from previous page)

```
    grid = conn.execute(sql, [extent.wkb, zone_side, Point(b[2], b[3]).wkb]).
↪fetchone()[0]
    grid = shapely.wkb.loads(grid)
```

Since we used the bounding box, we have way more zones than we wanted, so we clean them by only keeping those that intersect the network convex hull.

```
grid = [p for p in grid.geoms if p.intersects(geo)]
```

Let's re-number all nodes with IDs smaller than 300 to something bigger as to free space to our centroids to go from 1 to N.

```
nodes = network.nodes
for i in range(1, 301):
    nodes.renumber(i, i + 1300)
```

```
# Now we can add them to the model and add centroids to them while we are at it.
zone_table = project.tables.zones
for i, zone_geo in enumerate(simple_progress(grid, s, "Add zone centroids")):
    zone_id = i + 1
    zone_table.insert(zone_id=zone_id, geometry=zone_geo)
    # Omitting the point adds the centroid at the geometric center of the zone.
    zone_table.add_centroid(zone_id)
```

Centroid connectors

Let's connect our zone centroids to the network.

```
for zone in zone_table:
    # We will connect for walk, with 1 connector per zone
    nodes.connect_mode(zone.zone_id, mode_id="w", connectors=1, area=zone.geometry)

    # And for cars, for cars with 2 connectors per zone
    # We also specify the link types we accept to connect to (can be used to avoid␣
↪connection to ramps or freeways)
    nodes.connect_mode(zone.zone_id, mode_id="c", link_types="ytrusP", connectors=2,␣
↪area=zone.geometry)

    # This takes a few minutes to compute, so we will break after processing the␣
↪first 10 zones
    if zone.zone_id >= 10:
        break
```

Special generator zones

Let's add a special generator zone by adding a centroid at the airport terminal.

Let's use some silly number for its ID, like 10,000, just so we can easily differentiate it

```
airport = Point(166.91749582, -0.54472590)
nodes.new_centroid(10000, airport)
```

When connecting a centroid not associated with a zone, we need to tell AequilibraE what is the initial area around the centroid that needs to be considered when looking for candidate nodes.

```
nodes.connect_mode(10000, mode_id="c", link_types="ytrusP", connectors=1,␣
↪area=airport.buffer(0.01))
```

```
project.close()
```

CHAPTER

ELEVEN

API REFERENCE

aequilibrae
aequilibrae.project
aequilibrae.routing
aequilibrae.traffic
aequilibrae.traffic.vdf
aequilibrae.transit
aequilibrae.route_choice
aequilibrae.distribution
aequilibrae.matrix
aequilibrae.examples
aequilibrae.utils

11.1 aequilibrae

Classes

Project(project_path)	AequilibraE project class

class aequilibrae.**Project**(*project_path: PathLike | str*)

AequilibraE project class

Listing 1: Create Project

```
>>> new_project = Project.create(project_path)

# Safely closes the project
>>> new_project.close()
```

Listing 2: Open Project

```
>>> existing_project = Project(project_path)

>>> existing_project.close()
```

classmethod create(*project_path: PathLike | str*) → *Project*

Create a ready-to-use project.

Parameters
project_path – Full path to a new project data folder.

Returns
The open project, usable as a context manager.

Raises
FileExistsError – If the destination already exists.

classmethod open(*project_path: PathLike | str*) → *Project*

activate() → None

check_file_indices() → NoReturn

Makes results_database.sqlite and the matrices folder compatible with project database

clone_scenario(*scenario_name: str*, *description: str = ''*) → None

Clones the active scenario.

Arguments
scenario_name (str): scenario name

description (str): useful scenario description

close() → None

Safely closes the project

create_empty_scenario(*scenario_name: str*, *description: str = ''*) → None

Creates an empty scenario, without any links, nodes, and zones.

Arguments
scenario_name (str): scenario name

description (str): useful scenario description

deactivate() → None

list_scenarios() → DataFrame

Lists the existing scenarios.

Returns
scenarios (pd.DataFrame): Pandas DataFrame with existing scenarios

log() → Log

Returns a log object

allows the user to read the log or clear it

transaction()

Enter transaction contexts for every existing database connection.

upgrade(*ignore_project: bool = False*, *ignore_transit: bool = False*, *ignore_results: bool = False*)

Find and apply all applicable migrations.

Each database upgrade is applied within a single transaction.

Optionally ignore specific databases. This is useful when a database is known to be incompatible with some migrations but you'd still like to upgrade the others. Take care when ignoring a database. For a particular version of AequilibraE, it is assumed that all migrations have been applied successfully or the project was created with the latest schema, skipping/ignoring migrations will likely lead to issues/broken assumptions.

If skipping a specific migration is required, use the aequilibrae.project.tools.MigrationManager object directly. Consult it's documentation page for details. Take care when skipping migrations.

Arguments

ignore_project (`bool`, optional): Ignore the project database. No direct migrations will be
applied. Defaults to False.

ignore_transit (`bool`, optional): Ignore the transit database. No direct migrations will be
applied. Defaults to False.

ignore_results (`bool`, optional): Ignore the results database. No direct migrations will be
applied. Defaults to False.

use_scenario(*scenario_name: str*) → None

Switch the active scenario.

Arguments

scenario_name (`str`): name of the scenario to be activated

property about: About

property db_connection: NestedTransactionManager

property matrices: Matrices

property parameters: dict

property path_to_file: Path

property project_base_path: Path

property project_parameters: *Parameters*

property results: Results

property results_connection: NestedTransactionManager

property road: RoadNetwork

Road network and graph builder for the current scenario.

property run: dict[str, partial]

Load and return the AequilibraE run module with the default arguments from `parameters.yml` partially applied, along with an explicit `project` argument.

Refer to `run/__init__.py` file within the project folder for documentation.

property tables: Tables

property transit: Transit

property transit_connection: NestedTransactionManager

11.2 aequilibrae.project

Classes

MigrationManager(migration_file[, database])	Small utility class to manage, validate, and apply a set of Migrations.
NetworkSimplifier([project])	
Parameters([path])	Global parameters module.

class aequilibrae.project.**MigrationManager**(*migration_file: Path*, *database: str | None = None*)

Small utility class to manage, validate, and apply a set of Migrations.

Arguments

migration_file (pathlib.Path): A path to a Python file which defines a global migrations variable as a list of pathlib.Path objects.

database (str, optional): Name of the closure connection that owns the migrations table. Defaults to 'transit' when the migration file lives under a transit directory and 'project' otherwise.

find_applicable(*closure: ConnectionClosure*)

Find all applicable migrations.

A migration is applicable if all migrations before it (ordered by ID) have been applied or skipped.

If an out-of-order migration is detected a RuntimeError will be raised and manual intervention will be required.

Arguments

closure (ConnectionClosure): The scenario's connection closure.

mark_all_as_seen(*closure: ConnectionClosure*)

Mark all migrations as 'seen'.

Marking a migration as 'seen' will add it to the migrations table as MISSING if it is not already present. If it is present no change is made.

Arguments

closure (ConnectionClosure): The scenario's connection closure.

status(*closure: ConnectionClosure*) → dict[int, MigrationStatus]

Query the database for all migrations' status.

If the migrations table is not present all migrations are considered MISSING.

Arguments

closure (ConnectionClosure): The scenario's connection closure.

Returns

status (dict[int, MigrationStatus]): Migration status enums by their ID.

upgrade(*closure: ConnectionClosure*, *skip: set[int] | None = None*)

Find and apply all applicable migrations in one closure-owned transaction.

Optionally skip some migrations. Take care when skipping migrations.

Arguments

closure (ConnectionClosure): The scenario's connection closure.

skip (set[int]): Set of migration IDs to skip.

network_migration_file: Path = PosixPath('/home/runner/work/aequilibrae/aequilibrae/build-install/usr/lib/python3.11/site-packages/aequilibrae/project/database_specification/network/migrations/migrations.py')

transit_migration_file: Path = PosixPath('/home/runner/work/aequilibrae/aequilibrae/build-install/usr/lib/python3.11/site-packages/aequilibrae/project/database_specification/transit/migrations/migrations.py')

class aequilibrae.project.**NetworkSimplifier**(*project:* Project | *None* = *None*)

collapse_links_into_nodes(*links: List[int]*)

Collapses links into nodes, adjusting the network in the neighbourhood.

Arguments

links (`List[int]`): List containing link IDs to be collapsed.

simplify(*graph:* Graph, *max_speed_ratio: float = 1.1*)

Simplifies the network by merging links that are shorter than a given threshold

Arguments

graph (`Graph`): AequilibraE graph. Prepared graphs must use compact topology.

max_speed_ratio (`float`, *Optional*): Maximum ratio between the fastest and slowest speed for a link to be considered for simplification.

signal = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.project.**Parameters**(*path: PathLike | str | None = None*)

Global parameters module.

Parameters are used in many procedures, and are often defined in the `parameters.yml` file ONLY.

Parameters are organised in the following groups:

- assignment
- distribution
- network * links * modes * nodes * osm * gmns
- osm
- system

Please observe that OSM information handled on network is not the same on the OSM group.

```
>>> from aequilibrae.project import Parameters

>>> project = Project.create(project_path)

>>> p = project.project_parameters

>>> p.parameters['system']['logging_directory'] =  "/path_to/other_logging_
↪directory"
>>> p.parameters['osm']['overpass_endpoint'] = "http://192.168.0.110:32780/api"
>>> p.parameters['osm']['timeout'] = 180
>>> p.write_back()

>>> # You can also restore the software default values
>>> p.restore_default()

>>> project.close()
```

classmethod load_default()

get_vdfs(*exclude_builtins: bool = False*, *function_map: dict[str, tuple[Callable, Callable]] | None = None*)

Gets Volume Delay Functions (VDFs) specified in the parameters "vdfs" entry, as well as preset VDFs if exclude_builtins is False.

Each entry in the "vdfs" entry of the parameters must either be specifying a default vdf, or either a preset function via "function" or a custom functional form "functional_form". If "function" is specified, the name of a preset VDF or VDF in function_map must be specified, with an optional "spec" dict overriding the preset's default parameters. If "functional_form" is specified with a string representation of the VDF (to be interpreted by NumExpr), the specification is required in "spec" and its derivative can be optionally included by its string representation in "derivative_functional_form". If a derivative of a custom VDF is not specified, it will use a finite difference scheme to calculate the derivative.

Arguments

exclude_builtins (`bool`, *Optional*): Setting to exclude the built in preset VDFs, for example bpr. Defaults to False, so presets are included **function_map** (`dict[str, tuple[Callable, Callable]]`, *Optional*): mapping of user supplied function names to (function, derivative) tuples, taking precedence over preset vdf definitions. Then the parameters can reference these functions by their name and spec in the "function" entry.

Returns

results (`dict[str, VDF]`): mapping of VDF names to their constructed VDF objects.

Raises

ValueError: if a "function" entry references a preset not found in function_map or a built in preset, if a "spec" contains keys not present in the preset's default spec, or if an entry defines neither "function" nor "functional_form".

restore_default()

Restores parameters to generic default

write_back() → None

Write the parameters back to file.

Raises

ValueError – If no project folder was supplied.

file_default: Path = PosixPath('/home/runner/work/aequilibrae/aequilibrae/build-install/usr/lib/python3.11/site-packages/aequilibrae/parameters.yml')

11.3 aequilibrae.routing

Functions

available_heaps()	Return the priority queue implementations.
estimate_heuristic_scale(graph[, ...])	Calculate a conservative A* coefficient.

Classes

Graph(routing, mapping, skimming, mode[, ...])	One routing context with fixed topology and mappings to source links.
Network(links, *[, modes, centroids, ...])	A transportation network.
NetworkSkimming(graph[, origins, project])	
PathResults(graph, origin, destination[, ...])	Path computation result holder.

continues on next page

Table 5 – continued from previous page

SkimResults()	Network skimming result holder.

class aequilibrae.routing.**Graph**(*routing: GraphContext*, *mapping: GraphMapping*, *skimming: SkimmingContext*, *mode: str*, *cost_field: str | None = None*)

One routing context with fixed topology and mappings to source links.

Create graphs with `Network.build_graph()`. Network tables and construction intermediates are not retained. Routing, correspondence and skim fields belong to their respective sub-objects. Reconstruct the graph to change its configuration; routing costs can still be rebound independently of the fixed topology.

compute_path(*origin: int*, *destination: int*, ***kwargs: object*) → *PathResults*

Compute a path with nodes and mileposts on this built graph.

Additional arguments are passed to *PathResults*, including explicit coordinates and a heuristic scale when using A*.

compute_skims(*cores: int | None = None*) → *NetworkSkimming*

Compute centroid skims using this graph's routing and skim contexts.

disconnected_nodes() → ndarray

Identify nodes unable to reach and be reached from the main network.

The largest strongly connected component selects the main network. Blocked centroids can be endpoints, but cannot connect other nodes through themselves. Individual turn restrictions are not considered. With no active links, only the first retained node is considered connected.

Returns

Disconnected external node IDs in retained graph-node order.

selected_links(*selections: Mapping[str, Iterable[tuple[int, int]]]*) → dict[str, ndarray[tuple[Any, ...], dtype[int64]]]

Resolve named source-link selections, raising for links absent from this graph.

set_costs(*costs: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) → None

Replace directed network costs. Positive infinity closes a link.

property all_nodes: ndarray[tuple[Any, ...], dtype[uint64]]

property allow_uturns: bool

Whether U-turns were allowed when building the routing context.

property block_centroid_flows: bool

Whether the build requested blocking travel through centroids.

property centroid_count: int

Number of centroids supplied when building the routing context.

property centroids: ndarray[tuple[Any, ...], dtype[uint64]]

property compact: bool

Whether link contraction was requested when building the mapping.

property cost: ndarray[tuple[Any, ...], dtype[float64]]

cost_field: str | None = None

property has_turn_restrictions: bool

mapping: GraphMapping

mode: str

property num_links: int

property num_nodes: int

property num_zones: int

property remove_dead_ends: bool

Whether dead-end removal was requested when building the mapping.

routing: GraphContext

property skim_fields: list[str]

skimming: SkimmingContext

class aequilibrae.routing.**Network**(*links: DataFrame*, ** (Keyword-only parameters separator (PEP 3102))*, *modes: str | Iterable[str] | None = None*, *centroids: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None*, *turn_restrictions: DataFrame | None = None*, *allow_uturns: bool = False*, *lonlat: DataFrame | None = None*)

A transportation network.

Stores link attributes, travel modes, demand centroids, node coordinates and turn restrictions. The same network can support graphs with different modes, costs, skim fields and centroid-flow policies, while retaining a consistent directed-link order for costs and reported results. Modes are single-character mode codes, not mode names.

build_graph(*mode: str*, *cost: str | None = None*, ***, *centroids: ArrayLike | None = None*, *skimming: str | Iterable[str] = ()*, *compact: bool = False*, *remove_dead_ends: bool = False*, *block_centroid_flows: bool = True*, *turn_restrictions: pd.DataFrame | None = None*, *allow_uturns: bool | None = None*) → *Graph*

Prepare one routing graph from the stored source tables.

