AequilibraE 1.7.0

Pedro Camargo

Aug 03, 2026

CONTENTS

1 Installation 1

2 The AequilibraE project 5

3 Run module 73

4 Network Manipulation 77

5 Distribution Procedures 103

6 Path Computation 119

7 Static Traffic Assignment 131

8 Public Transport 187

9 Route Choice 213

10 Other Applications 231

11 API Reference 239

Python Module Index 461

Index 465

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 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).

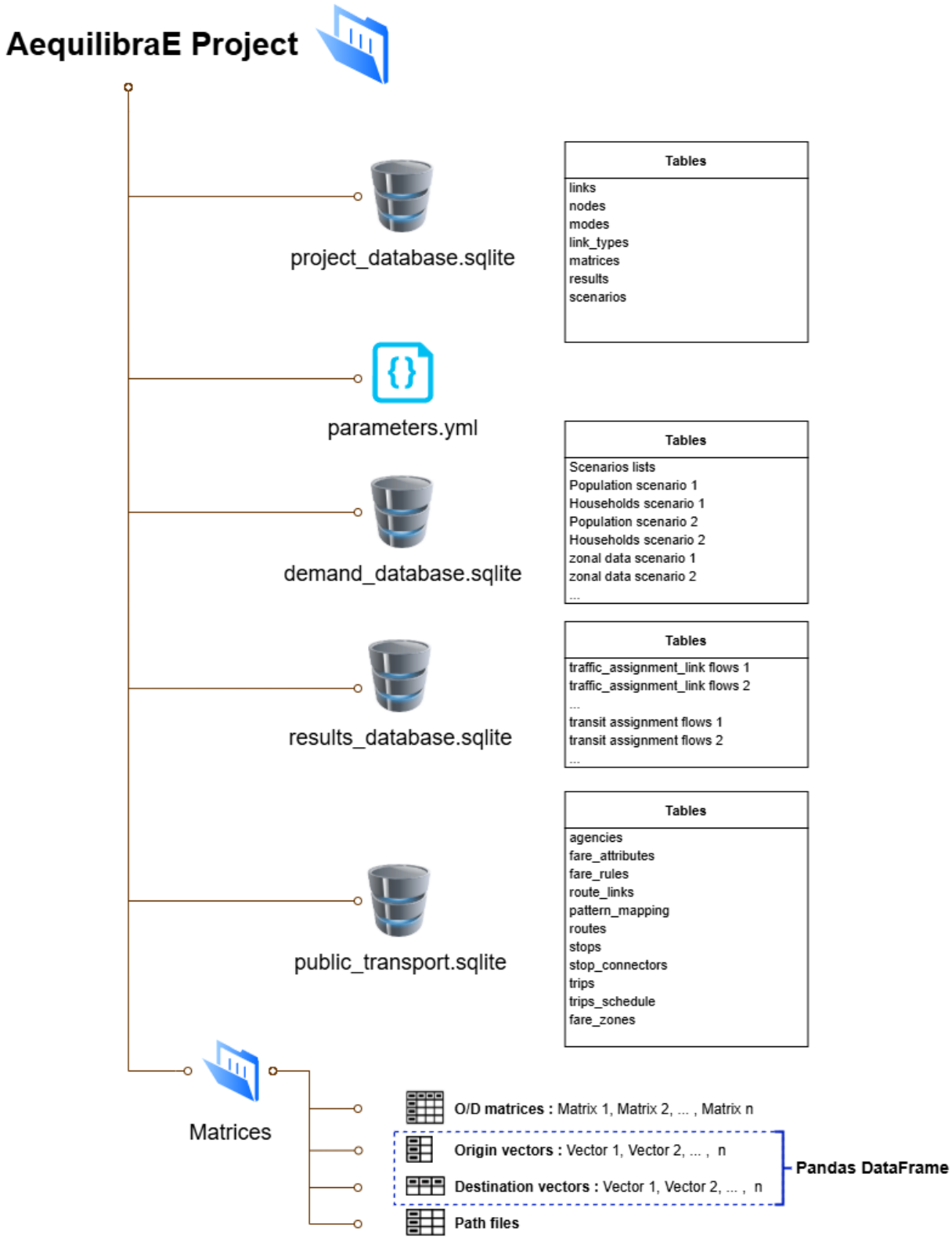
Thus, using/manipulating each one of an AequilibraE model components can be done in different ways depending on the tool you use for such.

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, as well as the documentation records of all matrices and procedure results stored in other folders and databases.

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 matrices in binary format (in AequilibraE's native AEM or OMX formats) 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** is created on-the-fly the first time a command to save a tabular result into the model is invoked, so the user does not need to worry about its existence until it is automatically created.

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

AequilibraE does not currently support turn penalties and/or bans. Their implementation requires a complete overahaul of the path-building code, so that is still a long-term goal, barred specific development efforts.

See also

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

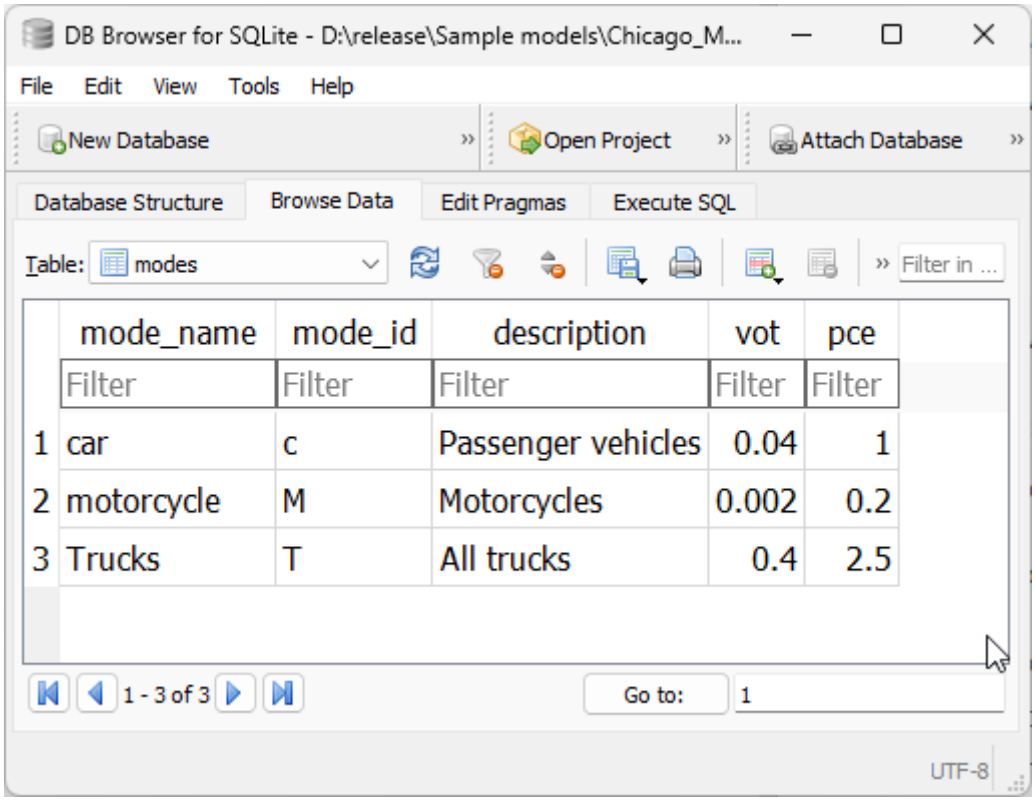
Modes table

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.

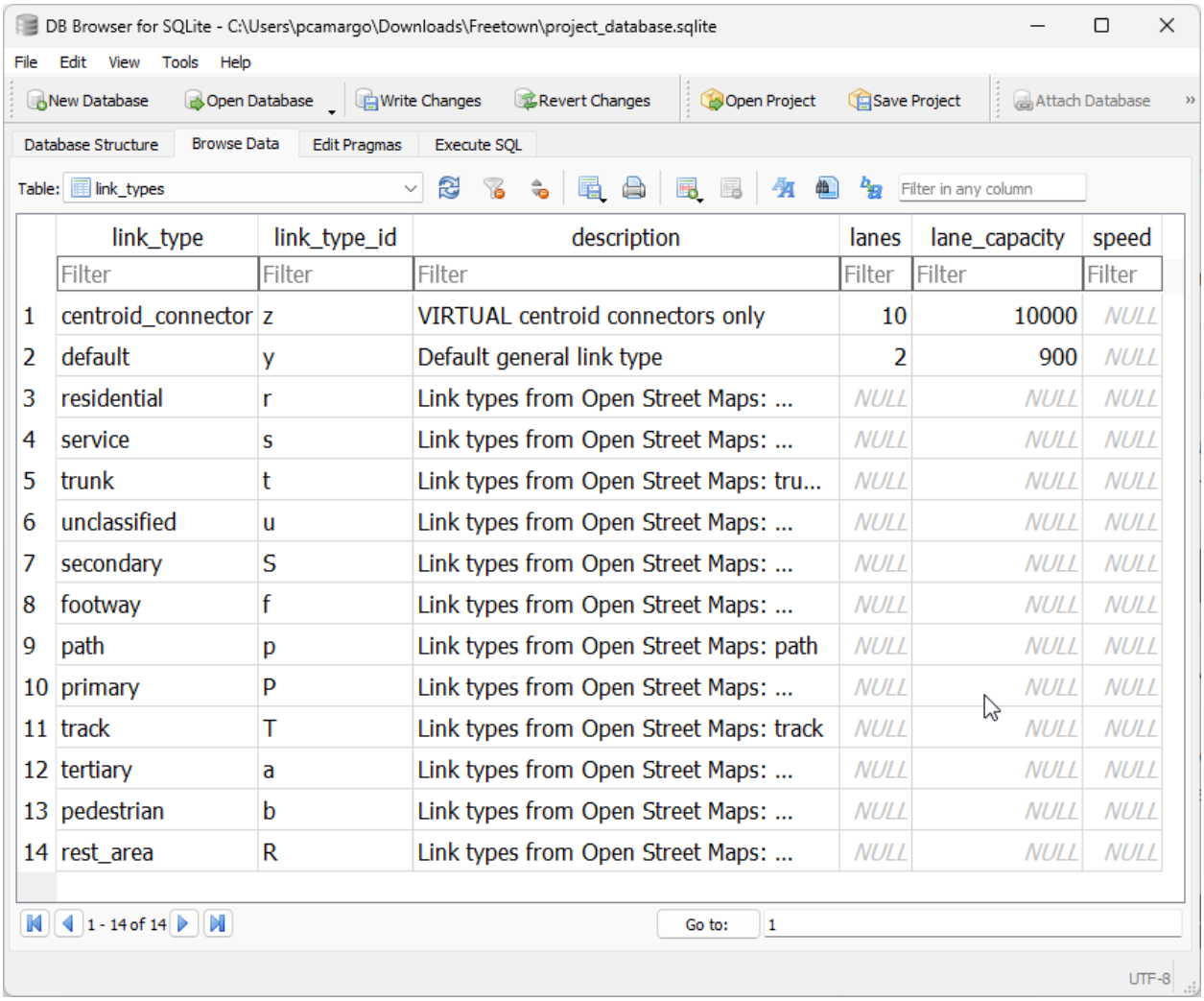
Reserved values

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.

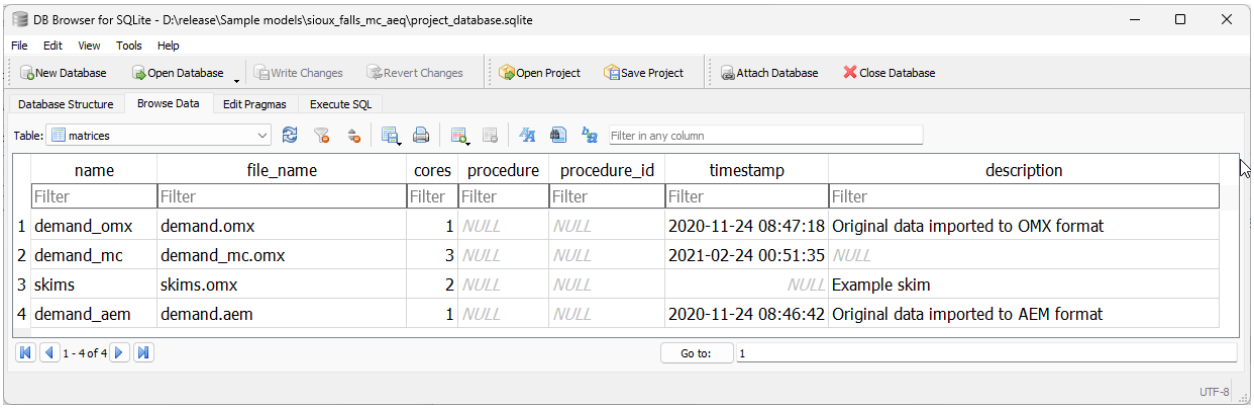
See also

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

Matrices table

The **matrices** table in the project_database is nothing more than an index of all matrix files contained in the matrices folder inside the AequilibraE project.

This index, which looks like below, has two main columns. The first one is the **file_name**, which contains the actual file name in disk as to allow AequilibraE to find the file, and **name**, which is the name by which the user should refer to the matrix in order to access it through the API.



As AequilibraE is fully compatible with OMX, the index can have a mix of matrix types (AEM and OMX) without prejudice to functionality.

See also

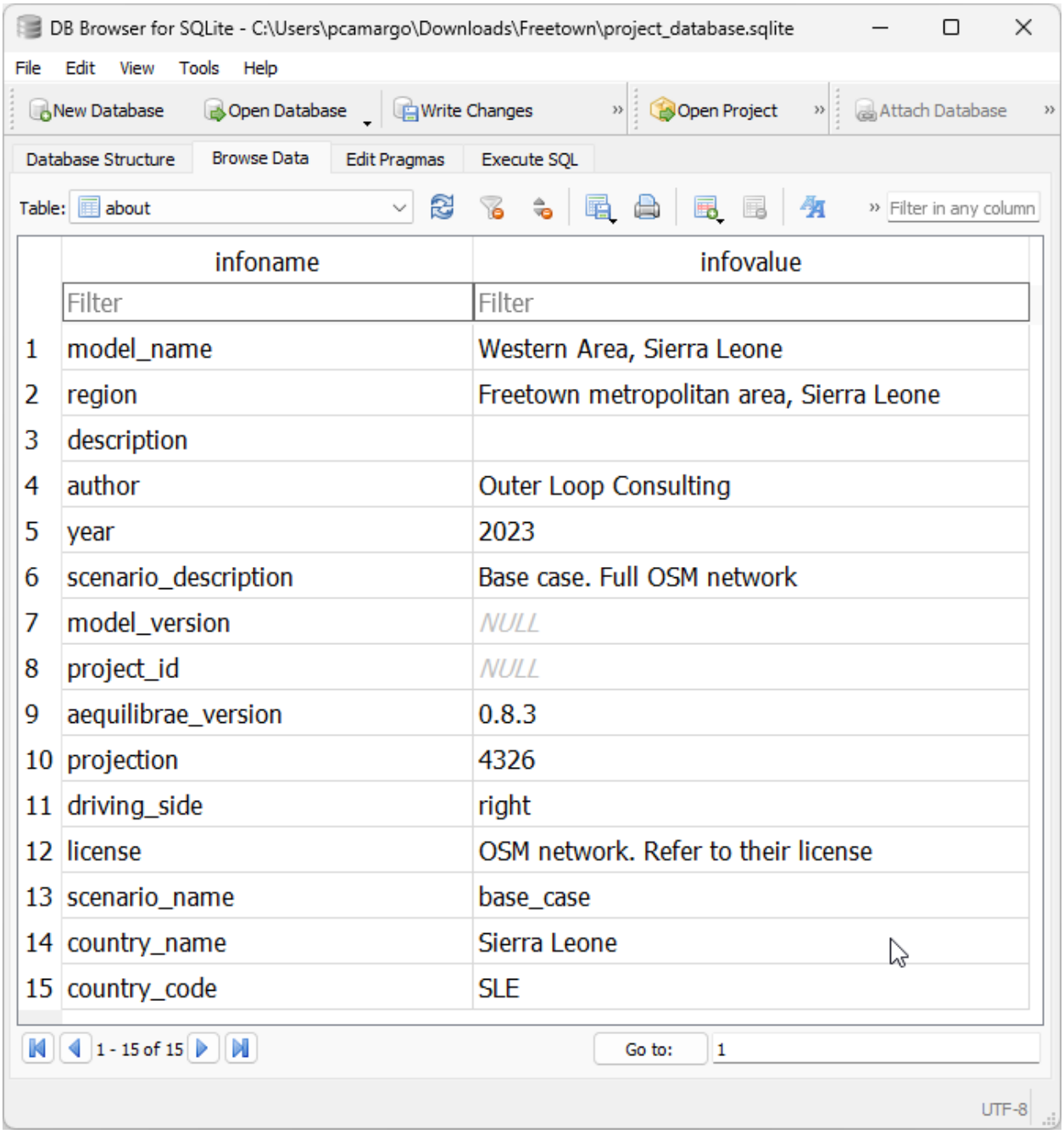
- *aequilibrae.project.data.matrices.Matrices()*
  Class documentation
- *matrices table structure*
  Data model

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.



	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

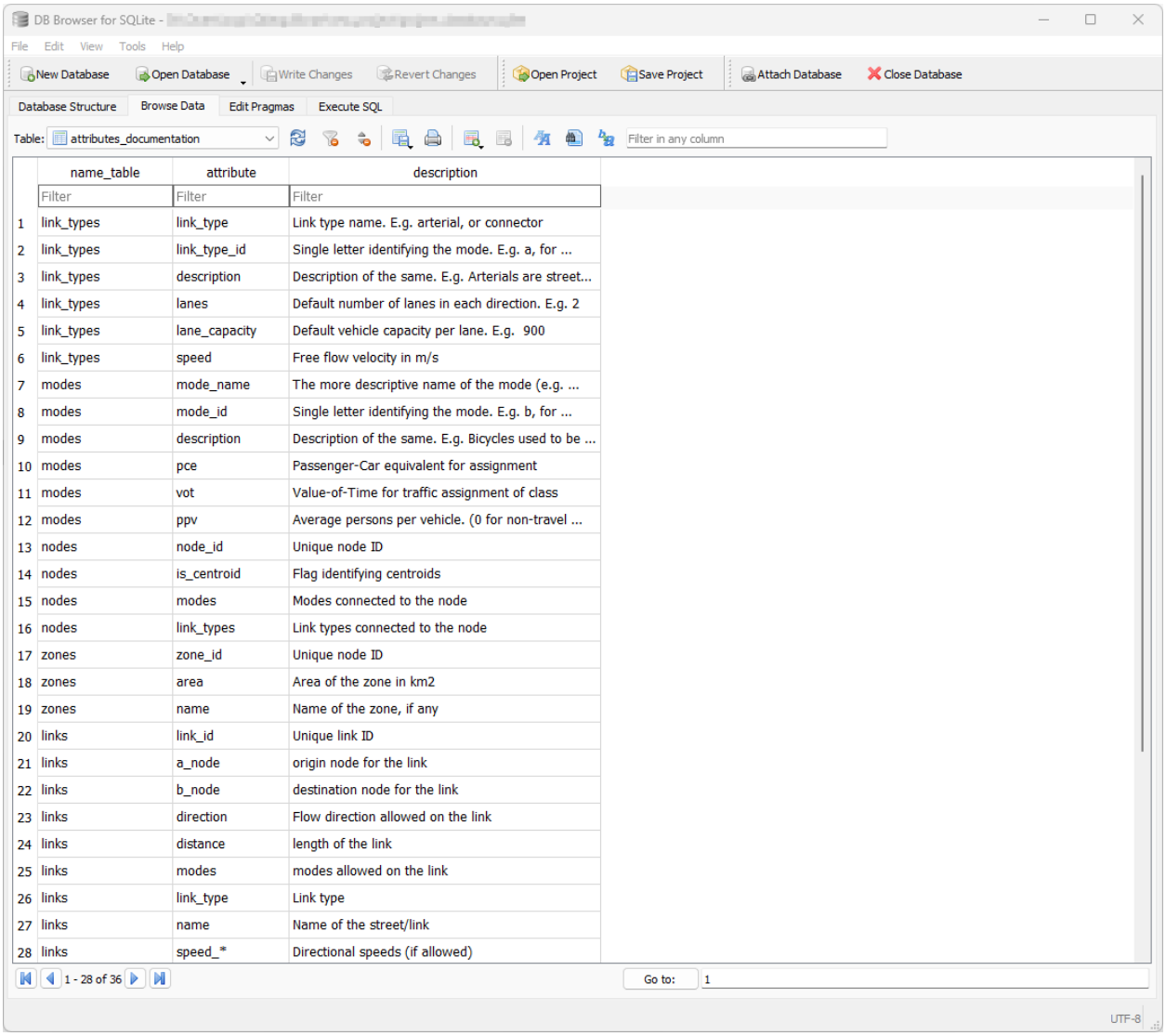
See also

- *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

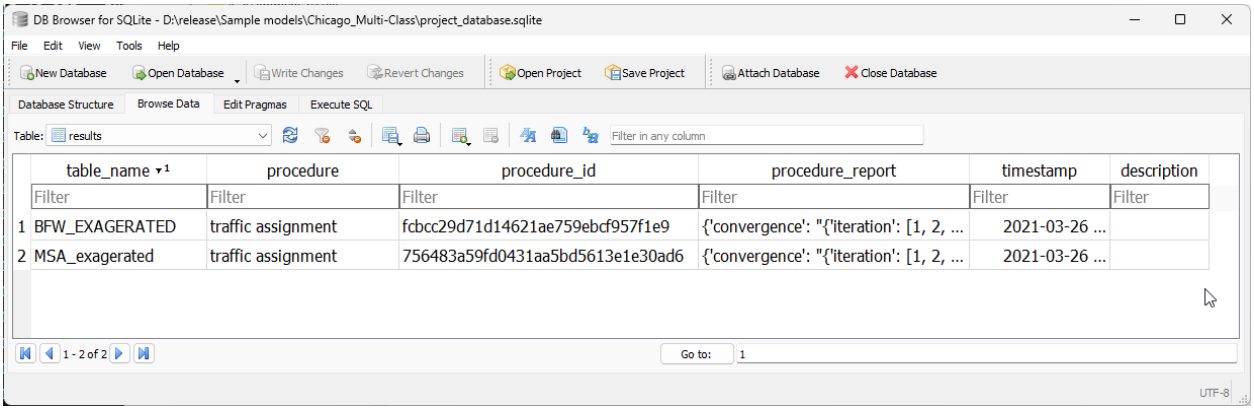
- *attributes documentation table structure*
  Data model

Results table

The **results** table exists to hold the metadata for the results stored in the **results_database.sqlite** 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**.

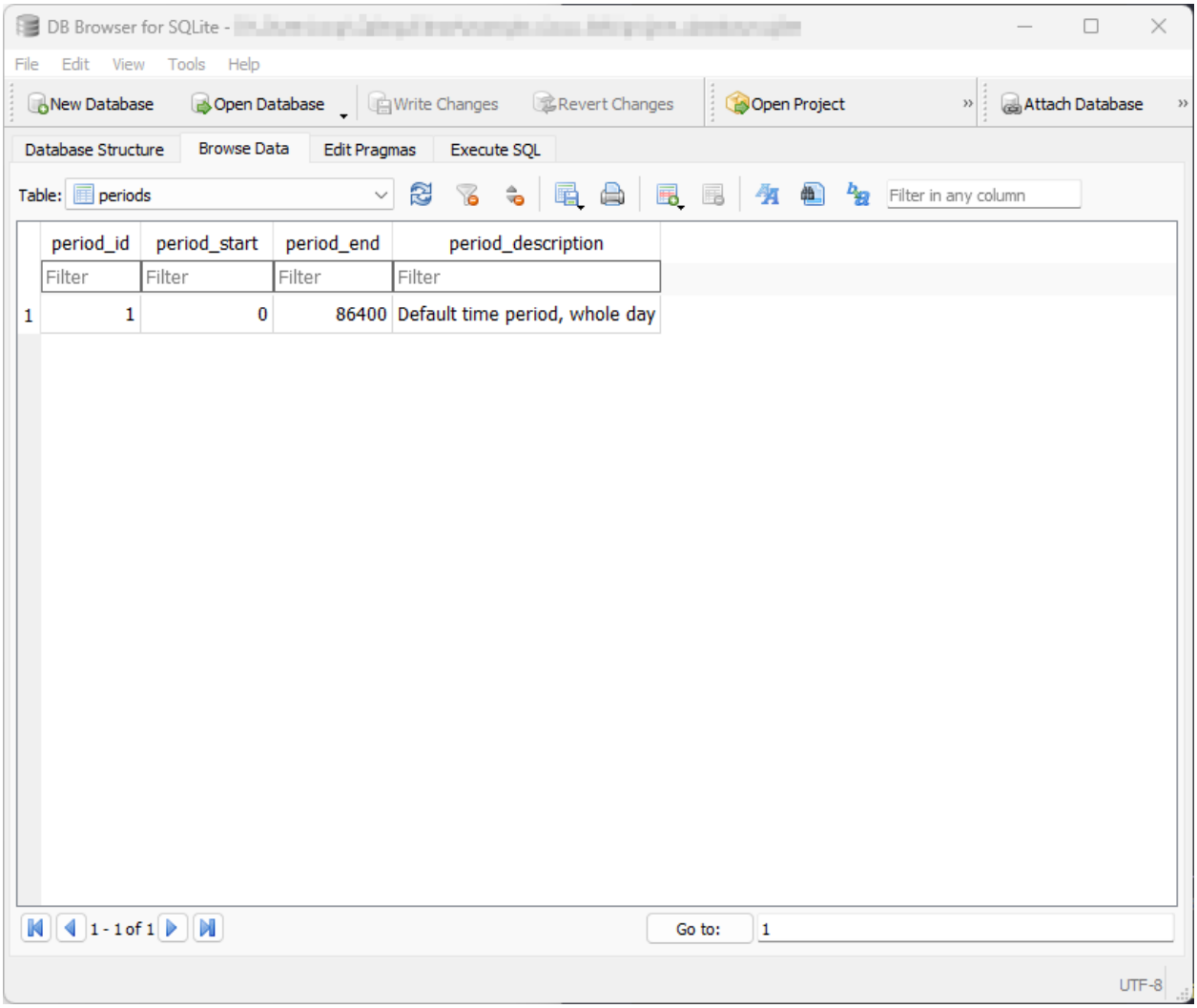
As a simple table, it looks as follows:



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');
```

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, 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	
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	
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	
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,
                                   modes           TEXT    NOT NULL,
                                   link_type       TEXT    REFERENCES link_types(link_
↪type) ON update RESTRICT ON delete RESTRICT,
```

(continues on next page)

(continued from previous page)

```
                                    'name'          TEXT,
                                    speed_ab        NUMERIC,
                                    speed_ba        NUMERIC,
                                    travel_time_ab  NUMERIC,
                                    travel_time_ba  NUMERIC,
                                    capacity_ab     NUMERIC,
                                    capacity_ba     NUMERIC
                                    CHECK(TYPEOF(link_id) == 'integer')
                                    CHECK(link_id > 0)
                                    CHECK(TYPEOF(a_node) == 'integer')
                                    CHECK(TYPEOF(b_node) == 'integer')
                                    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(
↪'links','name', 'Name of the street/link');
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(
```

(continues on next page)

(continued from previous page)

```
↪'links','travel_time_*', 'Directional free-flow travel time (if allowed)');
```

matrices table structure

The *matrices* table holds information about all matrices that exists in the project *matrix* folder.

The **name** field presents the name of the table.

The **file_name** field holds the file name.

The **cores** field holds the information on the number of cores used.

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

The **procedure_id** field holds a unique alphanumeric identifier for this procedure.

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
name*	TEXT	NO	
file_name	TEXT	NO	
cores	INTEGER	NO	1
procedure	TEXT	YES	
procedure_id	TEXT	YES	
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 matrices (name          TEXT     NOT NULL PRIMARY KEY,
                                     file_name     TEXT     NOT NULL UNIQUE,
                                     cores         INTEGER  NOT NULL DEFAULT 1,
                                     procedure     TEXT,
                                     procedure_id  TEXT,
                                     timestamp     DATETIME DEFAULT current_timestamp,
                                     description   TEXT);


CREATE INDEX name_matrices ON matrices (name);
```

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');
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	
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
                                  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);

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');
```

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');
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');
```

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);
```

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(
↪'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.');
```

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(
```

(continues on next page)

(continued from previous page)

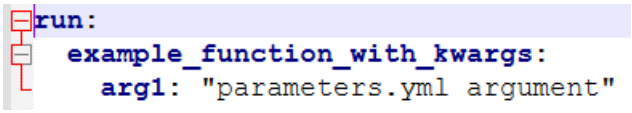
```
↪'zones','zone_id', 'Unique node ID');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','area', 'Area of the zone in km2');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
↪'zones','name', 'Name of the zone, if any');
```

2.2.3 Parameters YAML File

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

Run

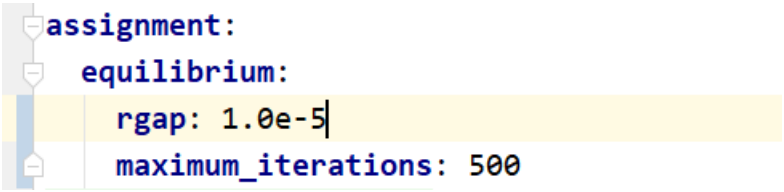
The run section of the parameter file defines the default keyword arguments for the callable objects in the *Run module*. Each subsection names a callable symbol within the `run/__init__.py` module, if the symbol does not exist a `RuntimeError` will be raised when `project.run` is accessed. The arguments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



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.



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.

```
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
```

For the case of all non-mandatory fields, two more parameters are possible: 'osm_source' and 'osm_behaviour'. Those two fields provide the necessary information for importing data from OpenStreetMap in case such resource is required, and they work in the following way:

'osm_source': The name of the tag for which data needs to be retrieved. Common tags are **highway**, **maxspeed** and **name**. The import result will contain a null value for all links that do not contain a value for such tag.

Within OSM, there is the concept of tags for each link direction, such as **maxspeed:forward** and **maxspeed:backward**. However, it is not always that a two-directional link contains tag values for both directions, and it might have only a tag value for **maxspeed**.

Although for **maxspeed** (which is the value for posted speed) we might want to copy the same value for both directions, that would not be true for parameters such as **lanes**, which we might want to split in half for both directions (cases with an odd number of lanes usually have forward/backward values tagged). For this reason, one can use the parameter 'osm_behaviour' to define what to do with numeric tag values that have not been tagged for both directions. the allowed values for this parameter are **copy** and **divide**, as shown below.

```
two-way:
  - lanes:
      description: lanes
      type: integer
      osm_source: lanes
      osm_behaviour: divide
  - capacity:
      description: capacity
      type: numeric
  - speed:
      description: speed
      type: numeric
      osm_source: maxspeed
      osm_behaviour: copy
```

The example below also shows that it is possible to mix fields that will be imported from OSM posted speed and number of lanes, and fields that need to be in the network but should not be imported from OSM, such as link capacities.

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 and that it is not possible yet to import any tagged values from OSM at the moment, and therefore the parameter *osm_source* would have no effect here.

OpenStreetMap

The **OSM** group of parameters has two specifications: **modes** and **all_link_types**.