`mode` is a single-character mode code, not a mode name. Costs use the named field. `Graph.set_costs()` accepts replacement arrays in directed network order, including infinity for link closures. Link contraction (`compact`) and dead-end removal are independent, opt-in reductions. Both retain centroids and require them to be supplied. Mappings expand every retained graph link back to its source links.

field_values(*field: str*) → ndarray[tuple[Any, ...], dtype[float64]]

Return numeric values in the shared mapping's directed network order.

mode_mask(*mode: str*) → ndarray[tuple[Any, ...], dtype[bool]]

validate_centroids(*centroids: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]*) → None

validate_mode(*mode: str*) → None

allow_uturns: bool = False

centroids: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None

property directions: ndarray[tuple[Any, ...], dtype[int8]]

property fields: list[str]

property link_ids: ndarray[tuple[Any, ...], dtype[uint64]]

links: DataFrame

lonlat: DataFrame | None = None

mapping: NetworkMapping

modes: str | Iterable[str] | None = None

turn_restrictions: DataFrame | None = None

class aequilibrae.routing.**NetworkSkimming**(*graph:* Graph, *origins=None*, *project=None*)

```
>>> from aequilibrae.routing import NetworkSkimming

>>> project = create_example(project_path)
>>> graph = project.road.build_graph('c', 'distance', skimming=['distance'],␣
↪compact=True)

>>> skm = NetworkSkimming(graph)
>>> skm.execute()

# The skim report (if any error generated) is available here
>>> skm.report
[]

# To access the in-memory skim matrix
>>> matrix = skm.results.skims

# Or you can save the results to disk
>>> matrix.save_as_omx(project_path / 'skimming_result.omx')

>>> project.close()
```

doWork()

execute()

Runs the skimming process as specified in the graph.

set_cores(*cores: int*) → None

Sets number of cores (threads) to be used in computation

Value of zero sets number of threads to all available in the system, while negative values indicate the number of threads to be left out of the computational effort.

Resulting number of cores will be adjusted to a minimum of zero or the maximum available in the system if the inputs result in values outside those limits

Arguments

cores (`int`): Number of cores to be used in computation

signal = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.routing.**PathResults**(*graph: Graph*, *origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*, *heap: str | None = None*, *, *coordinates: DataFrame | None = None*, *heuristic_scale: float | None = None*)

Path computation result holder.

`path` contains local graph-link indices. Use `get_network_links()` to obtain physical link IDs and directions, expanding any contracted chains. `path_nodes` and `milepost` describe the retained graph nodes on the path.

```
>>> from aequilibrae.routing import PathResults
>>> project = create_example(project_path)
>>> graph = project.road.build_graph(
...     'c', 'distance', skimming=['distance', 'free_flow_time']
... )
>>> res = PathResults(graph, 1, 17)
>>> res.update_trace(9)
>>> project.close()
```

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*, *heap: str | None = None*, *, *heuristic_scale: float | None = None*) → None

Search the prepared graph and trace a path between external IDs.

With `early_exit`, stop when the destination is finalised. Other missing terminals may still be reachable. `update_trace` will search again if needed. For A*, a scale above a consistent bound can give non-shortest paths. Use *aequilibrae.routing.estimate_heuristic_scale()* to calculate a conservative coefficient before searching.

Arguments

origin (`int`): External ID of the search origin.

destination (`int`): External ID of the traced destination.

early_exit (`bool`): Stop after finalising the destination.

a_star (`bool`): Whether to use A* for this search.

heuristic (`str` or None): `euclidean` or `haversine`. None uses the selection from *set_heuristic()*.

heap (`str` or None): Priority queue implementation to use.

heuristic_scale (`float` or None): Finite, nonnegative coefficient required for A*.

Returns

`None`. The search arrays, skims and traced path are updated in place.

Raises

KeyError: If an external node ID is not in the prepared graph. **ValueError**: If required A* coordinates or the explicit scale are invalid.

get_heaps() → list[str]

Return the available priority queue implementation names.

Returns

`list[str]`: Names accepted by *set_heap()*.

get_heuristics() → list[str]

Return the available A* heuristic names.

Returns

`list[str]`: Names accepted by *set_heuristic()*.

get_network_links() → tuple[ndarray, ndarray] | None

Return network link IDs and directions for the currently traced path.

Compact graph links are expanded in traversal order.

Returns

A tuple of `(link_ids, directions)` arrays, with directions +1 or -1. Returns `None` when no path is available, and two empty arrays for a path from a node to itself.

reset() → None

Clear search results, paths, labels and skims in place.

Returns

`None`. Allocated result buffers and their read-only views remain valid, but contain their reset sentinel or infinity values.

set_graph_data(*graph:* Graph, *, *coordinates: DataFrame | None = None*) → None

Prepare graph data for path and skim computation.

Arguments

graph (*Graph*): Prepared graph with a cost field. Fixed topology, mappings and skim fields are
shared.

coordinates (`pandas.DataFrame`, optional): `x` and `y` columns for Euclidean A*, or `lon`
and `lat` columns in degrees for Haversine A*, indexed by external node ID. Copied for later searches.

Returns

`None`. The existing result buffers are replaced and cleared before the next search.

Raises

ValueError: If the graph does not have context-compatible IDs or a required node/link mapping is
invalid.

set_heap(*heap: str*) → None

Select the priority queue implementation used for path computation.

Arguments

heap (`str`): Name returned by *get_heaps()*.

Returns

`None`. The selected heap is used by subsequent searches.

set_heuristic(*heuristic: str*) → None

Select the heuristic used by A* path computation.

Arguments

heuristic (`str`): Name returned by *get_heuristics()*.

Returns

`None`. The selected heuristic is used by subsequent A* searches.

update_trace(*destination: int*) → None

Update the traced destination using the current search tree.

A partial early-exit search is rerun when it has not finalised the requested destination, retaining its algorithm, heuristic, scale and heap. A full search reuses its finalised state tree.

Arguments

destination (`int`): External ID of the new destination.

Returns

`None`. `path`, `path_nodes` and `milepost` are replaced for the requested destination.

Raises

RuntimeError: If no search has been performed. **KeyError**: If the destination is not in the prepared graph.

property connectors: ndarray

property distances: ndarray

property node_ids: ndarray

External node IDs in context and skim destination order.

property predecessors: ndarray

property settlement_order: ndarray

property terminal_states: ndarray

property turn_costs: ndarray

class aequilibrae.routing.**SkimResults**

Network skimming result holder.

```
>>> from aequilibrae.routing import SkimResults

>>> project = create_example(project_path)

# Mode c is car in this project
# minimize travel time
# Skims travel time and distance
>>> car_graph = project.road.build_graph(
...     'c', 'free_flow_time', skimming=['free_flow_time', 'distance'],␣
↪compact=True
... )

>>> res = SkimResults()
>>> res.prepare(car_graph)

>>> res.skims.save_as_omx(project_path / "skim_matrices.omx")

>>> project.close()
```

get_heaps() → List[str]

Return the available priority queue implementations.

prepare(*graph:* Graph, *output: SkimmingOutputs | None = None*)

Prepares the object with dimensions corresponding to the graph objects

Arguments

graph (*Graph*): Needs to have been set with number of centroids and list of skims (if any)

output (`SkimmingOutputs`, *Optional*): Existing output buffer to share.

set_heap(*heap: str*) → None

Set the priority queue implementation used for path computation. Must be one of `get_heaps()`.

Arguments

heap (`str`): Heap to use.

aequilibrae.routing.**available_heaps**() → list

Return the priority queue implementations.

aequilibrae.routing.**estimate_heuristic_scale**(*graph*, *coordinates=None*, ***, *heuristic='euclidean'*) → float

Calculate a conservative A* coefficient.

Arguments

graph (*Graph*): Prepared graph with a nonnegative cost field.

coordinates (`pandas.DataFrame`): `x` and `y` columns for
Euclidean distance, or `lon` and `lat` columns in degrees for Haversine distance, indexed by external node ID. Required for either heuristic. All retained graph nodes must be present in one common coordinate system.

heuristic (`str`): `euclidean` (default) or `haversine`.

Returns

`float`: The smallest finite link cost divided by its positive
endpoint distance, with a small round-off margin. Zero is returned if there are no such bounds. A zero-cost link can also force a zero scale.

This bound works for every destination, including nonnegative turn costs. It ignores turn and centroid restrictions, which can only remove paths or increase their costs. Recalculate after changing coordinates or decreasing link costs. Project geographic geometry to a suitable local CRS before supplying x/y.

11.4 aequilibrae.traffic

Classes

VDF(name, function, spec[, derivative])	Volume-Delay function that describes the total travel time across a link for a based on its free flow travel time and capacity for a given volume of traffic.
AssignmentResults()	Public results for one traffic class, backed by compact output owners.
TrafficAssignment(*, network, classes, ...)	Traffic assignment class.
TrafficClass(*, mode, cost, demand, name, ...)	Define a traffic class independently of a network or prepared graph.

class aequilibrae.traffic.**AssignmentResults**

Public results for one traffic class, backed by compact output owners.

Skims and selected OD demand are the routing output objects. Full-network link loads are reporting snapshots, not a second mutable assignment state. Every load and turn total is in the original demand units, without PCE.

static get_heaps()

bind(*state*, *class_names*)

Attach the accepted or latest AoN state at the start of execution.

get_load_results()

Return directional link loads in external network link order.

get_sl_results()

Return directional link loads for each selection and demand column.

reset()

set_cores(*cores*, *threading_threshold=None*, *elementwise_cores=None*)

Set thread counts without reallocating or clearing existing results.

set_heap(*heap*)

property compact_link_loads

property link_loads

Read-only snapshot of loads in supernetwork order.

property output

property select_link_loading

property select_link_od

property skims

property state

property total_link_loads

property total_turn_penalty

property unassigned_demand

class aequilibrae.traffic.**TrafficAssignment**(*, *network:* Network, *classes: list[*TrafficClass*]*, *time_field: str*, *capacity_field: str*, *vdf:* VDF, *algorithm: str*, *name_mapping=None*, *preload=None*, *project=None*)

Traffic assignment class.

The constructor retains a supplied standalone Network, then checks compatibility. Do not mutate that Network during use. `execute()` prepares each class's routing graph and returns link results. Classes share the network reporting mapping, but build their own graph mappings. Each class owns its costs, demand, skims, and output buffers.

For a comprehensive example on use, see the Use examples page.

```
>>> from aequilibrae.traffic import TrafficAssignment, TrafficClass
>>> from aequilibrae.traffic.vdf import bpr
>>> project = create_example(project_path)
>>> network = project.road.build_network()
>>> proj_matrices = project.matrices
```

(continues on next page)

(continued from previous page)

```
# We will only assign one user class stored as 'matrix' inside the OMX file
>>> demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])

# Creates the assignment class
>>> assigclass = TrafficClass(
...     name="car", mode="c", cost="free_flow_time", demand=demand,
...     skimming=['free_flow_time', 'distance'], graph_options={'block_centroid_
↪flows': False},
... )
>>> assig = TrafficAssignment(
...     network=network, project=project, classes=[assigclass], time_field='free_
↪flow_time',
...     capacity_field='capacity', vdf=bpr,
...     name_mapping={"alpha": "b", "beta": "power"}, algorithm='bfw',
... )
>>> results_df = assig.execute(max_iterations=10, rgap=0.00001)

# If you want, it is possible to access the convergence report
>>> convergence_report = pd.DataFrame(assig.assignment.convergence_report)

# Assignment results can be viewed as a Pandas DataFrame
>>> results_df = assig.results()

# Information on the assignment setup can be recovered with
>>> info = assig.info()

# Or save it directly to the results database
>>> results = assig.save_results(table_name='base_year_assignment')

# skims are here
>>> avg_skims = assig.class_results['car'].skims # blended ones

>>> project.close()
```

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: Available algorithm names

execute(*, *max_iterations=None*, *rgap=None*, *cores=None*, *threading_threshold=None*, *elementwise_cores=None*, *bfw_conjugacy='approximate'*, *steps_below_needed_to_terminate=1*, *log_specification=True*) → DataFrame

Prepare graphs, run assignment, and return link results.

Execution options apply to this run. Source data comes from the network retained by the constructor. Each run owns fresh routing contexts and outputs.

`bfw_conjugacy` chooses how BFW solves for its direction coefficients. Ignored by every other algorithm.

- `"approximate"` (default) - appendix A of Mitradjieva & Lindberg, which assumes the two previous directions are already conjugate with respect to the *current* Hessian. They were made conjugate with respect to the previous one, so the assumption holds only in the limit; the paper notes BFW is therefore "correct only in the limit".

- `"exact"` - solve both conjugacy conditions jointly as a 2x2 system, keeping the cross term the appendix drops. Costs two extra Hessian contractions per iteration, which is negligible next to the shortest-path build. The system is singular when the two previous directions are H-parallel, in which case the iteration restarts from Frank-Wolfe.

Either way, negative coefficients are clamped to zero to keep the weights on the simplex, and that clamping breaks strict biconjugacy whenever it triggers.

When the algorithm is BFW the convergence report gains `conjugacy_prev`, `conjugacy_prev2` and `hessian_drift` columns to compare the two variants; see `LinearApproximation`.

Arguments

cores (`int`): Number of CPU cores to use

threading_threshold (`int`, *Optional*): Minimum number of array elements for elementwise operations to be threaded. Negative values disable threading for those operations.

elementwise_cores (`int`, *Optional*): Number of CPU cores for elementwise (`parallel_numpy`/VDF) operations. When not provided, it is resolved from the `AEQ_ELEMENTWISE_CPUS` environment variable or `parameters.yml`, defaulting to at most 8 threads

`get_skim_results``()`

Return assignment skims by class name.

`info``()` → dict

Returns information for the traffic assignment procedure

Dictionary contains keys 'Algorithm', 'Classes', 'Computer name', 'Procedure ID', 'Maximum iterations' and 'Target RGap'.

The classes key is also a dictionary with all the user classes per traffic class and their respective matrix totals

Returns

info (`dict`): Dictionary with summary information

`log_specification``()`

`report``()` → DataFrame

Returns the assignment convergence report

Returns

DataFrame (`pd.DataFrame`): Convergence report

`results``()` → DataFrame

Prepares the assignment results as a Pandas DataFrame

Class load columns use demand matrix names without a traffic-class prefix. Names must be unique across classes and shared network metrics.

Returns

DataFrame (`pd.DataFrame`): Pandas DataFrame with all the assignment results indexed on *link_id*

`save_results`(*table_name: str*, *keep_zero_flows=True*, *project=None*) → None

Saves the assignment results to results_database.sqlite

Method fails if table exists

Arguments

table_name (`str`): Name of the table to hold this assignment result

keep_zero_flows (`bool`): Whether we should keep records for zero flows. Defaults to `True`

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

save_select_link_flows(*table_name: str*, *project=None*) → None

Saves the select link link flows for all classes into the results database.

Arguments

table_name (`str`): Name of the table being inserted to. Note the traffic class

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

save_select_link_matrices(*matrix_name: str*, *project=None*) → None

Saves the Select Link matrices for each TrafficClass in the current TrafficAssignment class into OMX format.

Arguments

name (`str`): name of the matrices

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

save_select_link_results(*name: str*) → None

Saves both the Select Link matrices and flow results at the same time, using the same name.

Arguments

name (`str`): name of the matrices

save_skims(*matrix_name: str*, *which_ones='final'*, *format='omx'*, *project=None*) → list[str]

Saves the skims (if any) to the project's matrix folder

Arguments

matrix_name (`str`): Base file name for the matrices, suffixed with each traffic class ID

which_ones (`str`, *Optional*): 'final': Results of the final iteration,
'blended': Averaged results for all iterations, 'all': Saves skims for both the final iteration and the blended ones. Default is 'final'

format (`str`, *Optional*): File format ('omx'). Default is 'omx'

project (`Project`, *Optional*): Project we want to save the results to.
Defaults to the explicitly supplied constructor project

Returns

file_names (`list`): OMX file names saved for classes with skims to save.

select_link_flows() → dict[str, DataFrame]

Returns a dataframe of the select link flows for each class

skim_congested(*skim_fields=None*, *return_matrices=False*) → dict | None

Skims the congested network. The user can add a list of skims to be computed, which will be added to the congested time and the assignment cost from the last iteration of the assignment.