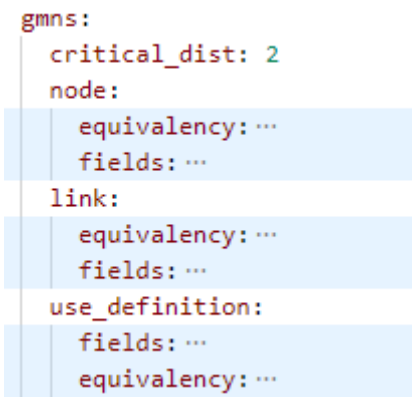
modes contains the list of key tags we will import for each mode. Description of tags can be found on OpenStreetMap Wiki, and we recommend not changing the standard parameters unless you are exactly sure of what you are doing.

For each mode to be imported there is also a mode filter to control for non-default behaviour. For example, in some cities pedestrians a generally allowed on cycleways, but they might be forbidden in specific links, which would be tagged as **pedestrian:no**. This feature is stored under the key *mode_filter* under each mode to be imported.

There is also the possibility that not all keywords for link types for the region being imported, and therefore unknown link type tags are treated as a special case for each mode, and that is controlled by the key *unknown_tags* in the parameters file.

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 Streeet Maps

The OSM section of the parameter file 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.

```
osm:
    overpass_endpoint: "http://overpass-api.de/api"
    nominatim_endpoint: "https://nominatim.openstreetmap.org/"
    accept_language: "en"
    max_attempts: 50
    timeout: 540
    max_query_area_size: 2500000000
    sleeptime: 10
```

The user is also welcome to change the maximum area for a single query to the Overpass API (m^2) and the pause duration between successive requests *sleeptime*.

It is also possible to set a custom address for the Nominatim server, but its use by AequilibraE is so small that it is likely not necessary to do so.

See also

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

2.2.4 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.

The public transport database is created on the run when the `Transit` module is executed for the first time and it can take a little while.

See also

- *aequilibrae.transit.transit.Transit()*
  Class documentation
- *aequilibrae.transit.transit_graph_builder.TransitGraphBuilder()*
  Class documentation

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.

2.2.5 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	INTEGER	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    INTEGER,
     FOREIGN KEY(fare_id) REFERENCES fare_attributes(fare_id) deferrable initially␣
↪deferred,
     FOREIGN KEY(route_id) REFERENCES routes(route_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,
                                        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) ON␣
```

(continues on next page)

(continued from previous page)

```
↪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));

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 ');
```

(continues on next page)

(continued from previous page)

```
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');
```

(continues on next page)

(continued from previous page)

```
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)');
```

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');
```

(continues on next page)

(continued from previous page)

```
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');
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,
                                  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,
     FOREIGN KEY(link) REFERENCES route_links (link) 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');
```

2.2.6 Project Data Components

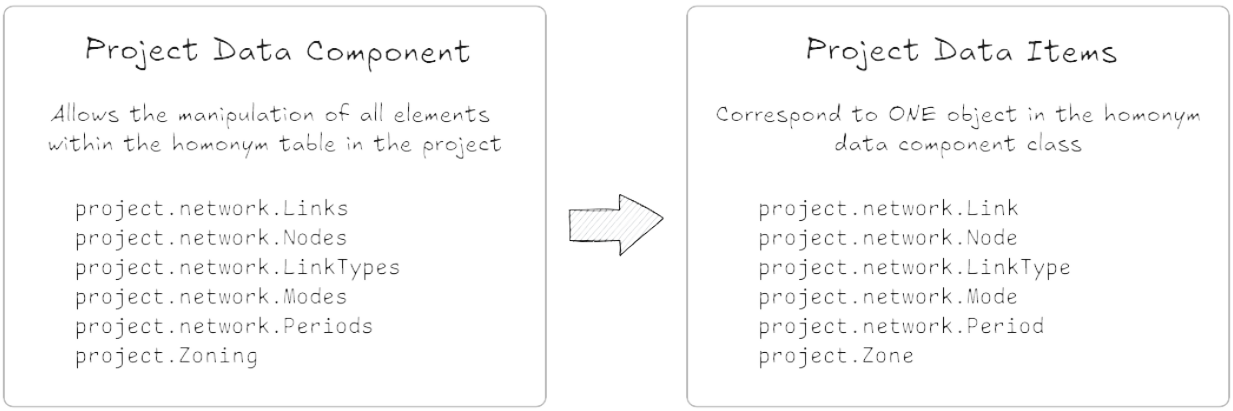
In the *Project structure* section, we present the structure of an AequilibraE project: databases, folders, and parameters. In this section, we present the data components of the project, that is, the data that is presented in the databases.

The components of an AequilibraE project are:

- `project.About`
- `project.FieldEditor`
- `project.Log`
- `project.Matrices`
- `project.Network`
- `project.Zoning`

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

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



Components

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

`project.network.links`

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

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

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

>>> project_links = project.network.links

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

# To save this modification, we must refresh the table
>>> project_links.refresh_fields()

# Let's add a new link to our project
>>> new_link = project_links.new()
>>> new_link.geometry = LineString([(-71.304754, -29.955233), (-71.304863, -29.
↪954049)])
>>> new_link.modes = "bctw"

# To add a new link, it must be explicitly saved
>>> new_link.save()

# The 'links' table has three fields which cannot be empty (i.e. with `NULL` values):
# `link_id`, `direction`, and `modes`. When we create a node, `new` automatically
# creates a `link_id`, and sets the default value (0) for direction. Thus, the modes
# information should be added, otherwise, it will raise an error.

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

# or
>>> project_links.delete(21337)

# The `copy_link` function creates a copy of a specified link
# It is very helpful case you want to split a link.
# You can check out in one of the usage examples.
>>> link_copy = project_links.copy_link(10972)

# Don't forget to save the modifications to the links layer
>>> project_links.save()

# And refresh the links in memory for usage
>>> project_links.refresh()
```

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.network.nodes

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

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

>>> project_nodes = project.network.nodes

# To get one 'Node' object
>>> node = project_nodes.get(10070)

# We can check the existing fields for each node in the 'nodes' table
>>> node.data_fields()
['node_id', 'is_centroid', 'modes', 'link_types', 'geometry', 'osm_id']

# Let's renumber this node and save it
>>> node.renumber(1000)
>>> node.save()

# A node can also be used to add a special generator
# `new_centroid` returns a `Node` object that we can edit
>>> centroid = project_nodes.new_centroid(2000)

# Don't forget to add a geometry to your centroid if it's a new node
# This centroid corresponds to the Port of Coquimbo!
>>> centroid.geometry = Point(-71.32, -29.94)

# 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.
>>> centroid.connect_mode(area=centroid.geometry.buffer(0.01), mode_id="c")

# Don't forget to update these changes to the nodes in memory
>>> project_nodes.refresh()

# And save them into your project
>>> project_nodes.save()

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

(continues on next page)

(continued from previous page)

```
>>> # 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.zoning

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

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

>>> project_zones = project.zoning

# 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
>>> 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
>>> zone.disconnect_mode("t")

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

# And adding the number of public parking spots in the field we just created
```

(continues on next page)

(continued from previous page)

```
>>> zone.parking_spots = 30

# You can save this changes if you want
>>> zone.save()

# 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' objects in the model
>>> project_zones.all_zones()
{1: ..., ..., 133: ...}

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

# 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 = project_zones.new(38)
>>> new_zone.geometry = zone_extent

# We can add a centroid to the zone we just created by specifying its location or
# pass `None` to use the geometric center of the zone
>>> new_zone.add_centroid(Point(-71.33, -29.95))

# Let's refresh our fields
>>> project_zones.refresh_geo_index()

# And save the new changes in the project
>>> project_zones.save()

# 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.zoning.Zoning()***
  Class documentation
- *Create a zone system based on Hex Bins*
  Usage example

project.about

This class provides an interface for editing the 'about' table of a project. We can add new fields or edit the existing ones as necessary, but every time you add or modify a field, you have to write back this information, otherwise it will be lost.

```
>>> project = Project()
>>> project.open("/tmp/accessing_sfalls_data")

>>> project.about.add_info_field("my_new_field")
>>> project.about.my_new_field = "add some useful information about the field"

# We can add data to an existing field
>>> project.about.author = "Your Name"

# And save our modifications
>>> project.about.write_back()

# To assert if 'my_new_field' was added to the 'about' table, we can check the␣
↪characteristics
# stored in the table by returning a list with all characteristics in the 'about'␣
↪table
>>> project.about.list_fields()
['model_name', ..., 'my_new_field']

# The 'about' table is created automatically when a project is created, but if you're
# loading a project created with an older AequilibraE version that didn't contain it,
# it is possible to create one too.
>>> project.about.create()

>>> 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, through the addition/deletion of 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 = Project()
>>> project.open("/tmp/accessing_nauru_data")
```

(continues on next page)

(continued from previous page)

```
# We'll edit the fields in the 'nodes' table
>>> node_fields = project.network.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")

# Don't forget to save these modifications
>>> node_fields.save()

# To edit the description of a field
>>> node_fields.osm_id = "number of the osm node_id"

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

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

>>> project.close()
```

All field descriptions are kept in the table 'attributes_documentation'.

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 = Project()
>>> project.open("/tmp/accessing_nauru_data")

>>> 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.log.Log()***
  Class documentation
- ***Checking AequilibraE's log***
  Usage example

project.matrices

This method is a gateway to all the matrices available in the model, which allows us to update the records in the 'matrices' table. Each item in the 'matrices' table is a `MatrixRecord` object.

```
>>> project = Project()
>>> project.open("/tmp/accessing_sfalls_data")

>>> matrices = project.matrices

# One can also check all the project matrices as a Pandas' DataFrame
>>> matrices.list()

# We can add a naw matrix
>>> matrices.new_record()

# To delete a matrix from the 'matrices' table, we can delete the record directly
>>> matrices.delete_record("demand_mc")

# or by selecting the matrix and deleting it
>>> mat_record = matrices.get_record("demand_mc")
>>> mat_record.delete()

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

# If a matrix was added or deleted by an external process, you should update or clean
# your 'matrices' table to keep your project organised.
>>> matrices.update_database()  # in case of addition

>>> matrices.clear_database()  # in case of deletion

# To reload the existing matrices in memory once again
>>> matrices.reload()

>>> project.close()
```

See also

- ***aequilibrae.project.data.matrices.Matrices()***
  Class documentation

- *Matrices table*
  Table documentation

project.network.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 a `LinkType` object.

```
>>> project = Project()
>>> project.open("/tmp/accessing_coquimbo_data")

>>> link_types = project.network.link_types

>>> new_link_type = link_types.new("A")  # Create a new LinkType with ID 'A'

# We can add information to the LinkType we just created
>>> new_link_type.description = "This is a description"
>>> new_link_type.speed = 35
>>> new_link_type.link_type = "Arterial"

# To save the modifications for `new_link_type`
>>> new_link_type.save()

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

# And don't forget to save the modifications you did in the 'link_types' table
>>> link_types.save()

# To check all `LinkTypes` in the project as a dictionary whose keys are the `link_
↪type_id`'s
>>> link_types.all_types()
{'z': <aequilibrae.project.network.link_type.LinkType object at 0x...>}

# 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.network.modes

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

```
>>> project = Project()
>>> project.open("/tmp/accessing_coquimbo_data")

>>> modes = project.network.modes

# We create a new mode
>>> new_mode = modes.new("k")
>>> new_mode.mode_name = "flying_car"

# And add it to the modes table
>>> modes.add(new_mode)

# When we add a new mode to the 'modes' table, it is automatically saved in the table
# But we can continue editing the modes, and save them as we modify them
>>> new_mode.description = "Like the one in the cartoons"
>>> new_mode.save()

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

# To check all `Modes` in the project as a dictionary whose keys are the `mode_id`'s
>>> modes.all_modes()
{'b': <aequilibrae.project.network.mode.Mode object at 0x...>}

# 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")

>>> project.close()
```

See also

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

project.network.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 = Project()
>>> project.open("/tmp/accessing_coquimbo_data")

>>> periods = project.network.periods

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

# To save this modification, we must refresh the table
>>> periods.refresh_fields()

# Let's get our default period and change the description for our new field
>>> select_period = periods.get(1)
>>> select_period.my_field = "hello world"

# And we save this period modification
>>> select_period.save()

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

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

# It is also possible to renumber a period
>>> new_period.renumber(9)

# And check the existing data fields for each period
>>> new_period.data_fields()
['period_id', 'period_start', 'period_end', 'period_description', 'my_field']

# Saving can be done after finishing all modifications in the table but for the sake
# of this example, we'll save the addition of a new period to our table right away
>>> periods.save()

>>> project.close()
```

See also

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

AequilibraE Matrix

The AequilibraEMatrix class is the AequilibraE vehicle to all things matrices, and in the following sections we'll cover the main points regarding them.

AequilibraeMatrix

This class allows the creation of a memory instance for a matrix, that can be used to load an existing matrix to the project, or to create a new one.

There are three ways of creating an `AequilibraeMatrix`:

- from an OMX file;
- from a trip list, which is nothing more than a CSV file containing the origins, destinations, and trip cores;
- from an empty matrix. In this case, the data type must be one of the following NumPy data types: `np.int32`, `np.int64`, `np.float32`, `np.float64`.

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> num_zones = 5
>>> index = np.arange(1, 6, dtype=np.int32)
>>> mtx = np.ones((5, 5), dtype=np.float32)
>>> names = ["only_ones"]

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=num_zones, matrix_names=names) #memory_only parameter␣
↪defaults to True

# `memory_only` parameter can be changed to `False` case you want to save the matrix␣
↪in disk.
    # This would, however, result in a file format that is being deprecated and will␣
↪not be available on AequilibraE 2.0

# Adds the matrix indexes, which are going to be used for computation
>>> mat.index[:] = index[:]

# Adds the matricial data stored in `mtx` to a matrix named "only_ones"
>>> mat.matrix["only_ones"][:,:] = mtx[:,:]
```

The following methods allow you to check the data in you AequilibraE matrix.

```
>>> mat.cores # displays the number of cores in the matrix
1

>>> mat.names # displays the names of the matrices
```

(continues on next page)

(continued from previous page)

```
['only_ones']

>>> mat.index # displays the IDs of the indexes
array([1, 2, 3, 4, 5])

# To return an array with the selected matrix data
>>> mat.get_matrix("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.]])
```

More than dealing with stored project data, AequilibraE matrices are objects necessary to run procedures, such as traffic assignment. Since a matrix object can hold multiple matrices (i.e. _matrix cores_), it is necessary to specify which matrices will be used in computation, dubbed a computational view in AequilibraE, which sets matrix data in memory in a way it can be used in parallelized algorithms.

Case you're using matricial data from an OMX file, this step also loads the data to memory.

```
>>> mat.computational_view(["only_ones"])
```

You can also export AequilibraE matrices, with your chosen set of _matrix cores_, to different file formats, such as CSV and OMX.

```
>>> mat.export(Path(my_folder_path) / 'my_new_omx_file.omx')

>>> mat.export(Path(my_folder_path) / 'my_new_csv_file.csv')
```

To avoid inconsistencies, once open, the same AequilibraE matrix can only be used once at a time in different procedures. To do so, you have to close the matrix.

```
>>> mat.close()
```

AequilibraE matrices in disk can be reused and loaded once again.

```
>>> mat = AequilibraeMatrix()
>>> mat.load(Path(my_folder_path) / 'my_new_omx_file.omx')

>>> mat.get_matrix("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.]])
```

See also

aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix()
Class documentatiom

Traffic Assignment without an AequilibraE Model
Usage example

OpenMatrix (OMX)

AequilibraE uses OMX files as a standard format for storing its matrices. If you're wondering what is OMX and what does it stand 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 `omx-python` library. In its project page, you can find a brief tutorial to OMX, and better understand how it works.

2.2.7 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.project.Project.upgrade()*

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

from aequilibrae.project.tools import MigrationManager
from aequilibrae.utils.create_example 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()
```

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.utils.create_example import create_example
```

```
from aequilibrae.utils.spatialite_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()
```

Logging to terminal

In this example, we show how to make all log messages show in the terminal.

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
import logging
import sys
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr)
logger = project.logger
```

With the project open, we can tell the logger to direct all messages to the terminal as well

```
stdout_handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s;%(levelname)s ; %(message)s")
stdout_handler.setFormatter(formatter)
logger.addHandler(stdout_handler)
```

```
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.project.Project.db_connection()*
- *aequilibrae.project.project.Project.results_connection()*
- *aequilibrae.project.project.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.utils.create_example 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:
    matrices = pd.read_sql("SELECT * FROM matrices", conn)
```

```
matrices
```

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.utils.create_example import create_example
from aequilibrae.paths import TrafficAssignment, TrafficClass
```

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

We build our graphs

```
project.network.build_graphs()

graph = project.network.graphs["c"]
graph.set_graph("free_flow_time")
graph.set_skimming(["free_flow_time", "distance"])
graph.set_blocked_centroid_flows(False)
```

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

```
proj_matrices = project.matrices
demand = proj_matrices.get_matrix("demand_omx")
demand.computational_view(["matrix"])
```

Now let's perform our traffic assignment

```
assig = TrafficAssignment()

assigclass = TrafficClass(name="car", graph=graph, matrix=demand)

assig.add_class(assigclass)
assig.set_vdf("BPR")
assig.set_vdf_parameters({"alpha": 0.15, "beta": 4.0})
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")
assig.set_algorithm("bfw")
assig.max_iter = 50
assig.rgap_target = 0.001

assig.execute()
```

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

In lines 1-7, we receive some warnings that our fields name and lane have `NaN` values. As they are not relevant to our example, we can move on.

In lines 8-9 we get the Traffic Class specifications. We can see that there is only one traffic class (car). Its **graph** key presents information on blocked flow through centroids, number of centroids, links, and nodes. In the **matrix** key, we find information on where in the disk the matrix file is located. We also have information on the number of centroids and nodes, as well as on the matrix/matrices used for computation. In our example, we only have one matrix named matrix, and the total sum of this matrix element is equal to 360,600. If you have more than one matrix its data will be also displayed in the *matrix_cores* and *matrix_totals* keys.

In lines 10-11 the log 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.

Line 12 displays us a warning to indicate that AequilibraE is converting the data type of the cost field.

Lines 13-61 indicate that we'll receive the outputs of a *bfw* algorithm. In the log there are also the number of the iteration, its relative gap, and the stepsize. The outputs in lines 15-60 are exactly the same as the ones provided by the function `assig.report()`. Finally, the last line shows us that the *bfw* assignment has finished after 46 iterations because its gap is smaller than the threshold we configured (0.001).

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

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

from aequilibrae.utils.create_example import create_example
from aequilibrae import TrafficAssignment, TrafficClass
```

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

```
# Build the network graph
project.network.build_graphs(fields=["distance", "capacity_ab", "capacity_ba"],␣
↪modes=["c"])
graph = project.network.graphs["c"]
graph.set_graph("distance")
graph.set_blocked_centroid_flows(False)

# Get the demand matrix
mat = project.matrices.get_matrix("demand_omx")
mat.computational_view()

# Create traffic assignment with alternative parameters
assigclass = TrafficClass("car", graph, mat)
assignment = TrafficAssignment(project)
assignment.add_class(assigclass)
assignment.set_vdf("BPR")

assignment.set_vdf_parameters({"alpha": 0.15, "beta": 4.0})
assignment.set_capacity_field("capacity")
assignment.set_time_field("distance")
assignment.max_iter = 10
assignment.set_algorithm("msa")

assignment.execute()

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

print(f"Assignment completed. Total flow: {assigclass.results.total_link_loads.sum():.
↪2f}")
```

Switch to empty scenario for modifications

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

(continues on next page)

(continued from previous page)

```
# 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.network.links.data)} links")

# 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.network.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
mat.close()
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`

The run module is a standard Python module that is dynamically imported when the `project.run` property is accessed. Objects named within `parameters.yml` under the `run` heading will have their arguments partially applied via `functools.partial` and return a `namedtuple`.

Not all objects within the module must be named `parameters.yml`. If an object is named within `parameters.yml`, then it must exist within the module otherwise a `RuntimeError` will be raised.

By default an AequilibraE project comes with four example functions: `matrix_summary`, `graph_summary`, `results_summary`, and `example_function_with_kwargs`. The summary functions are not named within the default `parameters.yml` as they take no arguments.

Functions should use the `get_active_project()` function to obtain a reference to the current project.

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.project.Project.run*

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

from aequilibrae.parameters import Parameters
from aequilibrae.utils.create_example 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.network
network.build_graphs(modes=[mode])
graph = network.graphs[mode]
graph.set_graph("distance")
graph.set_skimming("distance")
graph.set_blocked_centroid_flows(False)
```

With the method `graph_summary()`, we can check the total number of links, nodes, and zones, as well as the compact number of links and nodes used for computation. If we had more than one graph, its information would be displayed within the nested dictionary.

```
project.run.graph_summary()
```

If no graphs have been built, an empty dictionary will be returned.

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(source: str, name: str, computational_view: str, result_name: str,
↪ overwrite: bool=False):\n
\tfrom aequilibrae.utils.create_delaunay_network import DelaunayAnalysis\n
\tproject = get_active_project()\n
\tmatrix = project.matrices\n
\tmat = matrix.get_matrix(name)\n
\tmat.computational_view(computational_view)\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 = 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"]["computational_view"] = "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

Currently AequilibraE can import links and nodes from a network from OpenStreetMaps, GMNS, and from link layers. AequilibraE can also export the existing network into GMNS format. There is some valuable information on these topics in the following sections.

4.1.1 Importing from OpenStreetMap

You can check more specifications on OSM download on the *Parameters YAML File*.

Note

All links that cannot be imported due to errors in the SQL insert statements are written to the log file with error message AND the SQL statement itself, and therefore errors in import can be analyzed for re-downloading or fixed by re-running the failed SQL statements after manual fixing.

Python limitations

As it happens in other cases, Python's usual implementation of SQLite is incomplete, and does not include R-Tree, a key extension used by SpatiaLite for GIS operations.

If you want to learn a little more about this topic, you can access this blog post or check out the SQLite page on R-Tree.

This limitation issue is solved when installing SpatiaLite, as shown in *the dependencies page*.

Please also note that AequilibraE's network consistency triggers **will NOT work** before spatial indices have been created and/or if the editing is being done on a platform that does not support both R-Tree and SpatiaLite.

See also

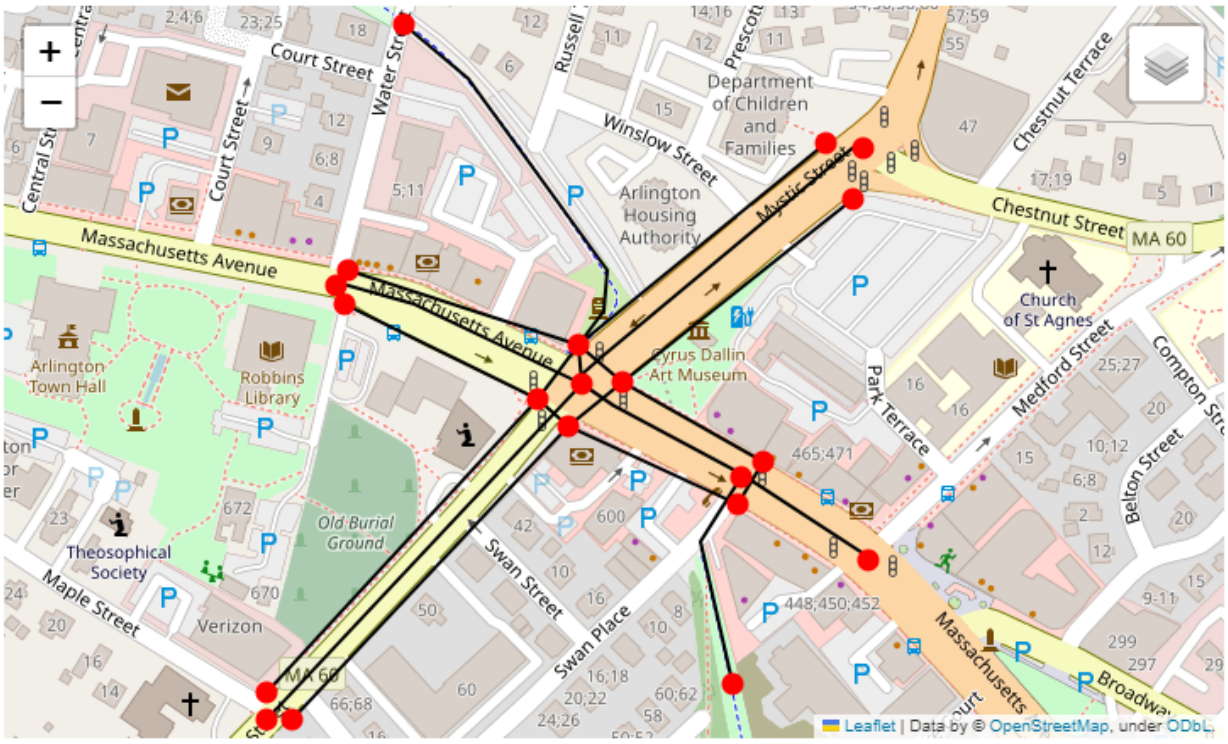
- ***aequilibrae.project.network.network.Network.create_from_osm()***
  Function documentation
- ***Create project from OpenStreetMap***
  Usage example

4.1.2 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*.

4.1.3 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, the method `create_from_gmns()` read 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

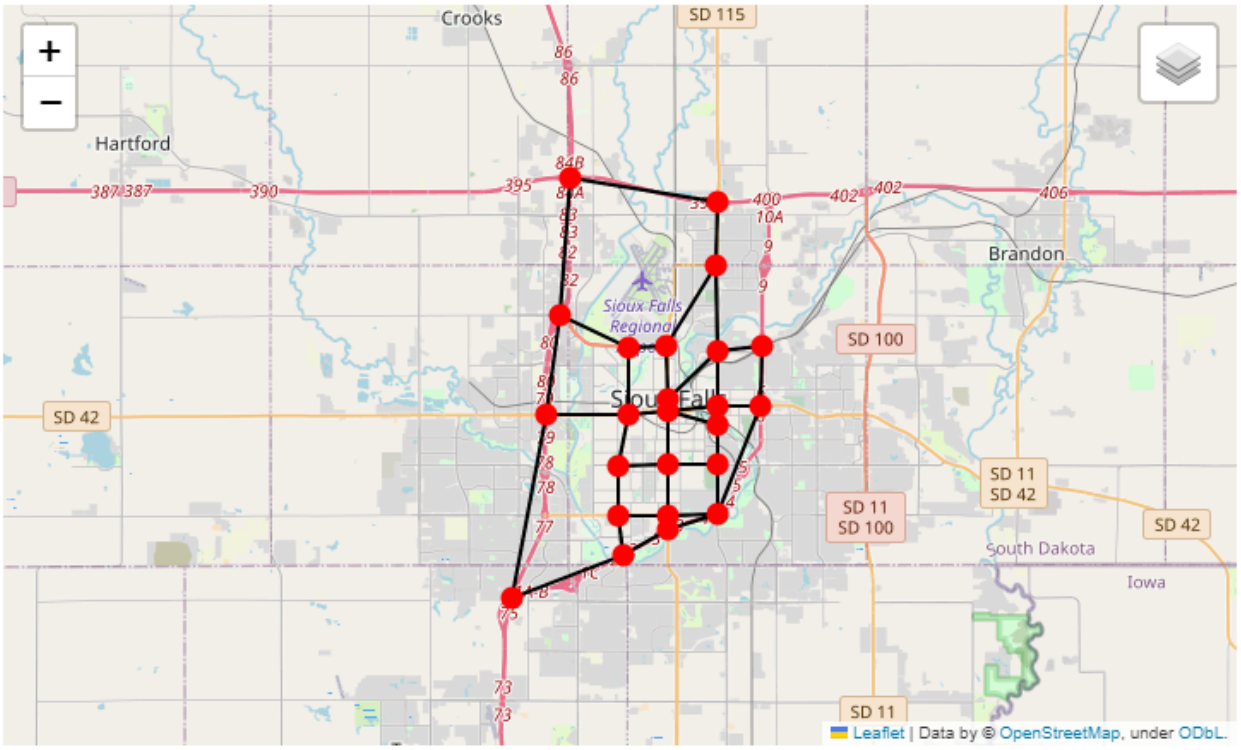
In the AequilibraE nodes table, if a node is to be identified as a centroid, its 'is_centroid' field has to be set to 1. However, this is not part of the GMNS specification. Thus, if you want a node to be identified as a centroid during the import process, in the GMNS node table you have to set the field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

See also

- ***aequilibrae.project.network.network.Network.create_from_gmns()***
  Function documentation
- *Create project from GMNS*
  Usage example

4.1.4 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

- ***aequilibrae.project.network.network.Network.export_to_gmns()***
  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.

4.3 Examples

4.3.1 Network Manipulation

Create project from OpenStreetMap

In this example, we show how to create an empty project and populate it with a network from OpenStreetMap.

This time we will use GeoPandas to visualize the network.

References

- *Importing from OpenStreetMap*

See also

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

- *aequilibrae.project.network.network.Network.create_from_osm()*

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

from aequilibrae import Project
```

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

project = Project()
project.new(fldr)
```

Now we can download the network from any place in the world (as long as you have memory for all the download and data wrangling that will be done).

We can create from a bounding box or a named place. For the sake of this example, we will choose the small nation of Nauru.

```
project.network.create_from_osm(place_name="Nauru")
```

We can also choose to create a model from a polygon (which must be in `EPSG:4326`) or from a Polygon defined by a bounding box, for example.

```
# project.network.create_from_osm(model_area=box(-112.185, 36.59, -112.179, 36.60))
```

We grab all the links data as a geopandas GeoDataFrame so we can process it easier

```
links = project.network.links.data
```

Let's plot our network!

```
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.utils.create_example 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.network.nodes
links = project.network.links
node = all_nodes.get(1)
new_geo = Point(node.geometry.x + 0.02, node.geometry.y - 0.02)
node.geometry = new_geo

# We can save changes for all nodes we have edited so far.
node.save()
```

If you want to show the path in Python.

We do NOT recommend this, though…. It is very slow for real networks.

```
# Let's refresh the links in memory for usage
links.refresh()
```

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()

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()
```

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 (`create_from_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:

- *aequilibrae.project.network.network.Network.export_to_gmns()*

```
# Imports
from uuid import uuid4
import os
from tempfile import gettempdir

from aequilibrae.utils.create_example 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.network.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()
```

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.utils.create_example 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.network.nodes
links = project.network.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)
link.geometry = LineString([node.geometry, new_extremity])

# and the link that goes from node 2 to node 1
link = links.get(3)
node2 = all_nodes.get(2)
link.geometry = LineString([new_extremity, node2.geometry])

# We save the changes and refresh the links in memory for usage
links.save()
links.refresh()
```

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()
```

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.utils.create_example 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.network.links.data
nodes = project.network.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.utils.create_example 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.network.links
all_nodes = project.network.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 = links.copy_link(37)

first_geometry = substring(link.geometry, 0, 0.5, normalized=True)
second_geometry = substring(link.geometry, 0.5, 1, normalized=True)

link.geometry = first_geometry
new_link.geometry = second_geometry
links.save()
```

The link objects in memory still don't have their ID fields updated, so we refresh them.

```
links.refresh()

link = links.get(37)
new_link = links.get(new_link.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.zoning()*

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

from aequilibrae.utils.create_example 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)
```

```
zoning = project.zoning
```

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.

```
zoning.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.