The matrices are always stored internally in the AequilibraE objects to be saved to the project if needed. If return_matrices is set to True, the matrices are also returned.

Arguments

skim_fields (`Union[None, str]`): Name of the skims to use. If None, uses default only
return_matrices (`Bool`): Returns a dictionary with skims. Defaults to False.

all_algorithms = ['all-or-nothing', 'msa', 'frank-wolfe', 'fw', 'cfw', 'bfw']

all_bfw_conjugacies = ['approximate', 'exact']

property class_results

Return results by traffic class name after execution.

class aequilibrae.traffic.**TrafficClass**(*, *mode: str*, *cost: str*, *demand:* AequilibraeMatrix | AequilibraeSparseMatrix, *name: str | None = None*, *pce: float = 1.0*, *vot: float = 1.0*, *skimming: tuple[str, ...]=()*, *fixed_cost_field: str | None = None*, *fixed_cost_multiplier: float = 1.0*, *select_link_sets: Mapping[str, tuple]=<factory>*, *select_link_od: bool = True*, *graph_options: Mapping[str, bool]=<factory>*, *heap: str = '4ary'*)

Define a traffic class independently of a network or prepared graph.

`cost` is the free-flow generalised cost in time units. Congestion adds the change in travel time. Fixed costs are multiplied by `fixed_cost_multiplier` and divided by `vot`.

Demand matrices must contain float64 values. `name` defaults to the mode. Classes with the same mode need distinct names. `graph_options` accepts `compact`, `remove_dead_ends`, `block_centroid_flows` and `allow_uturns`. Omitted U-turn policy inherits the network setting.

```
>>> from aequilibrae.traffic import TrafficClass

>>> project = create_example(project_path)
>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand.omx", subset=["matrix"])

>>> tc = TrafficClass(
...     name="car", mode="c", cost='free_flow_time', demand=demand, pce=1.3,
...     skimming=['free_flow_time', 'distance'], graph_options={'block_centroid_
↪flows': False},
... )

>>> project.close()
```

cost: str

demand: *AequilibraeMatrix* | *AequilibraeSparseMatrix*

fixed_cost_field: str | None = None

fixed_cost_multiplier: float = 1.0

graph_options: Mapping[str, bool]

heap: str = '4ary'

property info

mode: str

name: str | None = None

pce: float = 1.0

select_link_od: bool = True

select_link_sets: Mapping[str, tuple]

skimming: tuple[str, ...] = ()

vot: float = 1.0

class aequilibrae.traffic.**VDF**(*name: str*, *function: Callable | str*, *spec: dict*, *derivative: Callable | str | None = None*)

Volume-Delay function that describes the total travel time across a link for a based on its free flow travel time and capacity for a given volume of traffic. Supports multiplicative and additive delays, and returns the absolute congested time rather than a delay or factor***

```
>>> from aequilibrae.traffic import VDF
```

static convert_str_function_into_function(*function_def: str*) → Callable

apply_derivative(*, *delta*, *link_flows*, *fftime*, *capacity*, *cores: int*, ***link_attributes*)

apply_vdf(*, *congested_time*, *link_flows*, *fftime*, *capacity*, *cores: int*, ***link_attributes*)

check_valid(*num_points*, *link_attributes: dict[str, Any]*, *from_voc: float = 0.0*, *to_voc: float = 3.0*)

Implements checks for VDF requirements described by Spiess (1990) in "Technical Note—Conical Volume-Delay Functions" (*Transportation Science*, 24(2): 153-158).

The VDF is evaluated at num_points values of volume/capacity in the range between from_voc to to_voc. Then, these values are used in the following checks:

It is strictly increasing, found by comparing neighbouring function values and checking that the derivative is positive. It is convex from above, found by checking that the derivative only increases between evaluated points For zero volume on the road, the VDF returns the free flow travel time

Any violations found are printed, listing the offending volume/capacity values.

Arguments

num_points (`int`): the number of volume/capacity values at which to evaluate the VDF and its derivative between from_voc and to_voc.

link_attributes (`dict[str, Any]`): mapping of link attribute names to values used to evaluate the VDF and its derivative, for example secondary capacity or the parameters for the VDF.

from_voc (`float`, *Optional*): the lower bound of the volume/capacity range to evaluate. Defaults to 0.0.

to_voc (`float`, *Optional*): the upper bound of the volume/capacity range to evaluate. Defaults to 3.0.

Returns

vdf_valid_0_value (`bool`): True if the VDF evaluates to 1 (the free flow travel time) at zero volume/capacity, within a small tolerance.

vdf_increasing_f_vals (`bool`): True if the VDF's values are non-decreasing across the evaluated volume/capacity range.

vdf_nonnegative_derivative (`bool`): True if the VDF's derivative is non-negative across the evaluated volume/capacity range.

vdf_convex (`bool`): True if the VDF's derivative is non-decreasing across the evaluated volume/capacity range (i.e. the VDF is convex from above).

make_finite_difference_derivative(*eps: float = 0.0001*) → Callable

plot_vdf(*num_points: int*, *link_attributes: dict[str, Any]*)

Creates a plot of the vdf using num_points for velocity/capacity in the range [0,3]. Link attributes are directly passed self.apply_vdf() and self.apply_derivative, so they may need to be a numpy array the size of num_points.

11.5 aequilibrae.traffic.vdf

Functions

builtin_vdfs()	

aequilibrae.traffic.vdf.**builtin_vdfs**() → dict[str, *VDF*]

11.6 aequilibrae.transit

Classes

TransitAssignment(*args[, project])	
TransitAssignmentResults()	Transit loads are owned by the hyperpath computation.
TransitClass(*, name, matrix, graph, results)	Transit demand bound to a transit graph for hyperpath assignment.
TransitGraph(network, centroids, time_field, ...)	

class aequilibrae.transit.**TransitAssignment**(**args*, *project=None*, ***kwargs*)

add_class(*transport_class:* TransitClass) → None

Adds a Transit class to the assignment

Arguments
transport_class (*TransitClass*): Transit class

algorithms_available() → list

Returns all algorithms available for use

Returns
list: Available algorithm names

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → dict

Return assignment skims by class name.

info() → dict

Returns information for the transit assignment procedure

Dictionary contains keys 'Algorithm', 'Classes', 'Computer name', 'Procedure ID'.

The classes key is also a dictionary with all the user classes per transit class and their respective matrix totals

Returns
info (dict): Dictionary with summary information

log_specification()

report() → DataFrame

Returns the assignment convergence report

Returns
DataFrame (pd.DataFrame): Convergence report

results() → DataFrame

Prepares the assignment results as a Pandas DataFrame

Returns
DataFrame (pd.DataFrame): Pandas DataFrame with all the assignment results indexed on *link_id*

save_results(*table_name: str*, *keep_zero_flows=True*, *project=None*) → None

Saves the assignment results to results_database.sqlite

Method fails if table exists

Arguments
table_name (str): Name of the table to hold this assignment result

keep_zero_flows (bool): Whether we should keep records for zero flows. Defaults to True

project (Project, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

set_algorithm(*algorithm: str*)

Chooses the assignment algorithm. Currently only 'optimal-strategies' is available.

'os' is also accepted as an alternative to 'optimal-strategies'

Arguments
algorithm (str): Algorithm to be used

set_classes(*classes: list[TransitClass]*) → None

Sets Transit classes to be assigned

Arguments
classes (List[TransitClass]): List of TransitClass's for assignment

set_cores(*cores: int*, *threading_threshold: int | None = None*, *elementwise_cores: int | None = None*) → None

Allows one to set the number of cores to be used AFTER transit classes have been added

Inherited from AssignmentResultsBase

Arguments
cores (int): Number of CPU cores to use

threading_threshold (int, *Optional*): Minimum number of array elements for elementwise operations to be threaded. Negative values disable threading for those operations. When not provided, the current value is kept

elementwise_cores (int, *Optional*): Number of CPU cores for elementwise (parallel_numpy/VDF) operations. When not provided, it is resolved from the AEQ_ELEMENTWISE_CPUS environment variable or parameters.yml, defaulting to at most 8 threads

set_frequency_field(*frequency_field: str*) → None

Sets the graph field that contains the frequency -> e.g. 'freq'

Arguments
frequency_field (`str`): Field name

set_skimming_fields(*skimming_fields: list[str] = None*) → None

Sets the skimming fields for the transit assignment.

Also accepts predefined skimming fields:

- discrete: 'boardings', 'alightings', 'inner_transfers', 'outer_transfers', and 'transfers'.
- continuous: 'trav_time', 'on_board_trav_time', 'dwelling_time', 'egress_trav_time', 'access_trav_time', 'walking_trav_time', 'transfer_time', 'in_vehicle_trav_time', and 'waiting_time'.

Provide no argument to disable.

Arguments
skimming_fields (`list[str]`): Optional list of field names, or predefined skimming type.

set_time_field(*time_field: str*) → None

Sets the graph field that contains free flow travel time -> e.g. 'trav_time'

Arguments
time_field (`str`): Field name

algorithm: str

all_algorithms = ['optimal-strategies', 'os']

classes: list

cores: int

description: str

elementwise_cores: int

free_flow_tt: ndarray

threading_threshold: int

time_field: str

total_flow: ndarray

class aequilibrae.transit.**TransitAssignmentResults**

Transit loads are owned by the hyperpath computation.

get_load_results()

prepare(*graph*, *matrix*)

reset()

set_cores(*cores*, *threading_threshold=None*, *elementwise_cores=None*)

Set thread counts without reallocating or clearing existing results.

class aequilibrae.transit.**TransitClass**(*, *name: str*, *matrix:* AequilibraeMatrix | AequilibraeSparseMatrix, *graph: object = None*, *results:* TransitAssignmentResults = <*factory*>)

Transit demand bound to a transit graph for hyperpath assignment.

graph: object = None

property info

matrix: *AequilibraeMatrix* | *AequilibraeSparseMatrix*

name: str

results: *TransitAssignmentResults*

class aequilibrae.transit.**TransitGraph**(*network: DataFrame*, *centroids: ndarray*, *time_field: str*, *frequency_field: str*, *od_node_mapping: DataFrame*, *a_node_field: str = 'a_node'*, *b_node_field: str = 'b_node'*, *skimming_fields: list[str] | None = None*, *config: dict | None = None*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (`str`): Skimmeable field names

clear_turn_restrictions() → None

Clears all turn restrictions and rebuilds the prepared graph.

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*, ***, *coordinates: pd.DataFrame | None = None*, *heuristic_scale: float | None = None*) → *PathResults*

Returns the results from path computation result holder.

Arguments

origin (`int`): origin for the path

destination (`int`): destination for the path

early_exit (`bool`): stop constructing the shortest path tree once the destination is found. Doing so may cause subsequent calls to 'update_trace' to recompute the tree. Default is `False`.

a_star (`bool`): whether or not to use A* over Dijkstra's algorithm. When `True`, 'early_exit' is always `True`. Default is `False`.

heuristic (`str`): `euclidean` (default) or `haversine` if A* is enabled.

coordinates (`pandas.DataFrame`, optional): Planar `x` and `y` columns indexed by external node ID, required for Euclidean A*. Haversine uses the graph's longitude/latitude.

heuristic_scale (`float`): Finite, nonnegative coefficient required for A*. Use `aequilibrae.routing.estimate_heuristic_scale` for a conservative bound. A larger scale can give non-shortest paths.

compute_skims(*cores: int | None = None*)

Returns the results from network skimming result holder.

Arguments

cores (`Union[int, None]`): number of cores (threads) to be used in computation

create_compressed_link_network_mapping() → tuple[ndarray[tuple[Any, ...], dtype[int64]], ndarray[tuple[Any, ...], dtype[int64]], ndarray[tuple[Any, ...], dtype[generic]]]

Create three arrays providing a mapping of compressed ID to link ID.

Uses sparse compression. Index 'idx' by the by compressed ID and compressed ID + 1, the network IDs are then in the range `idx[id]:idx[id + 1]`.

Links not in the compressed graph are not contained within the 'data' array.

'node_mapping' provides an easy way to check if a node index is present within the compressed graph. If the value is -1 then the node has been removed, either by compression of dead end link removal. If the value is greater than or equal to 0, then that value is the compressed node index.

Road graphs expose the corresponding arrays through `graph.mapping`. These use directed network link indices and dense network node indices, rather than the legacy graph IDs and external node IDs used by this method.

```
>>> project = create_example(project_path)

>>> network = project.road.build_network()
>>> graph = network.build_graph("c", centroids=np.arange(1, 25), compact=True,
↪ remove_dead_ends=True)

>>> idx = graph.mapping.expansion_offsets
>>> data = graph.mapping.expansion_links
>>> node_mapping = graph.mapping.network_node_to_graph

>>> project.close()
```

Returns

idx (`np.array`): index array for `data`

data (`np.array`): array of link ids

node_mapping: (`np.array`): array of node_mapping ids

default_types(*tp: str*)

Returns the default integer and float types used for computation

Arguments

tp (`str`): data type. 'int' or 'float'

disconnected_nodes() → ndarray

Identify disconnected nodes using strongly connected components analysis.

This method is not available for builder graphs. Use a prepared Graph instead.

exclude_links(*links: list*) → None

Excludes a list of links from a graph by setting their B node equal to their A node

Arguments

links (`list`): List of link IDs to be excluded from the graph

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*, *allow_uturns_everywhere: bool = False*) → None

Prepares the graph for a computation for a certain set of centroids.

Under the hood, if sets all centroids to have IDs from 1 through **n**, which should correspond to the index of the matrix being assigned.

This is what enables having any node IDs as centroids, and it relies on the inference that all links connected to these nodes are centroid connectors.

Arguments

centroids (`np.ndarray` or `None`, optional): Array with centroid IDs. Mandatory type `Int64`, unique and positive.

remove_dead_ends (`bool`, optional): Whether or not to remove dead ends from the graph.
Defaults to `True`.

allow_uturns_everywhere (`bool`, optional): Allow U-turns at all possible places within the
graph. This will allow U-turns anywhere that has both an incoming and outgoing link, severely hindering the effectiveness of graph compression. This is option should rarely be used. **The U-turns allowed by this option will never be taken unless turn penalties are set**. Refer to `graph.set_turn_restrictions` to allow U-turns at intersections. This option will reduce the effectiveness of dead-end link removal and graph compression.

`reverse`()

`save_compressed_correspondence`(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

`set_blocked_centroid_flows`(*block_centroid_flows*) → None

Chooses whether paths are allowed to pass through centroid connector turns.

With explicit turn restrictions, turn restrictions between centroid connectors are inserted.

Default value is `True`.

Arguments
block_centroid_flows (`bool`): Whether to block flows through centroids.

`set_graph`(*cost_field*) → None

Sets the field to be used for path computation

Arguments
cost_field (`str`): Field name. Must be numeric

`set_skimming`(*skim_fields: list*) → None

Sets the list of skims to be computed

Skimming with A* may produce results that differ from traditional Dijkstra's due to its use a heuristic.

Arguments
skim_fields (`list`): Fields must be numeric

`set_skimming_fields`(*skimming_fields: list[str] | None = None*) → None

Set the fields to skim and rebuild the hyperpath context if needed.

`set_turn_penalty_dimension`(*unit: str*)

Sets the unit for turn penalties.

`set_turn_restrictions`(*turn_restrictions: DataFrame*, *allow_path_uturns: bool = False*) → None

Sets turn restrictions for the graph. When turn restrictions are set, the graph will use arc-based Dijkstra for path computation.