```
zoning.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.network.links.data
centroids = links[links["link_type"] == "centroid_connector"]
links = links[links["link_type"] != "centroid_connector"]

nodes = project.network.nodes.data
nodes = nodes[nodes["is_centroid"] == 1]
```

```
map = folium.Map(location=[-29.9568, -71.3456], zoom_start=14)
zoning.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:

- *aequilibrae.project.network.network.Network.create_from_gmns()*

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

(continues on next page)

(continued from previous page)

```
from tempfile import gettempdir

from aequilibrae import Project
from aequilibrae.parameters import Parameters
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()
project.new(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 = Parameters()
par.parameters["network"]["gmns"]["link"]["fields"].update(new_link_fields)
par.parameters["network"]["gmns"]["node"]["fields"].update(new_node_fields)
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.network.create_from_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.network.links.data
nodes = project.network.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.paths.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.utils.create_example import create_example
from aequilibrae.paths.results import PathResults
```

```
# 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")

# Let's analyze the mode car or 'c' in our model
mode = "c"
```

We need to create the graph, but before that, we need to have at least one centroid in our network.

```
# We get an arbitrary node to set as centroid and allow for the construction of graphs
nodes = project.network.nodes
centroid_count = nodes.data.query('is_centroid == 1').shape[0]

if centroid_count == 0:
    arbitrary_node = nodes.data["node_id"][0]
    nd = nodes.get(arbitrary_node)
    nd.is_centroid = 1
    nd.save()
```

(continues on next page)

(continued from previous page)

```
network = project.network
network.build_graphs(modes=[mode])
graph = network.graphs[mode]
graph.set_blocked_centroid_flows(False)

if centroid_count == 0:
    # Let's revert to setting up that node as centroid in case we had to do it

    nd.is_centroid = 0
    nd.save()
```

We set the graph for computation

```
graph.set_graph("distance")
graph.set_skimming("distance")
```

Get the nodes that are part of the car network

```
missing_nodes = nodes.data.query("modes.str.contains(@mode)")["node_id"].values
```

And prepare the path computation structure

```
res = PathResults()
res.prepare(graph)
```

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.reset()
    res.compute_path(missing_nodes[0], missing_nodes[1])
    res.predecessors[graph.nodes_to_indices[missing_nodes[0]]] = 0
    connected = graph.all_nodes[np.where(res.predecessors >= 0)]
    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
from uuid import uuid4
import urllib.request
from string import ascii_lowercase
from tempfile import gettempdir
from os.path import join
from shapely.wkt import loads as load_wkt
import pandas as pd

from aequilibrae import Project
```

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

project = Project()
project.new(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)
```

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.network.link_types
lt_dict = lt.all_types()
existing_types = [ltype.link_type for ltype in lt_dict.values()]
```

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):
    new_type = lt.new(ascii_lowercase[i])
    new_type.link_type = ltype
    # new_type.description = 'Your custom description here if you have one'
    new_type.save()
```

We need to use a similar process for modes

```
md = project.network.modes
md_dict = md.all_modes()
existing_modes = {k: v.mode_name for k, v in md_dict.items()}
```

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 i, mode_id in enumerate(modes_to_add):
    new_mode = md.new(mode_id)
    # You would need to figure out the right name for each one, but this will do
    new_mode.mode_name = f"Mode_from_original_data_{mode_id}"
    # new_type.description = 'Your custom description here if you have one'

    # It is a little different because you need to add it to the project
    project.network.modes.add(new_mode)
    new_mode.save()
```

We cannot use the existing link_id, so we create a new field to not loose this information

```
links = project.network.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 need to refresh the fields so the adding method can see it
links.refresh_fields()
```

We can now add all links to the project!

```
for idx, record in df.iterrows():
    new_link = links.new()

    # Now let's add all the fields we had
    new_link.source_id = record.link_id
    new_link.direction = record.direction
    new_link.modes = record.modes
    new_link.link_type = record.link_type
    new_link.name = record.name
    new_link.geometry = load_wkt(record.WKT)
    new_link.save()
```

We grab all the links data as a geopandas GeoDataFrame so we can process it easier

```
links = project.network.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
import branca
import folium
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
```

(continues on next page)

(continued from previous page)

```
from aequilibrae.utils.create_example import create_example
from aequilibrae.project.tools.network_simplifier import NetworkSimplifier
```

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 graph. As Nauru doesn't have any centroid in its network we have to create a centroid from an arbitrary node, otherwise we cannot create a graph.

```
nodes = project.network.nodes
centroid_count = nodes.data.query("is_centroid == 1").shape[0]

if centroid_count == 0:
    arbitrary_node = nodes.data["node_id"][0]
    nd = nodes.get(arbitrary_node)
    nd.is_centroid = 1
    nd.save()
```

```
# Let's analyze the mode car or 'c' in our model
mode = "c"
```

```
# Let's set the graph for computation
network = project.network
network.build_graphs(modes=[mode])
graph = network.graphs[mode]
graph.set_graph("distance")
graph.set_skimming("distance")
graph.set_blocked_centroid_flows(False)
```

```
# Let's revert to setting up that node as centroid in case we had to do it
if centroid_count == 0:
    nd.is_centroid = 0
    nd.save()
```

We check the number of links and nodes our project has initially.

```
links_before = project.network.links.data
nodes_before = project.network.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()
```

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)
net.rebuild_network()
```

Let's plot the previous and actual networks!

```
links_after = net.network.links.data
nodes_after = net.network.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)

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])
net.rebuild_network()
```

Let's plot the network once again and check the modifications!

```
links_after = net.network.links.data
nodes_after = net.network.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")
```

(continues on next page)

(continued from previous page)

```
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.

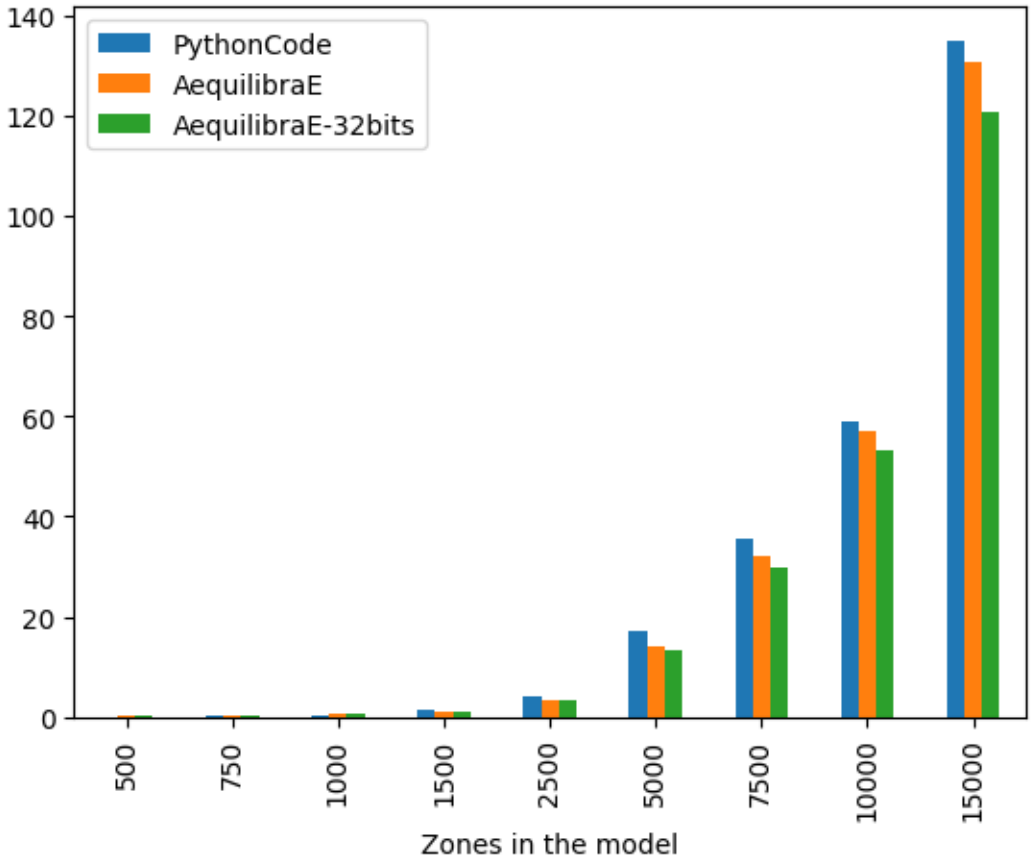


Fig. 1: AequilibraE's IPF runtime

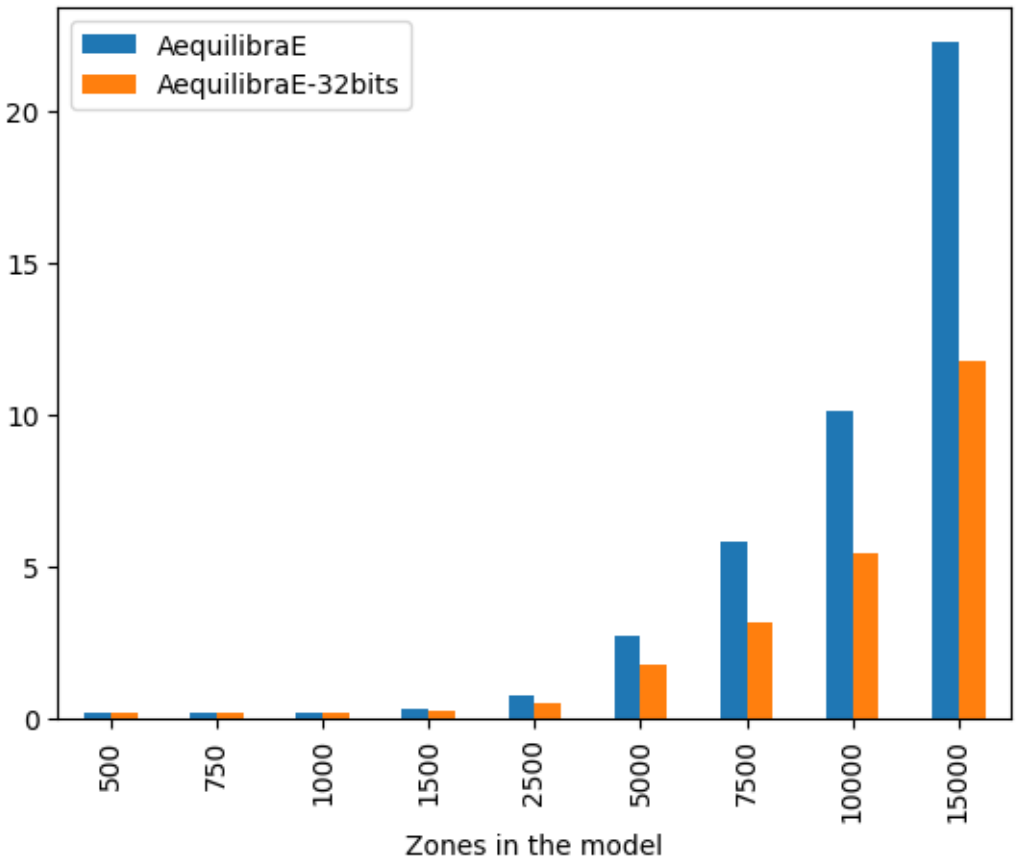


Fig. 2: AequilibraE's IPF runtime 32 vs 64 bits

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.

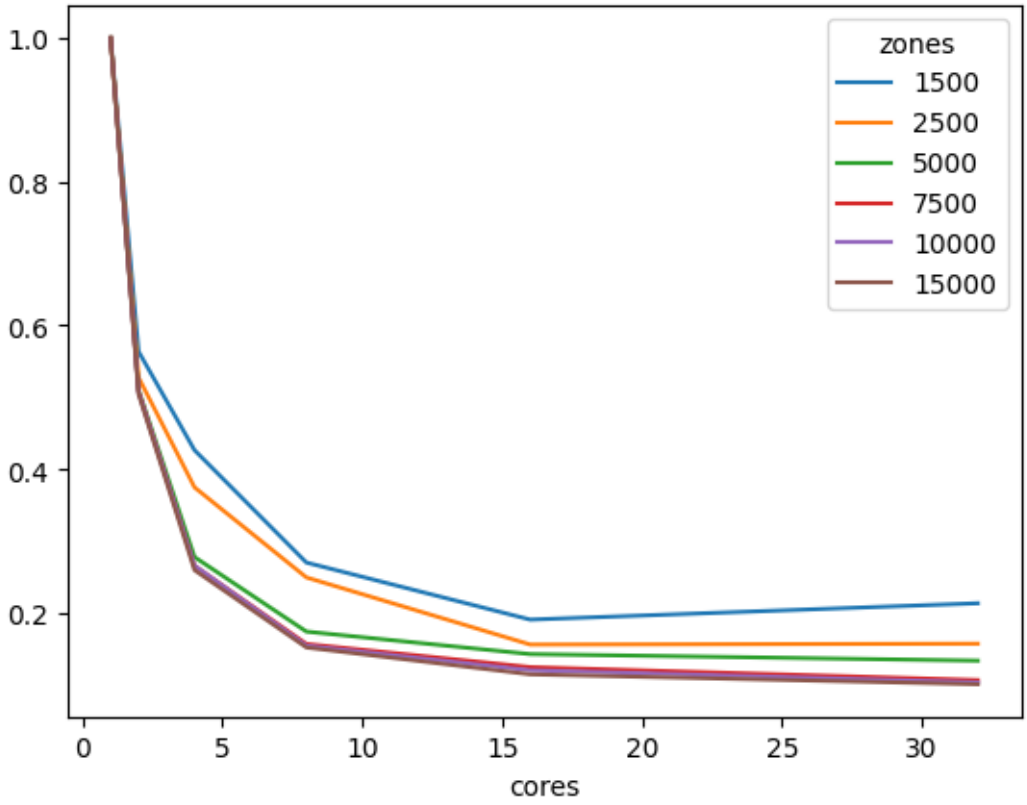


Fig. 3: number of cores used in IPF for 64 bit matrices

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
```

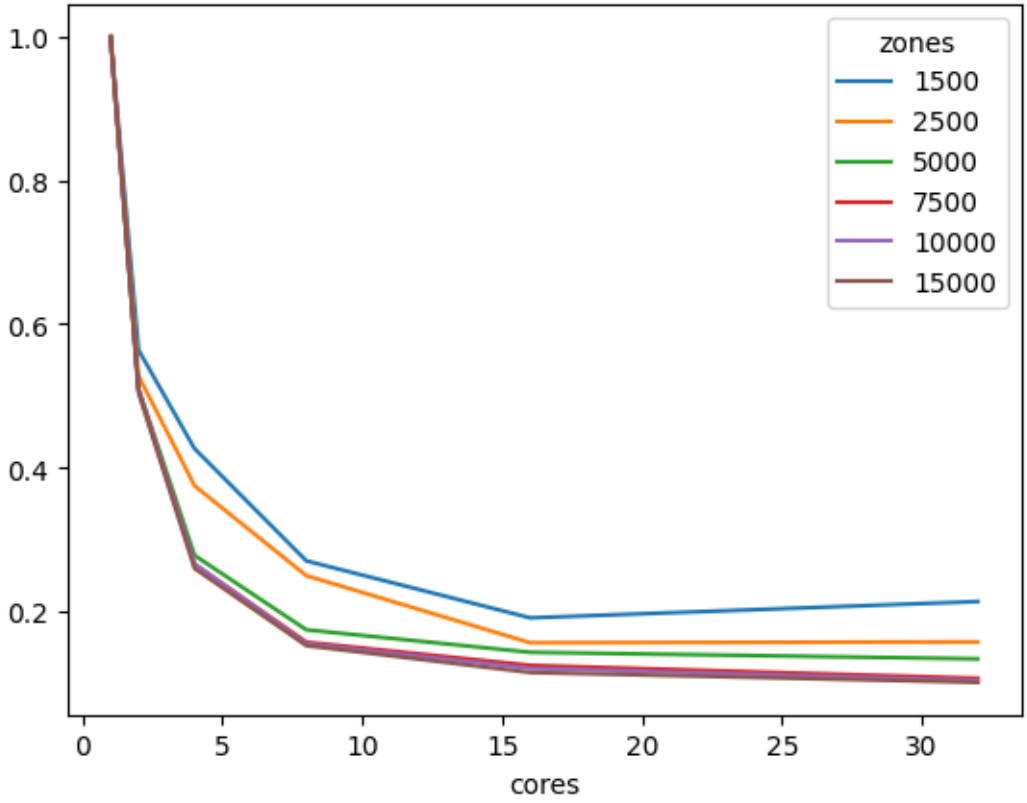


Fig. 4: number of cores used in IPF for 32 bit matrices

```
# 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)

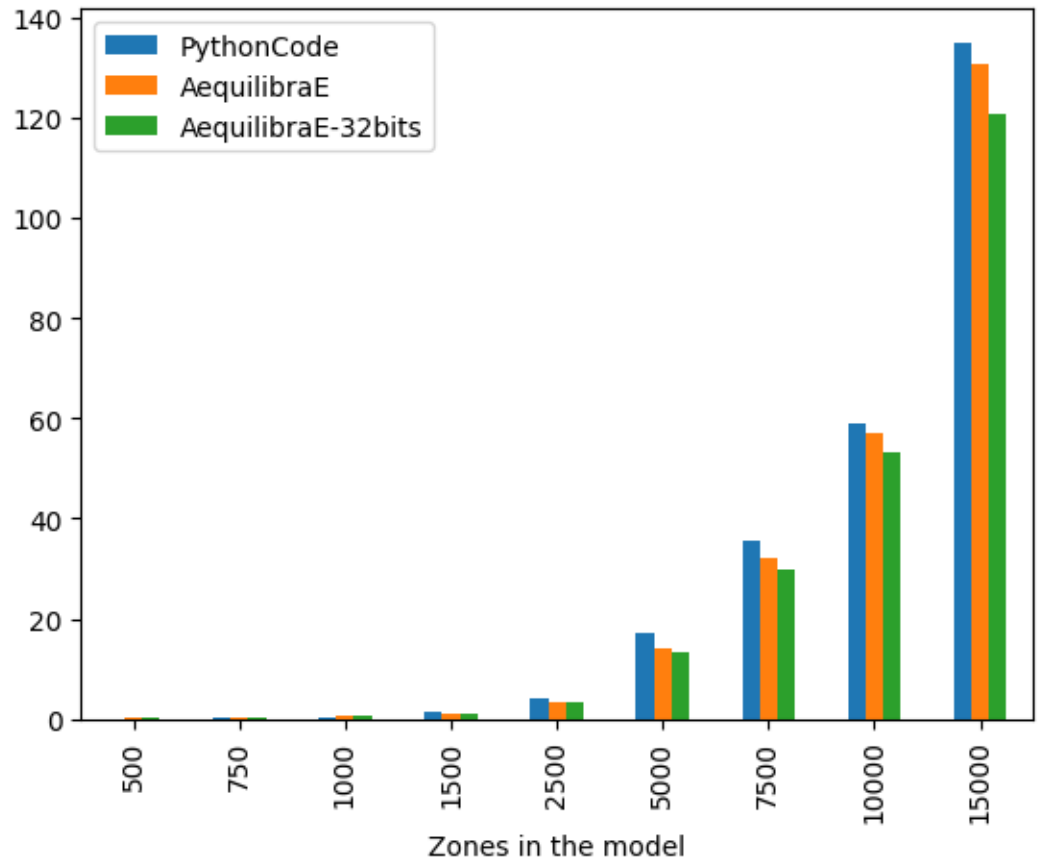
(continued from previous page)

```
        # 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)

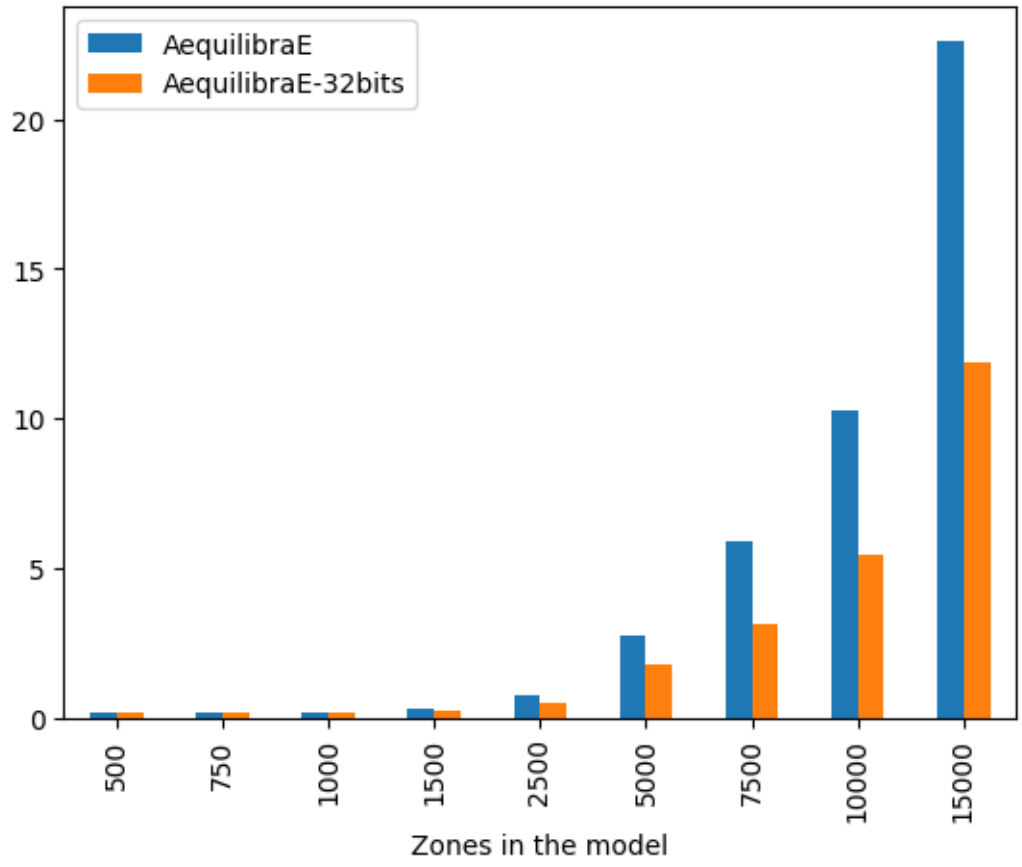
(continued from previous page)

```
↪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'>
```



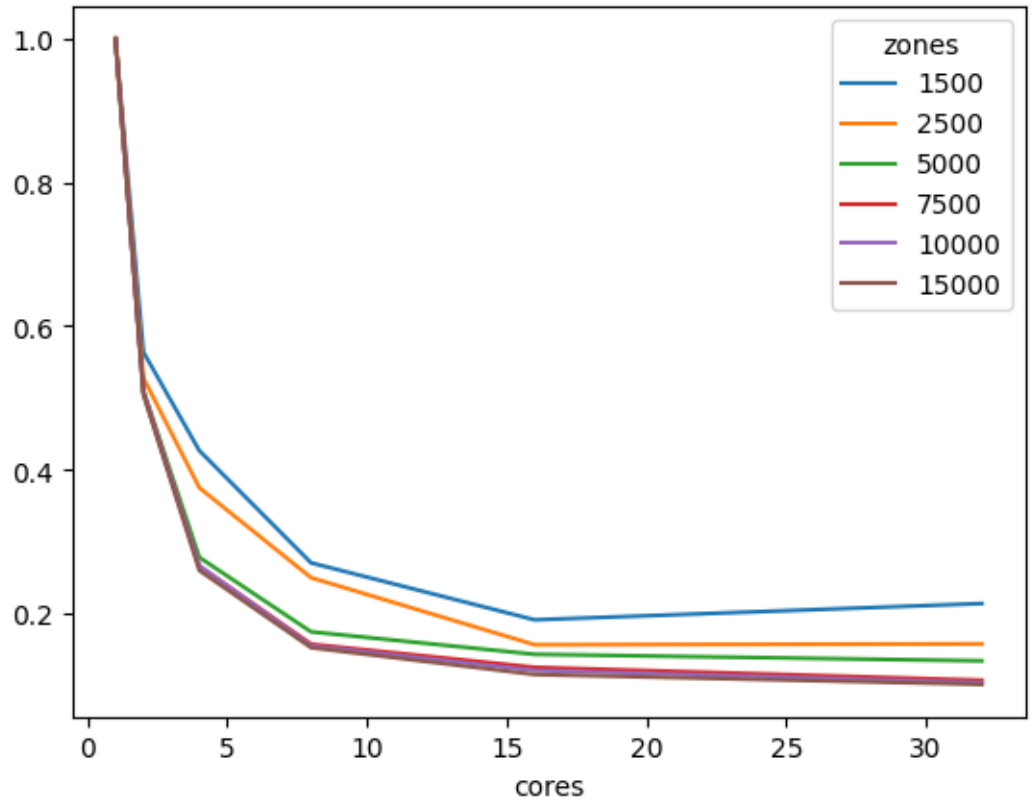

```
bench_df_parallel.groupby(["Zones in the model"]).mean()
```

```
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):
```

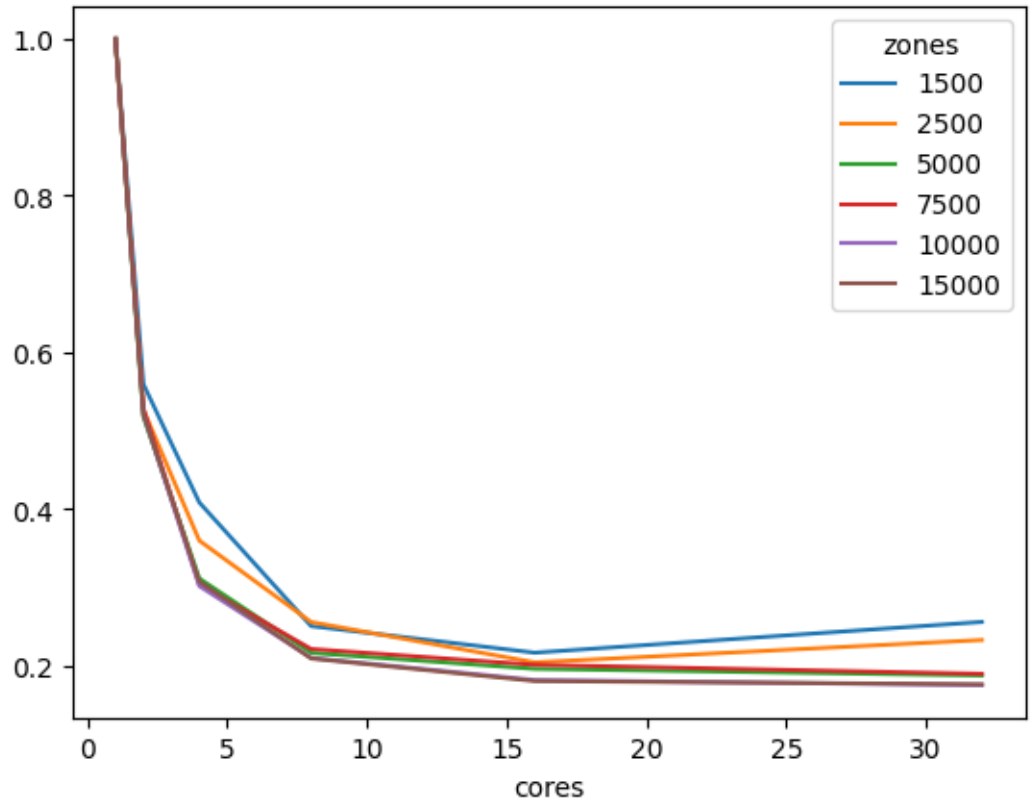
(continues on next page)

(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]
```

```
mtx = AequilibraeMatrix()
mtx.create_empty(file_name=join(folder, "matrix.aem"), zones=num_zones)
mtx.index[:] = np.arange(1, num_zones + 1)[:]
mtx.matrices[:, :, 0] = matrix[:]
mtx.computational_view()
```

```
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_view
```

```
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, all path computation in AequilibraE happens through `Graph` objects. While users can operate models by simply selecting the mode they want AequilibraE to create graphs for, `Graph` objects can also be manipulated in memory or even created from networks 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.paths.results.path_results.PathResults()***
  Class documentation
- ***Path computation***
  Usage example
- ***Network skimming***
  Usage example

6.1 AequilibraE Graphs

The AequilibraE Graph is a computational representation of the network. The Graph object is rather complex, but the difference between the graph and the physical links are the availability of three class member variables consisting of Pandas DataFrames: the *network*, the *graph*, and the *compressed_graph*.

```
>>> from aequilibrae.paths import Graph

>>> g = Graph()

>>> g.network
>>> g.graph
>>> g.compressed_graph
```

6.1.1 The network dataframe

Links in the *network* table (the Pandas representation of the project's *Links* table) are potentially bi-directional, and the directions allowed for traversal are dictated by the field *direction*, where -1 and 1 denote only BA and AB traversal respectively and 0 denotes bi-directionality.

Direction-specific fields must be coded in fields **_AB** and **_BA**, where the name of the field in the graph will be equal to the prefix of the directional fields. For example:

The fields *free_flow_travel_time_AB* and *free_flow_travel_time_BA* provide the same metric (*free_flow_travel_time*) for each of the directions of a link, and the field of the graph used to set computations (e.g. field to minimize during path-finding, skimming, etc.) will be *free_flow_travel_time*.

6.1.2 The graph dataframe

The graph dataframe is a very simple transformation of the network where all links are **directed**. This is achieved by decomposing bi-direction links two different links in the graph, each one representing a direction.

As described above, fields are made uni-directional, with bi-directional fields being transformed into a single field and all other fields remaining the same.

6.1.3 The compressed graph dataframe

The purpose of the compressed graph is to optimize its performance for the more recurrent and time consuming computations in modelling, which are traffic assignment and skimming.

The key characterist we can leverage to optimize performance for these operations is that paths are computed only between centroids, therefore two forms of compression are possible:

1. Topological simplification
2. Dead end removal

Topological simplification consists in creating a topological equivalent of the graph by contracting sequences of links between intersections (extra nodes). These links are often common in long links.

It is important to note that computations that cannot be handled at the compressed level without loss of fidelity, for example the computation of congestion during traffic assignment, is done at the uncompressed (graph) level, but we still extract the full benefits of the compression at the path-computation stage, which is the most time-consuming portion of assignment.

Dead end removal is an additional procedure, that consists in removing dead ends and fish spines from the network. For those not familiar with the term, a fish spine is a road network with multiple smaller streets branching off from it, often leading to dead ends or cul-de-sacs, creating a pattern that resembles a fish spine.

Whilst it's easy for humans to ignore dead ends when planning a route, the same cannot be said for computers. Dead end removal is done based on the observation that in a graph with non-negative weights a dead end will only ever appear in the results of a short(est) path if the origin or destination is present within that dead end.

Dead end removal is applied before topological simplification and results in a distinctive network, topologically speaking. However, all centroids are preserved. More about dead end removal can be found at this blog post.

It should be noted that not all fields are compressed in this process, but only the cost field.

6.1.4 Leveraging topological simplification

Topological simplification is a powerful tool that may want to be leveraged for other purposes. As such, we make it available for users to access it.