Turn penalties are in the same time unit as the graph's cost field. Prepared graphs are rebuilt using their existing preparation settings, and their cost and skim fields are restored.

Arguments

turn_restrictions (`pd.DataFrame`): DataFrame with columns:

- from_node: incoming leg origin node
- via_node: turn node
- to_node: outgoing leg destination node
- penalty: turn penalty in time units

`None`, `NaN` and `+inf` are treated as prohibited turns. Negative values are not allowed.

Restrictions are interpreted as directed movement sequences `from_node -> via_node -> to_node`. All three nodes and both directed legs must exist in the full graph.

allow_path_uturns (`bool`): Whether U-turns are allowed within paths.
U-turns are transitions that return to the same node immediately before the turn. Default is `False`.

property allow_path_uturns: bool

Returns the current allow path U-turn permission setting.

property allow_uturns_everywhere: bool

Returns the allow U-turns everywhere setting that was used to build the graph.

property config

property has_turn_restrictions: bool

Returns True if the graph has turn restrictions loaded and active.

11.7 aequilibrae.route_choice

Classes

RouteChoice(graph[, project, coordinates])	
SubAreaAnalysis(graph, subarea, demand[, ...])	

class aequilibrae.route_choice.**RouteChoice**(*graph: Graph*, *project=None*, *, *coordinates: DataFrame | None = None*)

execute(*perform_assignment: bool = True*) → None

Generate route choice sets between the previously supplied nodes, potentially performing an assignment.

To access results see `RouteChoice.get_results()`.

Arguments
perform_assignment (`bool`): Whether or not to perform an assignment. Defaults to `False`.

execute_from_pandas(*df: DataFrame*, *recompute_psl: bool = False*) → None

Perform an assignment using route sets from a Pandas DataFrame.

Requires the DataFrame contains the `origin id`, `destination id` and `route set` columns. The route sets must be lists of unsigned link IDs, with matching +1/-1 arrays in `route directions`. Additionally, when `recompute_psl` is `False`, the `probability` column must also be present.

When `recompute_psl` is `True`, the path-sized logit is recomputed for each route with respect to the graph's cost field and the `beta`, `disutility_cutoff_constant` and `disutility_cutoff_coefficient` parameters. With PSL recomputation, the supplied mask is combined with the cost-cutoff mask. Otherwise, supplied costs and masks are kept. Invalid routes wil raise ValueError, including masked routes, regardless

of PSL recomputation. This includes empty paths, incomplete graph-link expansions, wrong OD endpoints, disconnected links, prohibited turns, disallowed U-turns, blocked centroid flow and infinite costs. Masked routes are not loaded regardless of supplied probabilities. Without recomputation, unmasked probabilities are not changed. Links absent from the graph or compact graph raise an error.

All origin and destination IDs within the DataFrame must exist within the demand matrix.

All link IDs and directions must exist within the graph. Links must also be present within the compressed graph.

The table returned from `self.get_results()` includes recalculated costs and a combined mask when PSL is recomputed. Otherwise, missing costs and path overlap default to zero and masked probabilities are zero.

execute_from_path_files(*path_files: Path | str*, *recompute_psl: bool = False*) → None

Perform an assignment from an existing set of path-files.

This method expects the path-files to be written by the `self.save_path_files()` method, however any parquet hive dataset with the correct structure is accepted. This allows the use of AequilibraE's path-sized logit, link loading, select link analysis, and assignment while using externally generated routes. With `recompute_psl=True`, costs (including turn penalties), probabilities and overlap are recomputed. Otherwise, supplied results are used. Invalid routes always raise ValueError, including masked routes.

execute_single(*origin: int*, *destination: int*, *demand: float = 0.0*) → list[tuple[ndarray, ndarray]]

Generate route choice sets between origin and destination, potentially performing an assignment.

Does not require preparation.

Node IDs must be present in the compressed graph. To make a node ID always appear in the compressed graph add it as a centroid.

Arguments

origin (`int`): Origin node ID.

destination (`int`): Destination node ID.

demand (`float`): If provided an assignment will be
performed with this demand. Non-zero demand requires centroid endpoints.

Returns

route set (`list[tuple[np.ndarray, np.ndarray]]`): A list of routes as ID/direction array pairs.

get_load_results() → DataFrame

Translates the link loading results from the graph format into the network format.

Returns

dataset (`Union[Tuple[pd.DataFrame, pd.DataFrame], pd.DataFrame]`): A tuple of link loading results as DataFrames. Columns are the matrix name concatenated direction.

get_results() → DataFrame

Returns the results of the route choice procedure

Returns a table of OD pairs to lists of link IDs for each OD pair provided (as columns). Represents paths from `origin` to `destination`. Matching `route directions` arrays contain +1 for the ab direction and -1 for the ba direction.

Returns

results (`pd.DataFrame`): Table with the results of the route choice procedure

get_select_link_loading_results() → DataFrame

Get the select link loading results.

Returns

dataset (Tuple[pd.DataFrame, pd.DataFrame]): Select link loading results as DataFrames. Columns are the matrix name concatenated with the select link set and direction.

get_select_link_od_matrix_results() → Dict[str, *AequilibraeSparseMatrix*]

Get the select link OD matrix results as a sparse matrix.

Returns

select link OD matrix results (:obj:`Dict[str,
AequilibraeSparseMatrix]`): Returns a dict of select link set names to labelled sparse OD matrices.

info() → dict

Returns information for the transit assignment procedure

Dictionary contains keys:

- Algorithm,
- Matrix totals
- Computer name
- Procedure ID
- Parameters
- Select links

The classes key is also a dictionary with all the user classes per transit class and their respective matrix totals.

Returns

info (dict): Dictionary with summary information

log_specification()

prepare(*nodes: List[int] | List[Tuple[int, int]] | None = None*) → None

Prepare OD pairs for batch computation.

Arguments

nodes (Union[list[int], list[tuple[int, int]]]): List of node IDs to operate on. If a 1D list is provided, OD pairs are taken to be all pair permutations of the list. If a list of pairs is provided OD pairs are taken as is. All node IDs must be present in the compressed graph. To make a node ID always appear in the compressed graph add it as a centroid. Duplicates will be dropped on execution. If None is provided, all OD pairs with non-zero flows will be used.

recompute_psl(*df: DataFrame*) → DataFrame

Validate and recost routes, then return their PSL results by OD.

Invalid routes raise ValueError, including masked routes.

save_link_flows(*table_name: str*, *project=None*) → None

Saves the link link flows for all classes into the results database.

Arguments

table_name (str): Name of the table being inserted to.

project (Project, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

save_path_files(*where: Path | None = None*, ***to_parquet_kwargs*)

Save path-files to the directory specific.

Files will be saved as a parquet hive dataset partitioned by the origin ID. Existing path-files will not be removed to allow incremental route choice set generation.

Arguments

where (`Optional[pathlib.Path]`): Directory to save the dataset to. **to_parquet_kwargs** (`dict`): Keyword arguments to supply to the underlying `to_parquet` call.

save_select_link(*name: str*, *project=None*) → None

Save both select link flows and OD matrices using the same name.

save_select_link_flows(*table_name: str*, *project=None*) → None

Saves the select link link flows into the results database.

Arguments

table_name (`str`): Name of the table being inserted to.

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

save_select_link_matrices(*matrix_name: str*, *project=None*) → None

Save the select link OD matrices to a new OMX file in the project matrices folder.

set_choice_set_generation(*algorithm: str = None*, ***kwargs*) → None

Chooses the assignment algorithm and set its parameters.

Options for algorithm are 'bfsle' for breadth first search with link removal, or 'link-penalisation'/'link-penalization'. 'lp' is also accepted as an alternative to 'link-penalisation'. If `algorithm` is `None`, none will be set, but the parameters will be updated. This is useful when assigning from path-files.

BFSLE implementation based on "Route choice sets for very high-resolution data" by Nadine Rieser-Schüssler, Michael Balmer & Kay W. Axhausen (2013). DOI: 10.1080/18128602.2012.671383.

Setting the parameters for the route choice:

- `seed` is a BFSLE specific parameters.
- Although not required, setting `max_depth` or `max_misses`, is strongly recommended to prevent run-away algorithms.
- `max_misses` is the maximum amount of duplicate routes found per OD pair. If a set of routes is returned in a case where `max_misses` is exceeded, the number of routes may be fewer than `max_routes`. Assumes a default value of 100.
- When using **BFSLE** `max_depth` corresponds to the maximum height of the graph. It's value is largely dependent on the size of the paths within the network. For very small networks a value of 10 is a recommended starting point. For large networks a good starting value is 5. Increase the value until the number of desired routes is being consistently returned. If a set of routes is returned in a case where `max_depth` is exceeded, the number of routes may be fewer than `max_routes`.
- When using **LP**, `max_depth` corresponds to the maximum number of iterations performed. While not enforced, it should be higher than `max_routes`. It's value is dependent on the magnitude of the cost field, specifically if it's related to the log base `penalty` of the ratio of costs between two alternative routes. If a set of routes is returned in a case where `max_depth` is exceeded, the number of routes may be fewer than `max_routes`.
- Additionally BFSLE has the option to incorporate link penalisation. Every link in all routes found at a depth are penalised with the penalty factor for the next depth. So at a depth of 0 no links are penalised nor removed. At depth 1, all links found at depth 0 are penalised, then the links marked for removal are

removed. All links in the routes found at depth 1 are then penalised for the next depth. The penalisation compounds. Set `penalty=1.0` to disable. For both algorithms, `penalty` must be finite and at least 1.

- Set `a_star=True` to use A* instead of Dijkstra. `heuristic` accepts `euclidean` (default) or `haversine`. Euclidean distance uses the constructor's `coordinates`; Haversine uses the constructor's `lon` and `lat` coordinates in degrees.
- A* requires an finite, nonnegative `heuristic_scale` to convert distance to cost units. Use `estimate_heuristic_scale` to calculate a conservative value. A scale above a consistent bound can give non-shortest paths and change the generated route sets. A consistent bound remains valid as links are removed or penalised.
- When performing an assignment, `disutility_cutoff_constant` and `disutility_cutoff_coefficient` can be provided to exclude routes from the path-sized logit model. Any route with a cost higher when applied to this function `disutility_cutoff_coefficient * cost + disutility_cutoff_constant` within the route set is masked out.

Arguments

algorithm (`str`): Algorithm to be used

kwargs (`dict`): Dictionary with all parameters for the algorithm

set_cores(*cores: int*) → None

Allows one to set the number of cores to be used

Inherited from `AssignmentResultsBase`

Arguments

cores (`int`): Number of CPU cores to use

set_demand(*demand:* AequilibraeMatrix | AequilibraeSparseMatrix | *None*) → None

Set the demand matrix for the assignment.

Arguments

demand (`AequilibraeMatrix` or `AequilibraeSparseMatrix`, *Optional*): Demand for assignment. Node IDs are inferred from the index and must be graph centroids. Use *`aequilibrae.matrix.from_df()`* to convert a DataFrame.

set_save_routes(*where: str | None = None*, *to_parquet_kwargs: dict[str, Any] | None = None*) → None

Set save path for route choice results. Provide `None` to disable.

Arguments

where (`Optional[pathlib.Path]`): Directory to save the dataset to. **to_parquet_kwargs** (`dict`): Keyword arguments to supply to the underlying `to_parquet` call.

set_select_links(*links: Dict[Hashable, List[Tuple[int, int] | List[Tuple[int, int]]]]*, *link_loading=True*)

Set the selected links. Checks if the links and directions are valid. Supports **OR** and **AND** sets of links.

Dictionary values should be a list of either a single `(link_id, direction)` tuple or a list of `(link_id, direction)`.

The elements of the first list represent the **AND** sets, together they are OR'ed. If any of these sets is satisfied the link are loaded as appropriate.

The **AND** sets are comprised of either a single `(link_id, direction)` tuple or a list of `(link_id, direction)`. The single tuple represents an **AND** set with a single element.

All links and directions in an **AND** set must appear in any order within a route for it to be considered satisfied.

Supply `links=None` to disable select link analysis.

Arguments

links (`Union[None, Dict[Hashable, List[Union[Tuple[int, int], List[Tuple[int, int]]]]]]`): Name of link set and link IDs and directions to be used in select link analysis.

link_loading (`bool`): Enable select link loading. If disabled only OD matrix results are available.

```
all_algorithms = ['bfsle', 'lp', 'link-penalisation', 'link-penalization']

default_parameters = {'bfsle': {'penalty': 1.0}, 'generic': {'a_star': False,
'beta': 1.0, 'disutility_cutoff_coefficient': inf, 'disutility_cutoff_constant':
inf, 'heuristic': 'euclidean', 'heuristic_scale': None, 'max_depth': 0,
'max_misses': 100, 'max_routes': 0, 'penalty': 1.01, 'seed': 0, 'store_results':
True}, 'link-penalisation': {}}
```

class aequilibrae.route_choice.**SubAreaAnalysis**(*graph:* Graph, *subarea: GeoDataFrame*, *demand:* AequilibraeMatrix | AequilibraeSparseMatrix, *project=None*)

post_process(*demand_cols=None*, *keep_original_ods: bool = False*)

Apply the necessary post processing to the route choice assignment select link results.

Arguments

demand_cols (`[list[str]]`, optional): If provided, only construct the sub-area matrix for these demand matrices.

keep_original_ods (`bool`, optional): If provided, the original origin and destination IDs for the demand will be kept. This will create a significantly larger demand matrix but is more flexible.

Returns

sub_area_demand (`pd.DataFrame`): A DataFrame representing the sub-area demand matrix.

11.8 aequilibrae.distribution

Classes

GravityApplication([project])	Applies a synthetic gravity model.
GravityCalibration([project])	Calibrate a traditional gravity model
Ipf([project])	Iterative proportional fitting procedure
SyntheticGravityModel()	Simple class object to represent synthetic gravity models

class aequilibrae.distribution.**GravityApplication**(*project=None*, ***kwargs*)

Applies a synthetic gravity model.

Model is an instance of SyntheticGravityModel class.

Impedance is an instance of AequilibraeMatrix.

Vectors are a pandas DataFrame.

```
>>> import pandas as pd
>>> from aequilibrae.distribution import SyntheticGravityModel, GravityApplication

>>> project = create_example(project_path)

# We define the model we will use
>>> model = SyntheticGravityModel()

# Before adding a parameter to the model, you need to define the model functional␣
↪form
# You can select one of GAMMA, EXPO or POWER.
>>> model.function = "GAMMA"

# Only the parameter(s) applicable to the chosen functional form will have any␣
↪effect
>>> model.alpha = 0.1
>>> model.beta = 0.0001

# We load the impedance matrix
>>> matrix = project.matrices.get_matrix("skims.omx", subset=["distance_blended"])

# We create the vectors we will use
>>> query = "SELECT zone_id, population, employment FROM zones;"
>>> with project.db_connection as conn:
...     df = pd.read_sql(query, conn)
>>> df.sort_values(by="zone_id", inplace=True)
>>> df.set_index("zone_id", inplace=True)

# You create the vectors you would have
>>> df = df.assign(productions=df.population * 3.0)
>>> df = df.assign(attractions=df.employment * 4.0)
>>> vectors = df[["productions", "attractions"]]

# Balance the vectors
>>> vectors["attractions"] *= vectors["productions"].sum() / vectors["attractions
↪"].sum()

# Create the problem object
>>> args = {"impedance": matrix,
...         "vectors": vectors,
...         "row_field": "productions",
...         "model": model,
...         "column_field": "attractions",
...         "nan_as_zero":True
...         }
>>> gravity = GravityApplication(**args)

# Solve and save the outputs
>>> gravity.apply()
>>> gravity.output.save_as_omx(project_path / 'matrices' / 'gravity_omx.omx')

>>> project.close()
```

apply()

Runs the Gravity Application instance as instantiated

Resulting matrix is the *output* class member

save_to_project (*file_name: str*, *project=None*) → None

Saves the matrix output to the project file

Arguments

file_name (`str`): Name for the OMX matrix file

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

class aequilibrae.distribution.**GravityCalibration** (*project=None*, ***kwargs*)

Calibrate a traditional gravity model

Available deterrence function forms are: 'EXPO', 'POWER' or 'GAMMA'.

```
>>> from aequilibrae.distribution import GravityCalibration

>>> project = create_example(project_path)

# We load the demand matrix
>>> demand = project.matrices.get_matrix("demand.omx", subset=["matrix"])

# We load the skim matrix
>>> skim = project.matrices.get_matrix("skims.omx", subset=["time_final"])

>>> args = {"matrix": demand,
...         "impedance": skim,
...         "row_field": "productions",
...         "function": 'expo',
...         "nan_as_zero": True}
>>> gravity = GravityCalibration(**args)

# Solve and save outputs
>>> gravity.calibrate()
>>> gravity.model.save(project_path / 'dist_expo_model.mod')

>>> project.close()
```

calibrate()

Calibrate the model

Resulting model is in *output* class member

class aequilibrae.distribution.**Ipf** (*project=None*, ***kwargs*)

Iterative proportional fitting procedure

```
>>> from aequilibrae.distribution import Ipf

>>> project = create_example(project_path)

>>> matrix = project.matrices.get_matrix("demand.omx", subset=["matrix"])

>>> vectors = pd.DataFrame(
```

(continues on next page)

(continued from previous page)

```
...     {"productions": np.zeros(len(matrix.index)), "attractions": np.
↪zeros(len(matrix.index))},
...     index=matrix.index,
... )

>>> core = next(iter(matrix))
>>> vectors["productions"] = matrix[core].sum(axis=1)
>>> vectors["attractions"] = matrix[core].sum(axis=0)

>>> ipf_args = {"matrix": matrix,
...             "vectors": vectors,
...             "row_field": "productions",
...             "column_field": "attractions",
...             "nan_as_zero": False}

>>> fratar = Ipf(**ipf_args)
>>> fratar.fit()

# We can get back to our OMX matrix in the end
>>> fratar.output.save_as_omx(Path(my_folder_path) / "to_omx_output.omx")

>>> project.close()
```

fit()

Runs the IPF instance problem to adjust the matrix

Resulting matrix is the *output* class member

save_to_project(*file_name: str*, *project=None*) → None

Saves the matrix output to the project file

Arguments

file_name (`str`): Name for the OMX matrix file

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the explicitly supplied constructor project

class aequilibrae.distribution.**SyntheticGravityModel**

Simple class object to represent synthetic gravity models

load(*file_name*)

Loads model from disk. Extension is *.mod

save(*file_name*)

Saves model to disk in yaml format. Extension is *.mod

11.9 aequilibrae.matrix

Functions

from_df(...)	Create dense or sparse in-memory matrices from an origin-destination DataFrame.

continues on next page

Table 11 – continued from previous page

from_dict(...)	Create named square matrices from a mapping of names to array-like values.
from_file(...)	Load an OMX file as an in-memory matrix or an on-disk store.

Classes

AequilibraeMatrix(index[, index_name])	A dictionary-like collection of square matrices held in memory.
AequilibraeSparseMatrix(index[, index_name])	A dictionary-like collection of square SciPy sparse arrays held in memory.
MatrixStore(path[, index, index_name, mode])	A dictionary-like collection of square matrices backed by an OMX file.

class aequilibrae.matrix.**AequilibraeMatrix**(*index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str], index_name: str = 'main_index'*)

A dictionary-like collection of square matrices held in memory.

All matrices use the same zone index for rows and columns.

Assigning a whole matrix copies the values and keeps their dtype. Reading a matrix or taking a basic slice returns a view of its data. Use `.copy()` when you need an independent array. You can also use dictionary operations such as `keys()`, `items()` and `update()`.

Matrix and index names must be non-empty strings, not paths. They cannot be `"."` or `".."`, or contain a slash, backslash or null character.

```
>>> demand = AequilibraeMatrix(index=[101, 205], index_name="zones")
>>> demand["car"] = [[0, 12], [10, 0]]
>>> demand["bus"] = [[0, 2], [2, 0]]
>>> demand["total"] = demand["car"] + demand["bus"]
>>> list(demand)
['car', 'bus', 'total']
>>> demand["total"].tolist()
[[0, 14], [12, 0]]
>>> del demand["total"]
>>> len(demand), demand.shape
(2, (2, 2))
```

classmethod from_omx(*path: str | PathLike, index_name: str = 'main_index', subset: Sequence[str | dict[str, Any]] | None = None, **kwargs*) → Self

Load matrices from an OMX file into memory.

The file must contain the mapping named by `index_name`. By default, all matrices are loaded. Pass `subset` to choose matrices by name or by their attributes. A query matches matrices with all the given attributes, and each matching matrix is loaded once.

Matrix values, the zone index and metadata are copied. Changing the loaded object does not change the file.

Arguments

path (`str` or `os.PathLike`): Existing OMX file to load.

index_name (`str`, *Optional*): Existing zone mapping to read. Defaults to `"main_index"`.

subset (sequence of `str` or `dict`, *Optional*): Matrix names or attribute queries to load. Defaults to all matrices.

kwargs: Additional keyword arguments passed to the matrix constructor.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> demand = from_dict({"car": [[12]], "bus": [[2]]}, index=[101])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
...     demand.save_as_omx(path)
...     working = AequilibraeMatrix.from_omx(path, subset=[{"mode": "car"}])
>>> list(working)
['car']
>>> working["car"].tolist()
[[12]]
```

`clear`() → None

Remove all named matrices and their attributes.

The zone index and file-level metadata are kept.

```
>>> demand = from_dict({"car": [[0]]}, index=[101])
>>> demand.metadata["year"] = 2030
>>> demand.clear()
>>> list(demand), demand.index.tolist(), demand.metadata["year"]
([], [101], 2030)
```

`get`(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

`items`() → a set-like object providing a view on D's items

`keys`() → a set-like object providing a view on D's keys

`matrix_metadata`(*name: str*) → MutableMapping[str, Any]

Return the mutable attribute dictionary for an existing named matrix.

Assign a new value to an attribute to update it. Attributes are kept when you replace a matrix and can be used to select matrices with `subset()`.

Arguments

name (`str`): Name of an existing matrix.

```
>>> demand = from_dict({"car": [[12]]}, index=[101])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> demand["car"] = [[15.0]]
>>> demand.matrix_metadata("car")["mode"]
'car'
```

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

save_as_omx(*path: str | PathLike*, *mode: Literal['w', 'a', 'r+', 'w-', 'x'] = 'x'*, ***, *subset: Sequence[str | dict[str, Any]] | None = None*, ***kwargs*) → None

Save named matrices, their attributes, the zone index and file metadata to OMX.

By default, this saves all matrices to a new file and raises an error if the file already exists. Pass `subset` to select matrices by name or by their attributes. A query matches matrices with all the given attributes. Each matrix is saved once, even if it matches several entries.

When updating a file, matrices outside the subset are left unchanged. Attributes provided here replace matching attributes in the file. Other attributes are kept. The zone mapping and file metadata are always written, apart from attributes managed by OMX.

Arguments

path (`str` or `os.PathLike`): Destination OMX file.

mode (`str`, *Optional*): Use `"x"` or `"w-"` to
create a new file, `"w"` to replace or create a file, `"r+"` to update an existing file, or `"a"` to update or create a file. Defaults to `"x"`.

subset (sequence of `str` or `dict`, *Optional*):
Matrix names or attribute queries to save. Defaults to all matrices. If nothing matches, only the zone mapping and file metadata are written.

kwargs: Additional keyword arguments passed to
`openmatrix.open_file`.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
...     demand.save_as_omx(path, subset=[{"mode": "car"}])
...     loaded = from_file(path)
>>> loaded == demand
True
```

setdefault(*key: str*, *default: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | bsr_array | coo_array | csc_array | csr_array | dok_array | lil_array | None = None*) → _ArrayT

Return a named matrix, adding it from `default` if it is missing.

An existing matrix is returned unchanged. If the name is missing, `default` must contain a full square matrix. `None` cannot be used to create a matrix.

As with a normal lookup, the result is a view for a dense in-memory matrix, the stored array for a sparse matrix and a copy for a store.

Arguments

key (`str`): Matrix name to look up or insert.

default (array-like, *Optional*): Full square matrix to insert
if the name is missing. Defaults to `None`, which is only valid when the matrix already exists.

```
>>> demand = AequilibraeMatrix(index=[101])
>>> demand.setdefault("car", [[12]]).tolist()
[[12]]
>>> demand.setdefault("car", [[99]]).tolist()
[[12]]
```

subset(*subset: Sequence[str | dict[str, Any]]*) → Self

Copy selected matrices into a new in-memory matrix object.

Select matrices by name or by their attributes. A query matches matrices with all the given attributes. Each matrix is included once, even if it matches more than one entry.

The arrays, zone index and metadata are copied. Changes to the result do not affect the original object.

Arguments

subset (sequence of `str` or `dict`): Matrix names or
attribute queries to copy.

```
>>> demand = from_dict({"car": [[12]], "bus": [[2]]}, index=[101])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> working = demand.subset([{"mode": "car"}, "car"])
>>> list(working)
['car']
>>> working["car"] *= 2
>>> demand["car"].tolist(), working["car"].tolist()
([[12]], [[24]])
```

totals() → dict[str, Any]

Return each named matrix's total.

update(*[E,]**F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property imat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zero-based positions.

Use `imat[name]` for a whole matrix, `imat[name, row]` for rows, or `imat[name, row, column]` to select both rows and columns. Slices exclude the stop position.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.imat["car", :, 1].tolist()
[12, 0]
>>> demand.imat["car", 0, :] = [1, 2]
>>> demand.imat["car", :1].tolist()
[[1, 2]]
```

property index: ndarray

Zone IDs in row and column order, returned as a read-only uint64 array.

IDs must be unique integers between `0` and `2**64 - 1`. To relabel zones, assign a new index of the same length. This does not move any matrix values.

Changing a store's index requires write access. The mapping keeps its existing dtype and attributes.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.index.tolist()
[101, 205]
>>> demand.index.flags.writeable
False
>>> demand.index = [1001, 1002]
>>> int(demand.mat["car", 1001, 1002])
12
```

property index_name: str

Name of the zone mapping used when reading and writing OMX files.

```
>>> AequilibraeMatrix(index=[101, 205]).index_name
'main_index'
>>> AequilibraeMatrix(index=[101, 205], index_name="zones").index_name
'zones'
```

property mat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zone IDs.

Use `mat[name]` for a whole matrix, `mat[name, row]` for rows, or `mat[name, row, column]` to select both rows and columns. Slices follow the stored zone order and include the stop ID. An unknown zone ID raises `KeyError`.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.mat["car", 101:205].tolist()
[[0, 12], [10, 0]]
>>> demand.mat["car", 101, 205] = 42
>>> int(demand.mat["car", 101, 205])
42
```

property metadata: MutableMapping[str, Any]

File-level attributes that can be read and updated by name.

These are stored in a dictionary for in-memory matrices and in h5py attributes for stores. Assign a new value to an attribute to update it.

Attributes managed by OMX are visible on a store but are ignored when loading, copying, saving or comparing file metadata.

```
>>> demand = AequilibraeMatrix(index=[101, 205])
>>> demand.metadata["description"] = "Morning demand"
>>> demand.metadata["description"]
'Morning demand'
```

property shape: tuple[int, int]

Shape of each matrix, determined by the number of zone IDs.

```
>>> demand = AequilibraeMatrix(index=[101, 205, 309])
>>> demand.shape
(3, 3)
>>> len(demand)
0
```

class aequilibrae.matrix.**AequilibraeSparseMatrix**(*index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str], index_name: str = 'main_index'*)

A dictionary-like collection of square SciPy sparse arrays held in memory.

All matrices use the same zone index for rows and columns. Assigning a matrix copies the array and keeps its sparse format and dtype. Reading a whole matrix returns the stored array, so changes to it change the matrix. Use `.copy()` when you need an independent array. Selections through `mat` and `imat` follow the sparse format's own SciPy indexing rules.

Saving writes each array in the CSR format. This extends the OMX format with a `/sparse/<name>` group for each matrix. The group has a `row_index` (CSR row pointers), `col_index` (column positions) and `data` datasets. The group attribute `FORMAT` is set to `"csr"`. Zone mappings, file metadata, and shape use the standard OMX attributes.

When reading, only `/sparse` is read from, dense matrices in `/data` are ignored. Use *MatrixStore* or **:method:`AequilibraeMatrix.from_omx()`** instead.

Matrix and index names follow the same rules as *AequilibraeMatrix*.

```
>>> from scipy.sparse import coo_array
>>> demand = AequilibraeSparseMatrix(index=[101, 205])
>>> demand["car"] = coo_array([[0, 12], [10, 0]])
>>> demand["car"].format
'coo'
>>> demand["car"].toarray().tolist()
[[0, 12], [10, 0]]
```

classmethod from_omx(*path: str | PathLike, index_name: str = 'main_index', subset: Sequence[str | dict[str, Any]] | None = None, **kwargs*) → Self

Load CSR arrays from the OMX extension group `/sparse` into memory.

Dense matrices in `/data` are ignored. Values, the zone index and metadata are copied. Changes to the result do not change the file.

Arguments

path (`str` or `os.PathLike`): Existing OMX file.

index_name (`str`, *Optional*): Existing zone mapping to read. Defaults to `"main_index"`.

subset (sequence of `str` or `dict`, *Optional*): Matrix names or attribute queries. Defaults to all sparse matrices. Each matching matrix is loaded once.

kwargs: Additional keyword arguments passed to the constructor.

clear() → None

Remove all named matrices and their attributes.

The zone index and file-level metadata are kept.

```
>>> demand = from_dict({"car": [[0]]}, index=[101])
>>> demand.metadata["year"] = 2030
>>> demand.clear()
>>> list(demand), demand.index.tolist(), demand.metadata["year"]
([], [101], 2030)
```

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

matrix_metadata(*name: str*) → MutableMapping[str, Any]

Return mutable attributes for an existing named sparse matrix.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

save_as_omx(*path: str | PathLike*, *mode: Literal['w', 'a', 'r+', 'w-', 'x'] = 'x'*, ***, *subset: Sequence[str | dict[str, Any]] | None = None*, ***kwargs*) → None

Save named matrices, their attributes, the zone index and file metadata to OMX.

By default, this saves all matrices to a new file and raises an error if the file already exists. Pass `subset` to select matrices by name or by their attributes. A query matches matrices with all the given attributes. Each matrix is saved once, even if it matches several entries.

When updating a file, matrices outside the subset are left unchanged. Attributes provided here replace matching attributes in the file. Other attributes are kept. The zone mapping and file metadata are always written, apart from attributes managed by OMX.

Arguments

path (`str` or `os.PathLike`): Destination OMX file.

mode (`str`, *Optional*): Use `"x"` or `"w-"` to
create a new file, `"w"` to replace or create a file, `"r+"` to update an existing file, or `"a"` to update or create a file. Defaults to `"x"`.

subset (sequence of `str` or `dict`, *Optional*):
Matrix names or attribute queries to save. Defaults to all matrices. If nothing matches, only the zone mapping and file metadata are written.

kwargs: Additional keyword arguments passed to
`openmatrix.open_file`.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
```

(continues on next page)

(continued from previous page)

```
...     demand.save_as_omx(path, subset=[{"mode": "car"}])
...     loaded = from_file(path)
>>> loaded == demand
True
```

setdefault(*key: str*, *default: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | bsr_array | coo_array | csc_array | csr_array | dok_array | lil_array | None = None*) → _ArrayT

Return a named matrix, adding it from `default` if it is missing.