For this purpose, the user should be explicit in not removing dead ends from the graph, as that will result in simplification beyond pure topological simplification.

```
>>> graph.prepare_graph(np.array([13, 169, 2197, 28561, 37123], np.int32), remove_
↪dead_ends=False)
```

6.1.5 Graphs from a model

Building graphs directly from an AequilibraE model is the easiest option for beginners or when using AequilibraE in anger, as much of the setup is done by default.

```
>>> project = create_coquimbo_example

>>> project.network.build_graphs() # We build the graph for all modes
>>> graph = project.network.graphs['c'] # we grab the graph for cars
```

6.1.6 Manipulating graphs in memory

The AequilibraE Graph can be manipulated in memory, with all its components available for editing. One of the simple tools available directly in the API is a method call for excluding one or more links from the Graph, **which is done in place**.

```
>>> graph.exclude_links([123, 975])
```

When working with very large networks, it is possible to filter the database to a small area for computation by providing a polygon that delimits the desired area, instead of selecting the links for deletion. The selection of links and nodes is limited to a spatial index search, which is very fast but not accurate.

```
>>> polygon = Polygon([(-71.35, -29.95), (-71.35, -29.90), (-71.30, -29.90), (-71.30,␣
↪-29.95), (-71.35, -29.95)])
>>> project.network.build_graphs(limit_to_area=polygon)
```

More sophisticated graph editing is also possible, but it is recommended that changes to be made in the network DataFrame. For example:

```
# We can add fields to our graph
>>> graph.network["link_type"] = project.network.links.data["link_type"]

# And manipulate them
>>> graph.network.loc[graph.network.link_type == "motorway", "speed_ab"] = 100
>>> graph.network.loc[graph.network.link_type == "motorway", "speed_ba"] = 100
```

6.1.7 Skimming settings

Skimming the field of a graph when computing shortest path or performing traffic assignment must be done by setting the skimming fields in the Graph object, and there are no limits (other than memory) to the number of fields that can be skimmed.

```
>>> graph.set_skimming(["distance", "travel_time"])
```

6.1.8 Setting centroids

Like other elements of the AequilibraE Graph, the user can also manipulate the set of nodes interpreted by the software as centroids in the Graph itself. This brings the advantage of allowing the user to perform assignment of partial matrices, matrices of travel between arbitrary network nodes and to skim the network for an arbitrary number of centroids in parallel, which can be useful when using AequilibraE as part of more general analysis pipelines. As seen above, this is also necessary when the network has been manipulated in memory.

When setting regular network nodes as centroids, the user should take care in not blocking flows through "centroids".

```
>>> graph.prepare_graph(np.array([13, 169, 2197, 28561, 37123], np.int32))
>>> graph.set_blocked_centroid_flows(False)
```

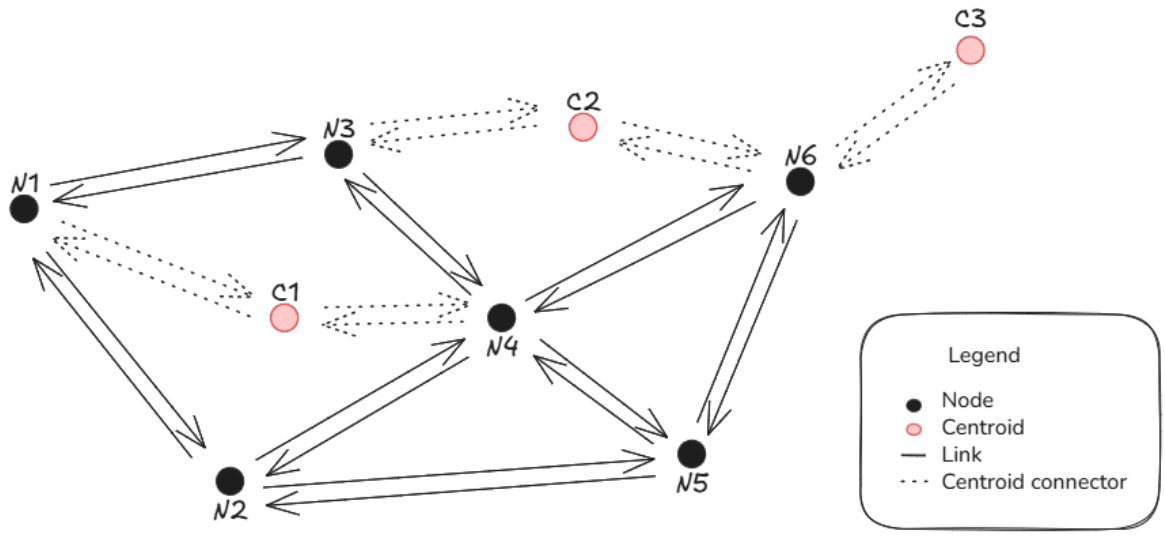
See also

- *aequilibrae.paths.graph.Graph()*
  Class documentation
- *aequilibrae.paths.graph.TransitGraph()*
  Class documentation

6.1.9 Blocking flows through centroids

When using AequilibraE Graph, it is possible to configure if one wants to allow paths through centroids or not. As centroid connectors are a bi-directional link type, in practice what blocking flows through centroids does is 'removing' graph links leaving from the centroid.

Suppose one wants to compute the shortest path between node N1 and centroid C3 in the figure below. An initial path guess would be N1 -> N3 -> C2 -> N6 -> C3 because all links are bi-directional. However, when we block paths through centroids, it is not possible to compute the path between C2 and N6 because we 'removed' the link leaving from the centroid.



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.

We are using Sioux Falls data, from TNTP.

See also

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

- *aequilibrae.paths.graph()*

```
# Imports
import numpy as np
import pandas as pd

from aequilibrae.paths import Graph
```

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 Graph object requires several default 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()
```

Building an AequilibraE graph from our network is pretty straightforward. We assign our network to be the graph's network …

```
graph = Graph()
graph.network = net
```

… and then set the graph's configurations.

```
graph.prepare_graph(np.arange(1, 25))  # sets the centroids for which we will perform␣
↪computation

graph.set_graph("length")  # sets the cost field for path computation

graph.set_skimming(["length", "free_flow_time"]) # sets the skims to be computed

graph.set_blocked_centroid_flows(False)  # we don't block flows through centroids␣
↪because all nodes
                                         # in the Sioux Falls network are centroids
```

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…

```
res.path_nodes
```

… and the path links.

```
res.path
```

For path computation, when we call the method `graph.compute_path(1, 17)`, we are calling the class `PathComputation` and storing its results into a variable.

Notice that other methods related to path computation, such as `milepost` can also be used with `res`.

For skim computation, the process is quite similar. When calligng the method `graph.compute_skims()` we are actually calling the class `NetworkSkimming`, and storing its results into `skm`.

```
skm = graph.compute_skims()
```

Let's get the values for 'free_flow_time' matrix.

```
skims = skm.results.skims
skims.get_matrix("free_flow_time")
```

Now we're all set!

Graph image credits to Behance-network icons created by Sumitsaengtong - Flaticon

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.paths.graph()*
- *aequilibrae.paths.results.path_results()*

Imports

```
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
```

We create the example project inside our temp folder

```
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
import logging
import sys
```

We the project opens, we can tell the logger to direct all messages to the terminal as well

```
logger = project.logger
stdout_handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s;%(levelname)s ; %(message)s")
stdout_handler.setFormatter(formatter)
logger.addHandler(stdout_handler)
```

Path Computation

We build all graphs

```
project.network.build_graphs()
# We get warnings that several fields in the project are filled with ``NaN``s.
# This is true, but we won't use those fields.
```

We grab the graph for cars,

```
graph = project.network.graphs["c"]
```

we'll also see what graphs are available.

```
project.network.graphs.keys()
```

Let's say we want to minimise the distance,

```
graph.set_graph("distance")
```

and will skim time and distance while we are at it.

```
graph.set_skimming(["travel_time", "distance"])
```

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 `PathComputation` class to achieve the same result.

```
# from aequilibrae.paths import PathResults

# res = PathResults()
# res.prepare(graph)
# res.compute_path(32343, 22041)
```

We can get the sequence of nodes we traverse

```
res.path_nodes
```

We can get the link sequence we traverse

```
res.path
```

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 behavior.

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)
```

If you prefer to find a potentially non-optimal path to the destination faster, provide `a_star=True` to use *A** with a heuristic. This method always recomputes the path's nodes, links, skims, and mileposts with `update_trace`. Note that a_star takes precedence over early_exit.

```
res = graph.compute_path(32343, 22041, a_star=True)
```

If you are using *a_star*, it is possible to use different heuristics to compute the path. By default, an equirectangular heuristic is used, and we can view the available heuristics via:

```
res.get_heuristics()
```

If you prefer a more accurate but slower heuristic, you can choose "haversine", by setting:

```
res = graph.compute_path(32343, 22041, a_star=True, heuristic="haversine")
```

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 set the `res.early_exit` and `res.a_star` attributes. Notice that this procedure is much faster when you have large networks.

```
res.a_star = False
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.network.links.data.set_index("link_id")
links = links.loc[res.path]
```

```
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.paths.graph()*
- *aequilibrae.paths.network_skimming()*

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

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
import logging
import sys
```

When the project opens, we can tell the logger to direct all messages to the terminal as well

```
logger = project.logger
stdout_handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s;%(levelname)s ; %(message)s")
stdout_handler.setFormatter(formatter)
logger.addHandler(stdout_handler)
```

Network Skimming

```
import numpy as np
```

Let's build all graphs

```
project.network.build_graphs()
# We get warnings that several fields in the project are filled with ``NaN``s.
# This is true, but we won't use those fields.
```

We grab the graph for cars

```
graph = project.network.graphs["c"]

# we also see what graphs are available
project.network.graphs.keys()

# let's say we want to minimize the distance
graph.set_graph("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
graph.set_skimming(["distance"])

# But let's say we only want a skim matrix for nodes 28-40, and 49-60 (inclusive),
# these happen to be a selection of western centroids.
graph.prepare_graph(np.array(list(range(28, 41)) + list(range(49, 91))))
```

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.paths import NetworkSkimming

# skm = NetworkSkimming(graph)
# skm.execute()
```

The result is an AequilibraEMatrix object

```
skims = skm.results.skims

# Which we can manipulate directly from its temp file, if we wish
skims.matrices[: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

```
skm.save_to_project("base_skims")
```

We can also retrieve this skim record to write something to its description

```
matrices = project.matrices
mat_record = matrices.get_record("base_skims")
mat_record.description = "minimized distance while also skimming distance for just a␣
↪few nodes"
mat_record.save()
```

```
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 modeling 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, add them to a traffic assignment object, set the traffic assignment parameters, and run the assignment.

7.1.1 `TrafficClass`

The `TrafficClass` object holds all the information pertaining to a specific traffic class to be assigned. There are three pieces of information that are required in the instantiation of this class:

- **name**: name of the class. It has to be unique among all classes used in a multi-class traffic assignment
- **graph**: it is the `Graph` object corresponding to that particular traffic class/mode
- **matrix**: it is the AequilibraE matrix with the demand for that traffic class, which can have an arbitrary number of user-classes setup as different layers (cores) of the matrix object.

```
>>> from aequilibrae.paths import TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# We get the graphs for cars and trucks
>>> graph_car = project.network.graphs['c']
>>> graph_truck = project.network.graphs['T']

# And also get the matrices for cars and trucks
>>> matrix_car = project.matrices.get_matrix("demand_mc")
>>> matrix_car.computational_view("car")
```

(continues on next page)

(continued from previous page)

```
>>> matrix_truck = project.matrices.get_matrix("demand_mc")
>>> matrix_truck.computational_view("trucks")

# We create the Traffic Classes
>>> tc_car = TrafficClass("car", graph_car, matrix_car)
>>> tc_truck = TrafficClass("truck", graph_truck, matrix_truck)
```

It is also possible to modify the default values for the following parameters of a traffic class by using a method call:

- **Passenger-car equivalent** (PCE) is the standard way of modeling multi-class traffic assignment equilibrium in a consistent manner (see[3] for the technical detail), and its value is set to 1.0 by default.

```
>>> tc_truck.set_pce(2.5)
```

- **Fixed costs**: in case there are fixed costs associated with the traversal of links in the network, the user can provide the name of the field in the graph that contains that network.

```
>>> tc_truck.set_fixed_cost("distance")
```

- **Value-of-Time** (VoT) is the mechanism to bring time and monetary costs into a consistent basis within a generalized cost function. In the event that fixed cost is measured in the same unit as free-flow travel time, then *vot* must be set to 1.0.

```
>>> tc_truck.set_vot(0.35)
```

7.1.2 TrafficAssignment

```
>>> from aequilibrae.paths import TrafficAssignment

>>> assig = TrafficAssignment()
```

AequilibraE's traffic assignment is organized within an object with the same name which contains a series of member variables that should be populated by the user, providing thus a complete specification of the assignment procedure.

- **classes**: list of completely specified traffic classes

```
# You can add one or more traffic classes to the assignment instance
>>> assig.add_class(tc_truck)

>>> assig.set_classes([tc_car, tc_truck])
```

- **vdf**: the volume-delay function (VDF) to be used, being one of `BPR`, `BPR2`, `CONICAL`, `INRETS`, or `AKCELIK`

```
>>> assig.set_vdf('BPR')
```

- **vdf_parameters**: the parameters to be used in the volume-delay function, other than volume, capacity and free-flow time. VDF parameters must be consistent across all graphs.

  Because AequilibraE supports different parameters for each link, its implementation is the most general possible while still preserving the desired properties for multi-class assignment, but the user needs to provide individual values for each link *OR* a single value for the entire network.

[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

Setting the VDF parameters should be done *AFTER* setting the VDF function of choice and adding traffic classes to the assignment, or it will *fail*.

```
# The VDF parameters can be either an existing field in the graph, passed as a␣
↪parameter:
>>> assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# Or as a global value:
>>> assig.set_vdf_parameters({"alpha": 0.15, "beta": 4})
```

- **time_field**: the field of the graph that corresponds to free-flow travel time. The procedure will collect this information from the graph associated with the first traffic class provided, but will check if all graphs have the same information on free-flow travel time

```
>>> assig.set_time_field("free_flow_time")
```

- **capacity_field**: the field of the graph that corresponds to the link capacity. The procedure will collect this information from the graph associated with the first traffic class provided, but will check if all graphs have the same information on capacity

```
>>> assig.set_capacity_field("capacity")
```

- **algorithm**: the assignment algorithm to be used, being one of `all-or-nothing`, `bfw`, `cfw`, `fw`, `franke-wolfe`, or `msa`.

```
>>> assig.set_algorithm("bfw")
```

Skimming while assigning

AequilibraE allows for skimming to be performed during assignnment, and maintains both the skimming of the final iteration, as well as the blended skim for all iterations.

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.

```
>>> assig.set_skimming_fields(["distance"])
>>> assig.execute()
>>> assig.save_skims("one_matrix_name")
```

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 for the assignment or any individual traffic class graphs**.

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.

```
>>> assig.execute()
>>> skims = assig.skim_congested(skim_fields=["distance"], return_matrices=True)
>>> assig.save_skims("another_matrix_name")
```

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.

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 modeling 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
- Behavioral 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**: Signalized 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:

```
>>> from aequilibrae.transit import Transit

# Times are specified in seconds from midnight
>>> transit = Transit(project)
>>> preload = transit.build_pt_preload(start=8*3600, end=10*3600)

# Add the preload to the assignment
>>> assig.add_preload(preload, 'PT_vehicles')
```

7.1.4 Executing an Assignment

Finally, run traffic assignment!

```
>>> assig.execute()

>>> 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 modeling traditions.