An existing matrix is returned unchanged. If the name is missing, `default` must contain a full square matrix. `None` cannot be used to create a matrix.

As with a normal lookup, the result is a view for a dense in-memory matrix, the stored array for a sparse matrix and a copy for a store.

Arguments

key (`str`): Matrix name to look up or insert.

default (array-like, *Optional*): Full square matrix to insert
if the name is missing. Defaults to `None`, which is only valid when the matrix already exists.

```
>>> demand = AequilibraeMatrix(index=[101])
>>> demand.setdefault("car", [[12]]).tolist()
[[12]]
>>> demand.setdefault("car", [[99]]).tolist()
[[12]]
```

subset(*subset: Sequence[str | dict[str, Any]]*) → Self

Copy selected sparse matrices, their zone index and metadata.

Select by matrix name or attribute query. Each matching matrix is included once. Changes to the result do not change the original. Sparse formats and dtypes are kept.

totals() → dict[str, Any]

Return each named matrix's total.

update(*[E,]**F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property imat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zero-based positions.

Use `imat[name]` for a whole matrix, `imat[name, row]` for rows, or `imat[name, row, column]` to select both rows and columns. Slices exclude the stop position.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.imat["car", :, 1].tolist()
[12, 0]
>>> demand.imat["car", 0, :] = [1, 2]
>>> demand.imat["car", :1].tolist()
[[1, 2]]
```

property index: ndarray

Zone IDs in row and column order, returned as a read-only uint64 array.

IDs must be unique integers between `0` and `2**64 - 1`. To relabel zones, assign a new index of the same length. This does not move any matrix values.

Changing a store's index requires write access. The mapping keeps its existing dtype and attributes.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.index.tolist()
[101, 205]
>>> demand.index.flags.writeable
False
>>> demand.index = [1001, 1002]
>>> int(demand.mat["car", 1001, 1002])
12
```

property index_name: str

Name of the zone mapping used when reading and writing OMX files.

```
>>> AequilibraeMatrix(index=[101, 205]).index_name
'main_index'
>>> AequilibraeMatrix(index=[101, 205], index_name="zones").index_name
'zones'
```

property mat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zone IDs.

Use `mat[name]` for a whole matrix, `mat[name, row]` for rows, or `mat[name, row, column]` to select both rows and columns. Slices follow the stored zone order and include the stop ID. An unknown zone ID raises `KeyError`.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.mat["car", 101:205].tolist()
[[0, 12], [10, 0]]
>>> demand.mat["car", 101, 205] = 42
>>> int(demand.mat["car", 101, 205])
42
```

property metadata: MutableMapping[str, Any]

File-level attributes that can be read and updated by name.

These are stored in a dictionary for in-memory matrices and in h5py attributes for stores. Assign a new value to an attribute to update it.

Attributes managed by OMX are visible on a store but are ignored when loading, copying, saving or comparing file metadata.

```
>>> demand = AequilibraeMatrix(index=[101, 205])
>>> demand.metadata["description"] = "Morning demand"
>>> demand.metadata["description"]
'Morning demand'
```

property shape: tuple[int, int]

Shape of each matrix, determined by the number of zone IDs.

```
>>> demand = AequilibraeMatrix(index=[101, 205, 309])
>>> demand.shape
(3, 3)
>>> len(demand)
0
```

class aequilibrae.matrix.**MatrixStore**(*path: str | PathLike*, *index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None*, *index_name: str = 'main_index'*, *mode: Literal['r', 'w', 'a', 'r+', 'w-', 'x'] = 'r'*, ***kwargs*)

A dictionary-like collection of square matrices backed by an OMX file.

Open an existing file by choosing a zone mapping with `index_name`. To create a file or relabel its zones, supply `index` and a writable mode. Stores are read-only by default.

Each read loads values from disk into a new array. Changing that array does not update the file. To write changes, assign a whole matrix by name or use `mat` or `imat` to update part of a dense matrix.

Sparse matrices use the CSR extension in `/sparse` and share the selected zone mapping with dense matrices in `/data`. Each sparse read returns an independent CSR array. Partial reads and writes through `mat` or `imat` are not supported, even for selections covering all values. Whole-matrix access through `mat[name]` and `imat[name]` is supported.

Use `subset()` to copy selected matrices of one type into memory for calculations. Keep the store open while using it. Using a `with` block is recommended. Delete a matrix before changing its storage type.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
...     with MatrixStore(path, index=[101, 205], mode="x") as store:
...         store["car"] = [[0, 12], [10, 0]]
...         values = store["car"]
...         values[0, 1] = 99
...         print(int(store.imat["car", 0, 1]))
...         store.mat["car", 101, 205] = 42
...         working = store.subset(["car"])
12
>>> working["car"].tolist()
[[0, 42], [10, 0]]
```

clear() → None

Remove all named matrices and their attributes.

The zone index and file-level metadata are kept.

```
>>> demand = from_dict({"car": [[0]]}, index=[101])
>>> demand.metadata["year"] = 2030
>>> demand.clear()
>>> list(demand), demand.index.tolist(), demand.metadata["year"]
([], [101], 2030)
```

close() → None

Close the OMX file.

This is called automatically when leaving a `with` block.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as directory:
...     store = MatrixStore(Path(directory) / "demand.omx", index=[101], mode=
↪"x")
...     store["car"] = [[12]]
...     working = store.subset(["car"])
...     store.close()
>>> working["car"].tolist()
[[12]]
```

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

matrix_metadata(*name: str*) → AttributeManager

Return h5py attributes for an existing matrix dataset or sparse group.

Use a writable store to change attributes, and assign each new value as a whole. Replacing a matrix with a different dtype creates a new dataset. Replacing a sparse matrix creates a new storage group. Call this method again afterwards to get its attributes. The sparse storage attribute `FORMAT` is ignored when writing.

Arguments

name (`str`): Name of an existing matrix dataset or sparse group.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as directory:
...     with MatrixStore(Path(directory) / "demand.omx", index=[101], mode="x
↪") as store:
...         store["car"] = [[12]]
...         store.matrix_metadata("car")["mode"] = "car"
...         print(store.matrix_metadata("car")["mode"])
car
```

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair

as a 2-tuple; but raise KeyError if D is empty.

save_as_omx(*path: str | PathLike*, *mode: Literal['w', 'a', 'r+', 'w-', 'x'] = 'x'*, ***, *subset: Sequence[str | dict[str, Any]] | None = None*, ***kwargs*) → None

Save named matrices, their attributes, the zone index and file metadata to OMX.

By default, this saves all matrices to a new file and raises an error if the file already exists. Pass `subset` to select matrices by name or by their attributes. A query matches matrices with all the given attributes. Each matrix is saved once, even if it matches several entries.

When updating a file, matrices outside the subset are left unchanged. Attributes provided here replace matching attributes in the file. Other attributes are kept. The zone mapping and file metadata are always written, apart from attributes managed by OMX.

Arguments

path (`str` or `os.PathLike`): Destination OMX file.

mode (`str`, *Optional*): Use `"x"` or `"w-"` to
create a new file, `"w"` to replace or create a file, `"r+"` to update an existing file, or `"a"` to update or create a file. Defaults to `"x"`.

subset (sequence of `str` or `dict`, *Optional*):
Matrix names or attribute queries to save. Defaults to all matrices. If nothing matches, only the zone mapping and file metadata are written.

kwargs: Additional keyword arguments passed to
`openmatrix.open_file`.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.matrix_metadata("car")["mode"] = "car"
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
...     demand.save_as_omx(path, subset=[{"mode": "car"}])
...     loaded = from_file(path)
>>> loaded == demand
True
```

`setdefault`(*key: str*, *default: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | bsr_array | coo_array | csc_array | csr_array | dok_array | lil_array | None = None*) → _ArrayT

Return a named matrix, adding it from `default` if it is missing.

An existing matrix is returned unchanged. If the name is missing, `default` must contain a full square matrix. `None` cannot be used to create a matrix.

As with a normal lookup, the result is a view for a dense in-memory matrix, the stored array for a sparse matrix and a copy for a store.

Arguments

key (`str`): Matrix name to look up or insert.

default (array-like, *Optional*): Full square matrix to insert
if the name is missing. Defaults to `None`, which is only valid when the matrix already exists.

```
>>> demand = AequilibraeMatrix(index=[101])
>>> demand.setdefault("car", [[12]]).tolist()
[[12]]
>>> demand.setdefault("car", [[99]]).tolist()
[[12]]
```

subset(*subset: Sequence[str | dict[str, Any]]*, *, *sparse: bool | None = None*) → *AequilibraeMatrix* | *AequilibraeSparseMatrix*

Copy selected matrices from the file into memory.

Select matrices by name or by their attributes. A query matches matrices with all the given attributes. Each matrix is included once, even if it matches more than one entry.

The zone index and metadata are also copied. You can keep using the result after the store closes, and changes to it do not affect the file. All selected matrices must have the same storage type.

Arguments

subset (sequence of `str` or `dict`): Matrix names or attribute queries to load.

sparse (`bool`, *Optional*): Expected storage type. Defaults to `None`, which infers the type from the first selected matrix.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as directory:
...     with MatrixStore(Path(directory) / "demand.omx", index=[101], mode="x
↪") as store:
...         store["car"] = [[12]]
...         store["bus"] = [[2]]
...         store.matrix_metadata("car")["mode"] = "car"
...         working = store.subset([{"mode": "car"}])
>>> list(working), working["car"].tolist()
(['car'], [[12]])
```

totals() → dict[str, Any]

Return each named matrix's total.

update([*E*,]**F*) → None. Update D from mapping/iterable E and F.

If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

property imat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zero-based positions.

Use `imat[name]` for a whole matrix, `imat[name, row]` for rows, or `imat[name, row, column]` to select both rows and columns. Slices exclude the stop position.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.imat["car", :, 1].tolist()
[12, 0]
>>> demand.imat["car", 0, :] = [1, 2]
>>> demand.imat["car", :1].tolist()
[[1, 2]]
```

property index: ndarray

Zone IDs in row and column order, returned as a read-only uint64 array.

IDs must be unique integers between `0` and `2**64 - 1`. To relabel zones, assign a new index of the same length. This does not move any matrix values.

Changing a store's index requires write access. The mapping keeps its existing dtype and attributes.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.index.tolist()
[101, 205]
>>> demand.index.flags.writeable
False
>>> demand.index = [1001, 1002]
>>> int(demand.mat["car", 1001, 1002])
12
```

property index_name: str

Name of the zone mapping used when reading and writing OMX files.

```
>>> AequilibraeMatrix(index=[101, 205]).index_name
'main_index'
>>> AequilibraeMatrix(index=[101, 205], index_name="zones").index_name
'zones'
```

property mat: _MatrixIndexer[_ArrayT]

Read or change matrix values using zone IDs.

Use `mat[name]` for a whole matrix, `mat[name, row]` for rows, or `mat[name, row, column]` to select both rows and columns. Slices follow the stored zone order and include the stop ID. An unknown zone ID raises `KeyError`.

Zone IDs are converted to positions before indexing. In-memory, dense matrices use NumPy selections while sparse matrices use SciPy selections. Stored dense matrices support the selections allowed by h5py. Stored sparse matrices only support whole-matrix access. Writing to a selection keeps the matrix's dtype. Values are broadcast and converted according to the array or dataset's rules.

```
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205])
>>> demand.mat["car", 101:205].tolist()
[[0, 12], [10, 0]]
>>> demand.mat["car", 101, 205] = 42
>>> int(demand.mat["car", 101, 205])
42
```

property metadata: MutableMapping[str, Any]

File-level attributes that can be read and updated by name.

These are stored in a dictionary for in-memory matrices and in h5py attributes for stores. Assign a new value to an attribute to update it.

Attributes managed by OMX are visible on a store but are ignored when loading, copying, saving or comparing file metadata.

```
>>> demand = AequilibraeMatrix(index=[101, 205])
>>> demand.metadata["description"] = "Morning demand"
>>> demand.metadata["description"]
'Morning demand'
```

property omx: File

Underlying open OMX file for advanced operations.

Changes to matrix storage groups through this handle are visible to the store without reopening it.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> with TemporaryDirectory() as directory:
...     with MatrixStore(Path(directory) / "demand.omx", index=[101], mode="x
↪") as store:
...         store["car"] = [[12]]
...         print(store.omx.list_matrices())
['car']
```

property shape: tuple[int, int]

Shape of each matrix, determined by the number of zone IDs.

```
>>> demand = AequilibraeMatrix(index=[101, 205, 309])
>>> demand.shape
(3, 3)
>>> len(demand)
0
```

aequilibrae.matrix.**from_df**(*df: DataFrame*, *row: str*, *col: str*, *subset: Sequence[str] | None = None*, ***, *index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None*, *index_name: str = 'main_index'*, *fill_value: float = 0*, *dtype: type[Any] | dtype[Any] | _HasDType[dtype[Any]] | _HasNumPyDType[dtype[Any]] | tuple[Any, Any] | list[Any] | _DTypeDict | str = float*, *sparse: Literal[False] = False*) → *AequilibraeMatrix*

aequilibrae.matrix.**from_df**(*df: DataFrame*, *row: str*, *col: str*, *subset: Sequence[str] | None = None*, ***, *index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str] | None = None*, *index_name: str = 'main_index'*, *fill_value: float = 0*, *dtype: type[Any] | dtype[Any] | _HasDType[dtype[Any]] | _HasNumPyDType[dtype[Any]] | tuple[Any, Any] | list[Any] | _DTypeDict | str = float*, *sparse: Literal[True]*) → *AequilibraeSparseMatrix*

Create dense or sparse in-memory matrices from an origin-destination DataFrame.

Choose the origin and destination columns with `row` and `col`. All other columns become a named matrix, or you can choose columns with `subset`. Values for repeated origin-destination pairs are summed. Missing pairs are filled with `fill_value`.

Pass `index` to set the zone order and include zones with no trips. If omitted, the index contains the sorted unique origin and destination IDs. All IDs must be valid integer zone IDs and appear in the index. Pass `sparse=True` to construct CSR arrays directly from the records without allocating dense matrices. Sparse matrices require `fill_value=0`.

Arguments

df (`pandas.DataFrame`): Origin-destination records with zone IDs and matrix value columns.

row (`str`): Column containing origin zone IDs.

col (`str`): Column containing destination zone IDs.

subset (sequence of `str`, *Optional*): Value columns to
convert to named matrices. Defaults to all columns except `row` and `col`.

index (array-like, *Optional*): Unique integer zone IDs shared by
rows and columns. Must include all origin and destination IDs. Defaults to their sorted union.

index_name (`str`, *Optional*): Zone mapping name. Defaults to
`"main_index"`.

fill_value (`float`, *Optional*): Value for pairs without
records. Defaults to `0`.

dtype (NumPy dtype-like, *Optional*): Dtype used for all output
matrices. Defaults to `float`.

sparse (`bool`, *Optional*): Return an
AequilibraeSparseMatrix instead of an *AequilibraeMatrix*. Defaults to `False`. Requires `fill_value=0`.

```
>>> trips = pd.DataFrame({"origin": [101, 101, 309], "destination": [205, 205,␣
↪101], "car": [2, 3, 7]})
>>> demand = from_df(
...     trips, row="origin", col="destination", subset=["car"],
...     index=[101, 205, 309], index_name="zones", fill_value=-1,
... )
>>> demand["car"].tolist()
[[-1.0, 5.0, -1.0], [-1.0, -1.0, -1.0], [7.0, -1.0, -1.0]]
>>> from_df(trips, row="origin", col="destination").index.tolist()
[101, 205, 309]
```

aequilibrae.matrix.**from_dict**(*data: Mapping[str, _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str]], index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str], *, index_name: str = 'main_index'*) → *AequilibraeMatrix*

aequilibrae.matrix.**from_dict**(*data: Mapping[str, bsr_array | coo_array | csc_array | csr_array | dok_array | lil_array], index: _Buffer | _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | complex | bytes | str | _NestedSequence[complex | bytes | str], *, index_name: str = 'main_index'*) → *AequilibraeSparseMatrix*

Create named square matrices from a mapping of names to array-like values.

The result is an in-memory *AequilibraeMatrix* or *AequilibraeSparseMatrix*, inferred from the first value. All values must have the same storage type. Mixed dense and sparse inputs will raise a `ValueError`. The mapping must contain at least one matrix.

Arrays are copied with their original dtypes. Each sparse array keeps its format. Its shape must match the number of zones in `index`, which sets the row and column order.

Arguments

data (mapping of `str` to array-like or SciPy sparse arrays):
Non-empty collection of named square matrices matching the index length. Values must all be dense or all be sparse, with integer, float, boolean or complex dtypes.

index (array-like): One-dimensional sequence of unique integer zone
IDs from `0` to `2**64 - 1` in row and column order.

index_name (`str`, *Optional*): Zone mapping name. Defaults to
`"main_index"`.

```
>>> demand = from_dict(
...     {"car": [[0, 12], [10, 0]], "bus": [[0, 2], [2, 0]]},
...     index=[101, 205], index_name="zones",
... )
>>> demand["car"].tolist()
[[0, 12], [10, 0]]
>>> from scipy.sparse import csr_array
>>> demand = from_dict(
...     {"car": csr_array([[0, 12], [10, 0]])},
...     index=[101, 205], index_name="zones",
... )
>>> demand["car"].toarray().tolist()
[[0, 12], [10, 0]]
>>> demand.index.tolist()
[101, 205]
```

aequilibrae.matrix.**from_file**(*path: str | PathLike*, *mode: None = None*, ***, *index_name: str | None = None*, *subset: Sequence[str | dict[str, Any]] | None = None*, *store: Literal[False] = False*, *sparse: Literal[False]*) → *AequilibraeMatrix*

aequilibrae.matrix.**from_file**(*path: str | PathLike*, *mode: Literal['r', 'r+'] | None = None*, ***, *index_name: str | None = None*, *subset: None = None*, *store: Literal[True]*, *sparse: bool | None = None*) → *MatrixStore*

aequilibrae.matrix.**from_file**(*path: str | PathLike*, *mode: None = None*, ***, *index_name: str | None = None*, *subset: Sequence[str | dict[str, Any]] | None = None*, *store: Literal[False] = False*, *sparse: bool | None = None*) → *AequilibraeMatrix* | *AequilibraeSparseMatrix*

Load an OMX file as an in-memory matrix or an on-disk store.

By default, this copies all matrices and metadata into an in-memory *AequilibraeMatrix* or *AequilibraeSparseMatrix*, inferred from the first selected matrix. Pass `subset` to load only the matrices you need. Mixed storage types raise `ValueError`.

Use `store=True` to work with the file through a *MatrixStore* instead. The store opens read-only unless you pass `mode="r+"`. Stores do not accept `subset`. Use their `subset()` method to copy selected matrices into memory.

Pass `sparse=True` or `False` to require sparse or dense matrices, respectively. This validates the selection without filtering or converting matrices. An empty selection returns an AequilibraeMatrix. The option is ignored when `store=True`; stores expose both storage types.

If the file has only one zone mapping, it is selected automatically. If there are several, `main_index` is used when available. Otherwise, choose a mapping with `index_name`.

Arguments

path (`str` or `os.PathLike`): Existing OMX file to open.

mode (`str`, *Optional*): Store access mode, `"r"` or
`"r+"`. Defaults to `None`, which opens stores read-only. Must be omitted for in-memory loading.

index_name (`str`, *Optional*): Zone mapping to read. Defaults
to automatic selection of the only mapping, or `"main_index"` when several exist. Required

if neither choice is available.

subset (sequence of `str` or `dict`, *Optional*): Matrix
names or attribute queries to load into memory. Defaults to all matrices. Cannot be combined with `store=True`.

store (`bool`, *Optional*): Return an open
MatrixStore instead of an in-memory *AequilibraeMatrix*. Defaults to `False`.

sparse (`bool`, *Optional*): Expected in-memory storage type.
Defaults to `None`, which infers it from the first selected matrix. Ignored when `store=True`.

```
>>> from pathlib import Path
>>> from tempfile import TemporaryDirectory
>>> demand = from_dict({"car": [[0, 12], [10, 0]]}, index=[101, 205], index_name=
↪"zones")
>>> with TemporaryDirectory() as directory:
...     path = Path(directory) / "demand.omx"
...     demand.save_as_omx(path)
...     loaded = from_file(path, subset=["car"])
...     with from_file(path, store=True, mode="r+") as store:
...         store.mat["car", 101, 205] = 42
...         working = store.subset(["car"])
>>> loaded["car"].tolist()
[[0, 12], [10, 0]]
>>> working["car"].tolist()
[[0, 42], [10, 0]]
```

11.10 aequilibrae.examples

Functions

create_example(path[, from_model])	Copies an example model to a new project project and returns the project handle
list_examples()	

aequilibrae.examples.**create_example**(*path: PathLike | str*, *from_model='sioux_falls'*) → *Project*

Copies an example model to a new project project and returns the project handle

Arguments

path (`str`): Path where to create a new model. must be a non-existing folder/directory.

from_model (`str`, *Optional*): Example to create from *sioux_falls*, *nauru* or *coquimbo*. Defaults to *sioux_falls*

Returns

project (`Project`): Aequilibrae Project handle (open)

aequilibrae.examples.**list_examples**() → List[str]

11.11 aequilibrae.utils

Functions

basic_config(level[, stream])	Configures the root logger for AequilibraE.
ensure_spatialite_binaries()	
set_known_spatialite_folder(spatialite_folder)	

aequilibrae.utils.**basic_config**(*level: int = 20*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*, *format: str = '%(asctime)s;%(levelname)s ; %(message)s'*) → Handler | None

Configures the root logger for AequilibraE.

Sets up a specific handler that works well with tqdm progress bars if available and not running inside QGIS. It disables propagation to avoid interference from root loggers.

Arguments

level (`int`, optional): The logging level to set. Defaults to logging.INFO.

stream (`IO`, optional): The stream to write logs to. Defaults to sys.stdout.

format (`str`, optional): The log format string. Defaults to DEFAULT_FORMAT.

Returns

handler (`logging.Handler`): The handler attached to the 'aequilibrae' logger. If the logger already
has a handler writing to standard out or standard error, returns None.

aequilibrae.utils.**ensure_spatialite_binaries**() → None

aequilibrae.utils.**set_known_spatialite_folder**(*spatialite_folder: PathLike | str*)

PYTHON MODULE INDEX

a

aequilibrae, 267
aequilibrae.distribution, 297
aequilibrae.examples, 318
aequilibrae.matrix, 300
aequilibrae.project, 269
aequilibrae.route_choice, 292
aequilibrae.routing, 272
aequilibrae.traffic, 279
aequilibrae.traffic.vdf, 286
aequilibrae.transit, 286
aequilibrae.utils, 319

INDEX

A

about (*aequilibrae.Project property*), 269
activate() (*aequilibrae.Project method*), 268
add_class() (*aequilibrae.transit.TransitAssignment method*), 286
aequilibrae
    module, 267
aequilibrae.distribution
    module, 297
aequilibrae.examples
    module, 318
aequilibrae.matrix
    module, 300
AequilibraeMatrix (*class in aequilibrae.matrix*), 301
aequilibrae.project
    module, 269
aequilibrae.route_choice
    module, 292
aequilibrae.routing
    module, 272
AequilibraeSparseMatrix (*class in aequilibrae.matrix*), 306
aequilibrae.traffic
    module, 279
aequilibrae.traffic.vdf
    module, 286
aequilibrae.transit
    module, 286
aequilibrae.utils
    module, 319
algorithm (*aequilibrae.transit.TransitAssignment attribute*), 288
algorithms_available() (*aequilibrae.traffic.TrafficAssignment method*), 281
algorithms_available() (*aequilibrae.transit.TransitAssignment method*), 286
all_algorithms (*aequilibrae.route_choice.RouteChoice attribute*), 297
all_algorithms (*aequilibrae.traffic.TrafficAssignment attribute*), 283
all_algorithms (*aequilibrae.transit.TransitAssignment attribute*), 288
all_bfw_conjugacies (*aequilibrae.traffic.TrafficAssignment attribute*), 283
all_nodes (*aequilibrae.routing.Graph property*), 273
allow_path_uturns (*aequilibrae.transit.TransitGraph property*), 292
allow_uturns (*aequilibrae.routing.Graph property*), 273
allow_uturns (*aequilibrae.routing.Network attribute*), 274
allow_uturns_everywhere (*aequilibrae.transit.TransitGraph property*), 292
apply() (*aequilibrae.distribution.GravityApplication method*), 298
apply_derivative() (*aequilibrae.traffic.VDF method*), 285
apply_vdf() (*aequilibrae.traffic.VDF method*), 285
AssignmentResults (*class in aequilibrae.traffic*), 279
available_heaps() (*in module aequilibrae.routing*), 279
available_skims() (*aequilibrae.transit.TransitGraph method*), 289

B

basic_config() (*in module aequilibrae.utils*), 319
bind() (*aequilibrae.traffic.AssignmentResults method*), 280
block_centroid_flows (*aequilibrae.routing.Graph property*), 273
build_graph() (*aequilibrae.routing.Network method*), 274
builtin_vdfs() (*in module aequilibrae.traffic.vdf*), 286

C

calibrate() (*aequilibrae.distribution.GravityCalibration method*), 299
centroid_count (*aequilibrae.routing.Graph property*), 273
centroids (*aequilibrae.routing.Graph property*), 273
centroids (*aequilibrae.routing.Network attribute*), 274
check_file_indices() (*aequilibrae.Project method*), 268
check_valid() (*aequilibrae.traffic.VDF method*), 285

class_results (*aequilibrae.traffic.TrafficAssignment property*), 284
classes (*aequilibrae.transit.TransitAssignment attribute*), 288
clear() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
clear() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 306
clear() (*aequilibrae.matrix.MatrixStore method*), 310
clear_turn_restrictions() (*aequilibrae.transit.TransitGraph method*), 289
clone_scenario() (*aequilibrae.Project method*), 268
close() (*aequilibrae.matrix.MatrixStore method*), 311
close() (*aequilibrae.Project method*), 268
collapse_links_into_nodes() (*aequilibrae.project.NetworkSimplifier method*), 271
compact (*aequilibrae.routing.Graph property*), 273
compact_link_loads (*aequilibrae.traffic.AssignmentResults property*), 280
compute_path() (*aequilibrae.routing.Graph method*), 273
compute_path() (*aequilibrae.routing.PathResults method*), 276
compute_path() (*aequilibrae.transit.TransitGraph method*), 289
compute_skims() (*aequilibrae.routing.Graph method*), 273
compute_skims() (*aequilibrae.transit.TransitGraph method*), 289
config (*aequilibrae.transit.TransitGraph property*), 292
connectors (*aequilibrae.routing.PathResults property*), 278
convert_str_function_into_function() (*aequilibrae.traffic.VDF static method*), 285
cores (*aequilibrae.transit.TransitAssignment attribute*), 288
cost (*aequilibrae.routing.Graph property*), 273
cost (*aequilibrae.traffic.TrafficClass attribute*), 284
cost_field (*aequilibrae.routing.Graph attribute*), 273
create() (*aequilibrae.Project class method*), 267
create_compressed_link_network_mapping() (*aequilibrae.transit.TransitGraph method*), 289
create_empty_scenario() (*aequilibrae.Project method*), 268
create_example() (*in module aequilibrae.examples*), 318

D

db_connection (*aequilibrae.Project property*), 269
deactivate() (*aequilibrae.Project method*), 268
default_parameters (*aequilibrae.route_choice.RouteChoice attribute*), 297
default_types() (*aequilibrae.transit.TransitGraph method*), 290
demand (*aequilibrae.traffic.TrafficClass attribute*), 284
description (*aequilibrae.transit.TransitAssignment attribute*), 288
directions (*aequilibrae.routing.Network property*), 275
disconnected_nodes() (*aequilibrae.routing.Graph method*), 273
disconnected_nodes() (*aequilibrae.transit.TransitGraph method*), 290
distances (*aequilibrae.routing.PathResults property*), 278
doWork() (*aequilibrae.routing.NetworkSkimming method*), 275

E

elementwise_cores (*aequilibrae.transit.TransitAssignment attribute*), 288
ensure_spatialite_binaries() (*in module aequilibrae.utils*), 319
estimate_heuristic_scale() (*in module aequilibrae.routing*), 279
exclude_links() (*aequilibrae.transit.TransitGraph method*), 290
execute() (*aequilibrae.route_choice.RouteChoice method*), 292
execute() (*aequilibrae.routing.NetworkSkimming method*), 275
execute() (*aequilibrae.traffic.TrafficAssignment method*), 281
execute() (*aequilibrae.transit.TransitAssignment method*), 286
execute_from_pandas() (*aequilibrae.route_choice.RouteChoice method*), 292
execute_from_path_files() (*aequilibrae.route_choice.RouteChoice method*), 293
execute_single() (*aequilibrae.route_choice.RouteChoice method*), 293

F

field_values() (*aequilibrae.routing.Network method*), 274
fields (*aequilibrae.routing.Network property*), 275
file_default (*aequilibrae.project.Parameters attribute*), 272
find_applicable() (*aequilibrae.project.MigrationManager method*), 270
fit() (*aequilibrae.distribution.Ipf method*), 300
fixed_cost_field (*aequilibrae.traffic.TrafficClass attribute*), 284
fixed_cost_multiplier (*aequilibrae.traffic.TrafficClass attribute*), 284
free_flow_tt (*aequilibrae.transit.TransitAssignment attribute*), 288

from_df() (*in module aequilibrae.matrix*), 315
from_dict() (*in module aequilibrae.matrix*), 316
from_file() (*in module aequilibrae.matrix*), 317
from_omx() (*aequilibrae.matrix.AequilibraeMatrix class method*), 301
from_omx() (*aequilibrae.matrix.AequilibraeSparseMatrix class method*), 306

G

get() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
get() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
get() (*aequilibrae.matrix.MatrixStore method*), 311
get_heaps() (*aequilibrae.routing.PathResults method*), 276
get_heaps() (*aequilibrae.routing.SkimResults method*), 278
get_heaps() (*aequilibrae.traffic.AssignmentResults static method*), 280
get_heuristics() (*aequilibrae.routing.PathResults method*), 276
get_load_results() (*aequilibrae.route_choice.RouteChoice method*), 293
get_load_results() (*aequilibrae.traffic.AssignmentResults method*), 280
get_load_results() (*aequilibrae.transit.TransitAssignmentResults method*), 288
get_network_links() (*aequilibrae.routing.PathResults method*), 277
get_results() (*aequilibrae.route_choice.RouteChoice method*), 293
get_select_link_loading_results() (*aequilibrae.route_choice.RouteChoice method*), 293
get_select_link_od_matrix_results() (*aequilibrae.route_choice.RouteChoice method*), 294
get_skim_results() (*aequilibrae.traffic.TrafficAssignment method*), 282
get_skim_results() (*aequilibrae.transit.TransitAssignment method*), 286
get_sl_results() (*aequilibrae.traffic.AssignmentResults method*), 280
get_vdfs() (*aequilibrae.project.Parameters method*), 271
graph (*aequilibrae.transit.TransitClass attribute*), 288
Graph (*class in aequilibrae.routing*), 273
graph_options (*aequilibrae.traffic.TrafficClass attribute*), 284
GravityApplication (*class in aequilibrae.distribution*), 297
GravityCalibration (*class in aequilibrae.distribution*), 299

H

has_turn_restrictions (*aequilibrae.routing.Graph property*), 273
has_turn_restrictions (*aequilibrae.transit.TransitGraph property*), 292
heap (*aequilibrae.traffic.TrafficClass attribute*), 284

I

imat (*aequilibrae.matrix.AequilibraeMatrix property*), 304
imat (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 308
imat (*aequilibrae.matrix.MatrixStore property*), 313
index (*aequilibrae.matrix.AequilibraeMatrix property*), 304
index (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 309
index (*aequilibrae.matrix.MatrixStore property*), 313
index_name (*aequilibrae.matrix.AequilibraeMatrix property*), 305
index_name (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 309
index_name (*aequilibrae.matrix.MatrixStore property*), 314
info (*aequilibrae.traffic.TrafficClass property*), 284
info (*aequilibrae.transit.TransitClass property*), 289
info() (*aequilibrae.route_choice.RouteChoice method*), 294
info() (*aequilibrae.traffic.TrafficAssignment method*), 282
info() (*aequilibrae.transit.TransitAssignment method*), 286
installation, 1
Ipf (*class in aequilibrae.distribution*), 299
items() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
items() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
items() (*aequilibrae.matrix.MatrixStore method*), 311

K

keys() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
keys() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
keys() (*aequilibrae.matrix.MatrixStore method*), 311

L

link_ids (*aequilibrae.routing.Network property*), 275
link_loads (*aequilibrae.traffic.AssignmentResults property*), 280
links (*aequilibrae.routing.Network attribute*), 275
list_examples() (*in module aequilibrae.examples*), 318
list_scenarios() (*aequilibrae.Project method*), 268

load() (*aequilibrae.distribution.SyntheticGravityModel method*), 300
load_default() (*aequilibrae.project.Parameters class method*), 271
log() (*aequilibrae.Project method*), 268
log_specification() (*aequilibrae.route_choice.RouteChoice method*), 294
log_specification() (*aequilibrae.traffic.TrafficAssignment method*), 282
log_specification() (*aequilibrae.transit.TransitAssignment method*), 286
lonlat (*aequilibrae.routing.Network attribute*), 275

M

make_finite_difference_derivative() (*aequilibrae.traffic.VDF method*), 285
mapping (*aequilibrae.routing.Graph attribute*), 274
mapping (*aequilibrae.routing.Network attribute*), 275
mark_all_as_seen() (*aequilibrae.project.MigrationManager method*), 270
mat (*aequilibrae.matrix.AequilibraeMatrix property*), 305
mat (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 309
mat (*aequilibrae.matrix.MatrixStore property*), 314
matrices (*aequilibrae.Project property*), 269
matrix (*aequilibrae.transit.TransitClass attribute*), 289
matrix_metadata() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
matrix_metadata() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
matrix_metadata() (*aequilibrae.matrix.MatrixStore method*), 311
MatrixStore (*class in aequilibrae.matrix*), 310
metadata (*aequilibrae.matrix.AequilibraeMatrix property*), 305
metadata (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 309
metadata (*aequilibrae.matrix.MatrixStore property*), 314
MigrationManager (*class in aequilibrae.project*), 270
mode (*aequilibrae.routing.Graph attribute*), 274
mode (*aequilibrae.traffic.TrafficClass attribute*), 284
mode_mask() (*aequilibrae.routing.Network method*), 274
modes (*aequilibrae.routing.Network attribute*), 275
module
    aequilibrae, 267
    aequilibrae.distribution, 297
    aequilibrae.examples, 318
    aequilibrae.matrix, 300
    aequilibrae.project, 269
    aequilibrae.route_choice, 292
    aequilibrae.routing, 272
    aequilibrae.traffic, 279
    aequilibrae.traffic.vdf, 286
    aequilibrae.transit, 286
    aequilibrae.utils, 319

N

name (*aequilibrae.traffic.TrafficClass attribute*), 284
name (*aequilibrae.transit.TransitClass attribute*), 289
Network (*class in aequilibrae.routing*), 274
network_migration_file (*aequilibrae.project.MigrationManager attribute*), 270
NetworkSimplifier (*class in aequilibrae.project*), 271
NetworkSkimming (*class in aequilibrae.routing*), 275
node_ids (*aequilibrae.routing.PathResults property*), 278
num_links (*aequilibrae.routing.Graph property*), 274
num_nodes (*aequilibrae.routing.Graph property*), 274
num_zones (*aequilibrae.routing.Graph property*), 274

O

omx (*aequilibrae.matrix.MatrixStore property*), 314
open() (*aequilibrae.Project class method*), 268
output (*aequilibrae.traffic.AssignmentResults property*), 280

P

parameters (*aequilibrae.Project property*), 269
Parameters (*class in aequilibrae.project*), 271
path_to_file (*aequilibrae.Project property*), 269
PathResults (*class in aequilibrae.routing*), 276
pce (*aequilibrae.traffic.TrafficClass attribute*), 284
plot_vdf() (*aequilibrae.traffic.VDF method*), 285
pop() (*aequilibrae.matrix.AequilibraeMatrix method*), 302
pop() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
pop() (*aequilibrae.matrix.MatrixStore method*), 311
popitem() (*aequilibrae.matrix.AequilibraeMatrix method*), 303
popitem() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
popitem() (*aequilibrae.matrix.MatrixStore method*), 311
post_process() (*aequilibrae.route_choice.SubAreaAnalysis method*), 297
predecessors (*aequilibrae.routing.PathResults property*), 278
prepare() (*aequilibrae.route_choice.RouteChoice method*), 294
prepare() (*aequilibrae.routing.SkimResults method*), 278
prepare() (*aequilibrae.transit.TransitAssignmentResults method*), 288
prepare_graph() (*aequilibrae.transit.TransitGraph method*), 290
Project (*class in aequilibrae*), 267
project_base_path (*aequilibrae.Project property*), 269

project_parameters (*aequilibrae.Project property*), 269

R

recompute_psl() (*aequilibrae.route_choice.RouteChoice method*), 294
remove_dead_ends (*aequilibrae.routing.Graph property*), 274
report() (*aequilibrae.traffic.TrafficAssignment method*), 282
report() (*aequilibrae.transit.TransitAssignment method*), 287
reset() (*aequilibrae.routing.PathResults method*), 277
reset() (*aequilibrae.traffic.AssignmentResults method*), 280
reset() (*aequilibrae.transit.TransitAssignmentResults method*), 288
restore_default() (*aequilibrae.project.Parameters method*), 272
results (*aequilibrae.Project property*), 269
results (*aequilibrae.transit.TransitClass attribute*), 289
results() (*aequilibrae.traffic.TrafficAssignment method*), 282
results() (*aequilibrae.transit.TransitAssignment method*), 287
results_connection (*aequilibrae.Project property*), 269
reverse() (*aequilibrae.transit.TransitGraph method*), 291
road (*aequilibrae.Project property*), 269
RouteChoice (*class in aequilibrae.route_choice*), 292
routing (*aequilibrae.routing.Graph attribute*), 274
run (*aequilibrae.Project property*), 269

S

save() (*aequilibrae.distribution.SyntheticGravityModel method*), 300
save_as_omx() (*aequilibrae.matrix.AequilibraeMatrix method*), 303
save_as_omx() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 307
save_as_omx() (*aequilibrae.matrix.MatrixStore method*), 311
save_compressed_correspondence() (*aequilibrae.transit.TransitGraph method*), 291
save_link_flows() (*aequilibrae.route_choice.RouteChoice method*), 294
save_path_files() (*aequilibrae.route_choice.RouteChoice method*), 295
save_results() (*aequilibrae.traffic.TrafficAssignment method*), 282
save_results() (*aequilibrae.transit.TransitAssignment method*), 287
save_select_link() (*aequilibrae.route_choice.RouteChoice method*), 295
save_select_link_flows() (*aequilibrae.route_choice.RouteChoice method*), 295
save_select_link_flows() (*aequilibrae.traffic.TrafficAssignment method*), 283
save_select_link_matrices() (*aequilibrae.route_choice.RouteChoice method*), 295
save_select_link_matrices() (*aequilibrae.traffic.TrafficAssignment method*), 283
save_select_link_results() (*aequilibrae.traffic.TrafficAssignment method*), 283
save_skims() (*aequilibrae.traffic.TrafficAssignment method*), 283
save_to_project() (*aequilibrae.distribution.GravityApplication method*), 299
save_to_project() (*aequilibrae.distribution.Ipf method*), 300
select_link_flows() (*aequilibrae.traffic.TrafficAssignment method*), 283
select_link_loading (*aequilibrae.traffic.AssignmentResults property*), 280
select_link_od (*aequilibrae.traffic.AssignmentResults property*), 280
select_link_od (*aequilibrae.traffic.TrafficClass attribute*), 284
select_link_sets (*aequilibrae.traffic.TrafficClass attribute*), 284
selected_links() (*aequilibrae.routing.Graph method*), 273
set_algorithm() (*aequilibrae.transit.TransitAssignment method*), 287
set_blocked_centroid_flows() (*aequilibrae.transit.TransitGraph method*), 291
set_choice_set_generation() (*aequilibrae.route_choice.RouteChoice method*), 295
set_classes() (*aequilibrae.transit.TransitAssignment method*), 287
set_cores() (*aequilibrae.route_choice.RouteChoice method*), 296
set_cores() (*aequilibrae.routing.NetworkSkimming method*), 275
set_cores() (*aequilibrae.traffic.AssignmentResults method*), 280
set_cores() (*aequilibrae.transit.TransitAssignment method*), 287
set_cores() (*aequilibrae.transit.TransitAssignmentResults method*), 288
set_costs() (*aequilibrae.routing.Graph method*), 273
set_demand() (*aequilibrae.route_choice.RouteChoice method*), 296
set_frequency_field() (*aequili-*

brae.transit.TransitAssignment method), 287
set_graph() (*aequilibrae.transit.TransitGraph method*), 291
set_graph_data() (*aequilibrae.routing.PathResults method*), 277
set_heap() (*aequilibrae.routing.PathResults method*), 277
set_heap() (*aequilibrae.routing.SkimResults method*), 279
set_heap() (*aequilibrae.traffic.AssignmentResults method*), 280
set_heuristic() (*aequilibrae.routing.PathResults method*), 277
set_known_spatialite_folder() (*in module aequilibrae.utils*), 319
set_save_routes() (*aequilibrae.route_choice.RouteChoice method*), 296
set_select_links() (*aequilibrae.route_choice.RouteChoice method*), 296
set_skimming() (*aequilibrae.transit.TransitGraph method*), 291
set_skimming_fields() (*aequilibrae.transit.TransitAssignment method*), 288
set_skimming_fields() (*aequilibrae.transit.TransitGraph method*), 291
set_time_field() (*aequilibrae.transit.TransitAssignment method*), 288
set_turn_penalty_dimension() (*aequilibrae.transit.TransitGraph method*), 291
set_turn_restrictions() (*aequilibrae.transit.TransitGraph method*), 291
setdefault() (*aequilibrae.matrix.AequilibraeMatrix method*), 303
setdefault() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 308
setdefault() (*aequilibrae.matrix.MatrixStore method*), 312
settlement_order (*aequilibrae.routing.PathResults property*), 278
shape (*aequilibrae.matrix.AequilibraeMatrix property*), 305
shape (*aequilibrae.matrix.AequilibraeSparseMatrix property*), 310
shape (*aequilibrae.matrix.MatrixStore property*), 315
signal (*aequilibrae.project.NetworkSimplifier attribute*), 271
signal (*aequilibrae.routing.NetworkSkimming attribute*), 276
simplify() (*aequilibrae.project.NetworkSimplifier method*), 271
skim_congested() (*aequilibrae.traffic.TrafficAssignment method*), 283
skim_fields (*aequilibrae.routing.Graph property*), 274
skimming (*aequilibrae.routing.Graph attribute*), 274
skimming (*aequilibrae.traffic.TrafficClass attribute*), 285
SkimResults (*class in aequilibrae.routing*), 278
skims (*aequilibrae.traffic.AssignmentResults property*), 280
state (*aequilibrae.traffic.AssignmentResults property*), 280
status() (*aequilibrae.project.MigrationManager method*), 270
SubAreaAnalysis (*class in aequilibrae.route_choice*), 297
subset() (*aequilibrae.matrix.AequilibraeMatrix method*), 304
subset() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 308
subset() (*aequilibrae.matrix.MatrixStore method*), 312
SyntheticGravityModel (*class in aequilibrae.distribution*), 300

T

tables (*aequilibrae.Project property*), 269
terminal_states (*aequilibrae.routing.PathResults property*), 278
threading_threshold (*aequilibrae.transit.TransitAssignment attribute*), 288
time_field (*aequilibrae.transit.TransitAssignment attribute*), 288
total_flow (*aequilibrae.transit.TransitAssignment attribute*), 288
total_link_loads (*aequilibrae.traffic.AssignmentResults property*), 280
total_turn_penalty (*aequilibrae.traffic.AssignmentResults property*), 280
totals() (*aequilibrae.matrix.AequilibraeMatrix method*), 304
totals() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 308
totals() (*aequilibrae.matrix.MatrixStore method*), 313
TrafficAssignment (*class in aequilibrae.traffic*), 280
TrafficClass (*class in aequilibrae.traffic*), 284
transaction() (*aequilibrae.Project method*), 268
transit (*aequilibrae.Project property*), 269
transit_connection (*aequilibrae.Project property*), 269
transit_migration_file (*aequilibrae.project.MigrationManager attribute*), 270
TransitAssignment (*class in aequilibrae.transit*), 286
TransitAssignmentResults (*class in aequilibrae.transit*), 288
TransitClass (*class in aequilibrae.transit*), 288
TransitGraph (*class in aequilibrae.transit*), 289
turn_costs (*aequilibrae.routing.PathResults property*), 278
turn_restrictions (*aequilibrae.routing.Network attribute*), 275

U

unassigned_demand (*aequilibrae.traffic.AssignmentResults property*), 280
update() (*aequilibrae.matrix.AequilibraeMatrix method*), 304
update() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 308
update() (*aequilibrae.matrix.MatrixStore method*), 313
update_trace() (*aequilibrae.routing.PathResults method*), 277
upgrade() (*aequilibrae.Project method*), 268
upgrade() (*aequilibrae.project.MigrationManager method*), 270
use_scenario() (*aequilibrae.Project method*), 269

V

validate_centroids() (*aequilibrae.routing.Network method*), 274
validate_mode() (*aequilibrae.routing.Network method*), 274
values() (*aequilibrae.matrix.AequilibraeMatrix method*), 304
values() (*aequilibrae.matrix.AequilibraeSparseMatrix method*), 308
values() (*aequilibrae.matrix.MatrixStore method*), 313
VDF (*class in aequilibrae.traffic*), 285
vot (*aequilibrae.traffic.TrafficClass attribute*), 285

W

write_back() (*aequilibrae.project.Parameters method*), 272