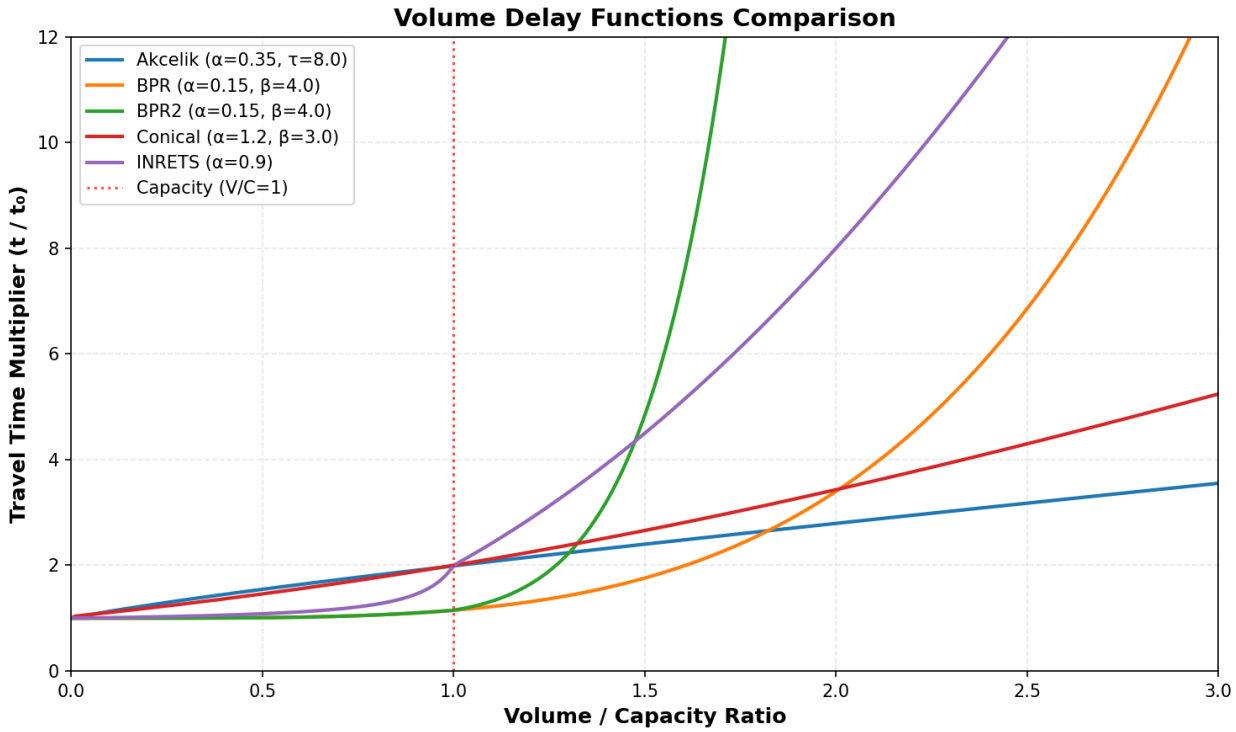
7.2.1 Available VDF Functions

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 behavior of all available VDF functions with their example values:

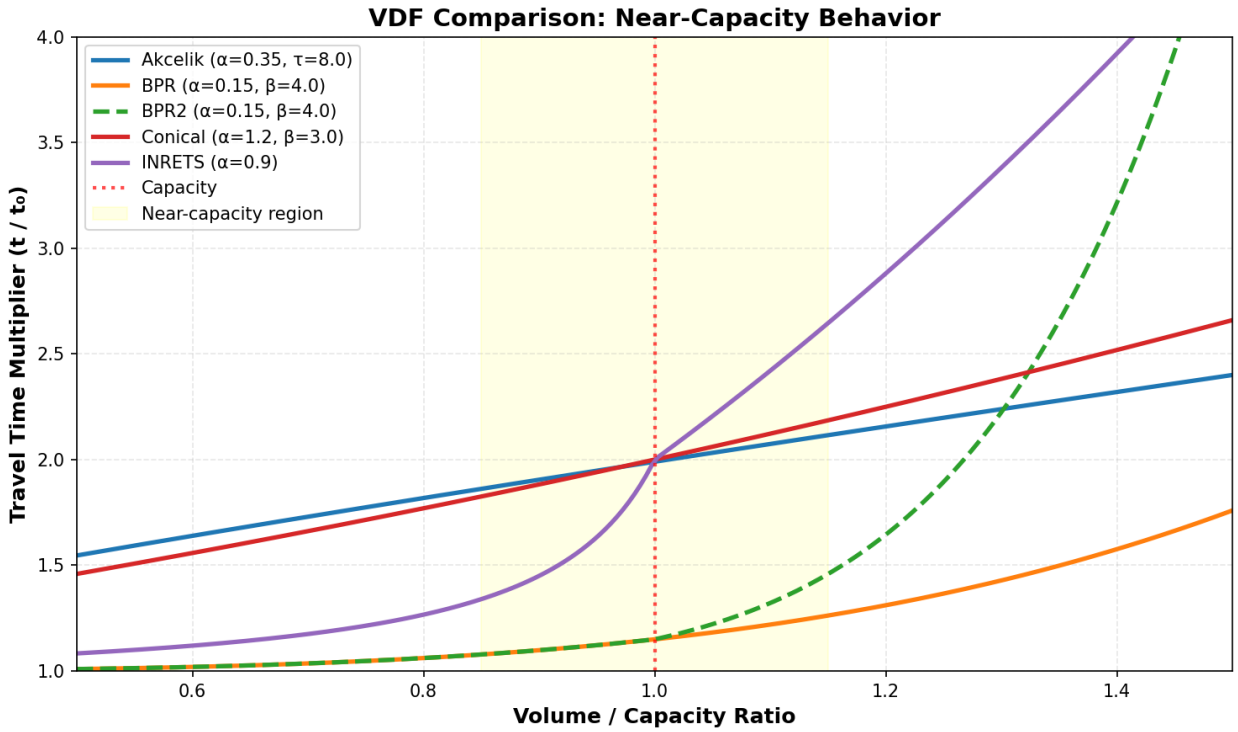


Key observations:

- All functions show increasing travel times as the Volume/Capacity (V/C) ratio increases
- Functions differ significantly in their behavior near and above capacity (V/C = 1)
- BPR and Conical are smooth throughout the range
- BPR2 and INRETS have distinct behavioral changes at capacity
- Akcelik shows moderate growth rates suitable for signalized intersections

Near-Capacity Behavior

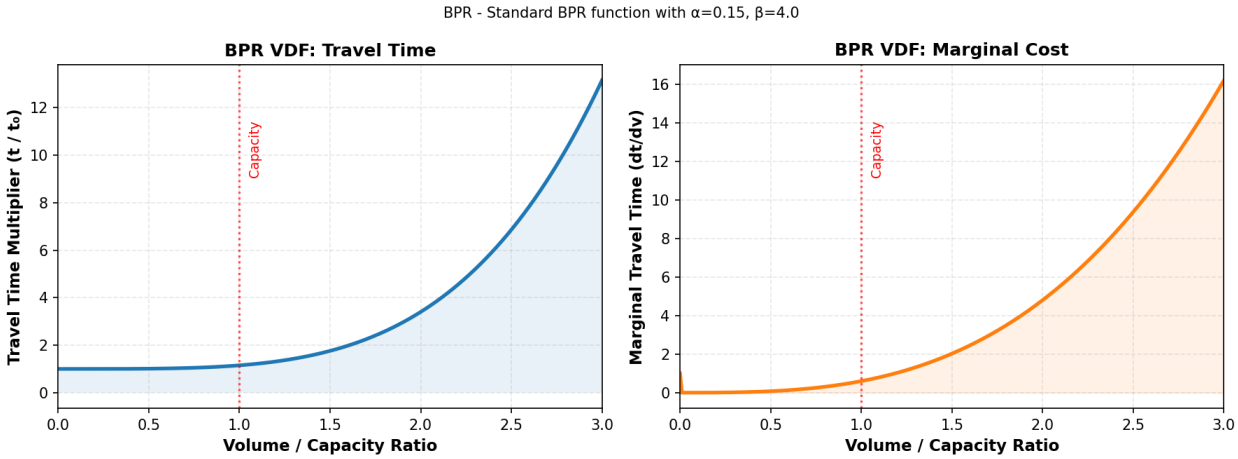
The behavior of VDFs near capacity is particularly important for congested urban networks:



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

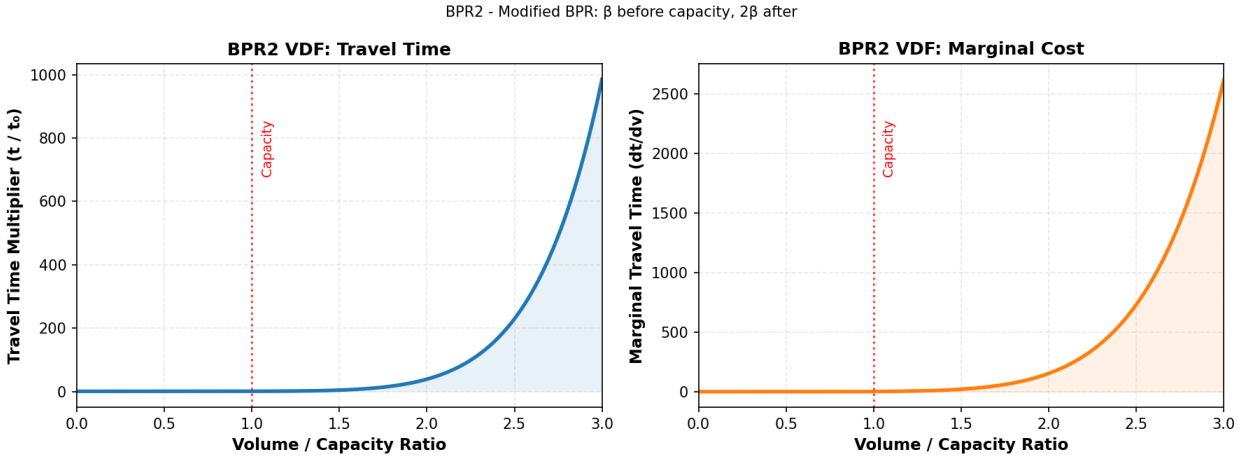
When to Use:

- Highway and freeway networks
- Long-distance travel modeling
- 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 behavior below capacity
- Non-differentiable at V/C = 1 (but continuous)
- More aggressive congestion penalty than standard BPR

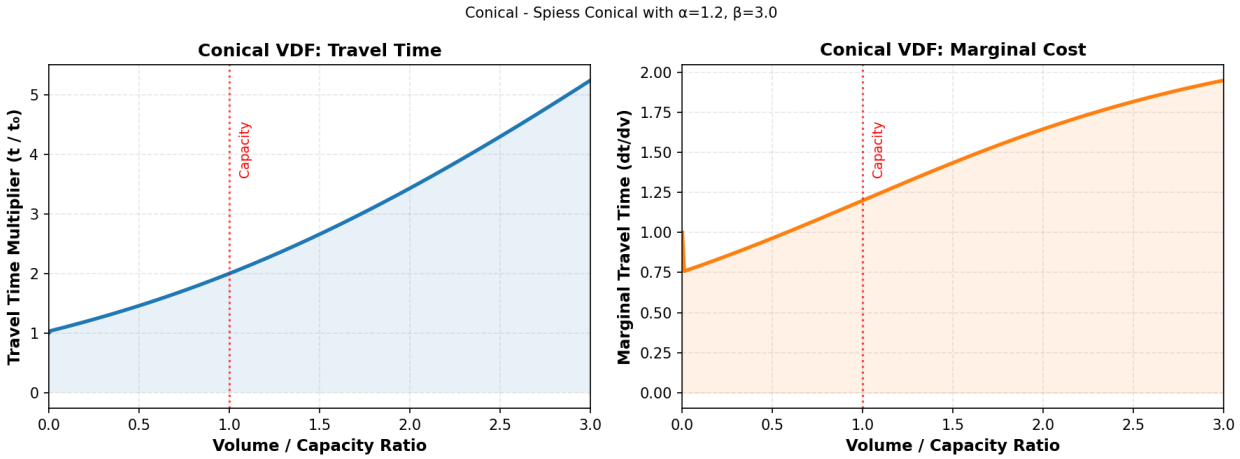
When to Use:

- 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 behavior

Conical (Spiess)



Mathematical Formula:

$$t = t_0 \left(2 + \sqrt{\alpha^2 \left(1 - \frac{v}{c}\right)^2 + \beta^2} - \alpha \left(1 - \frac{v}{c}\right) - \beta\right)$$

Standard Parameters:

- $\alpha = 0.15$
- $\beta = 4.0$

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 behavior as V/C approaches infinity
- Guaranteed positive marginal costs
- Strong theoretical foundation
- Good convergence properties in equilibrium algorithms

When to Use:

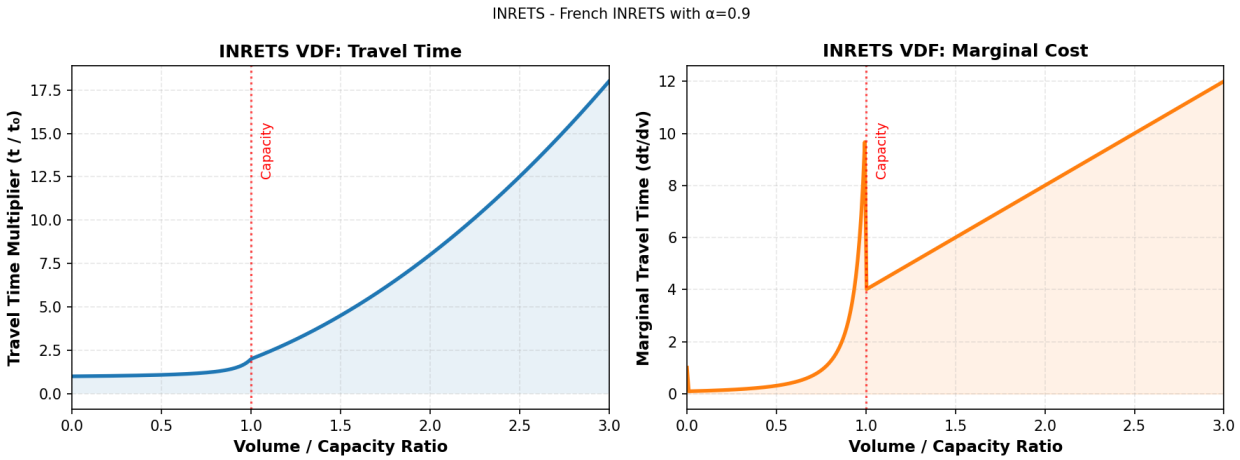
- When theoretical rigor is important
- Transit assignment (where it was originally designed)
- Networks requiring guaranteed convergence
- Academic studies and benchmarking

- 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 behavior
- The α parameter must be less than or equal to 1.0
- Hyperbolic behavior before capacity
- Quadratic behavior after capacity
- Designed for urban arterial roads

- Reflects observed traffic patterns in French cities

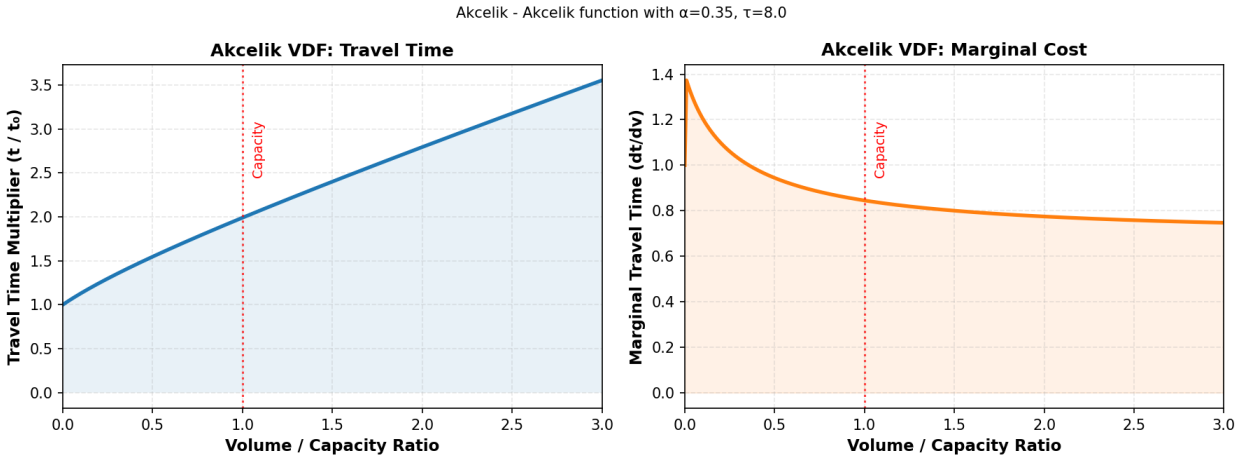
When to Use:

- Urban arterial networks
- European-style road networks
- When modeling contexts similar to French urban environments
- Networks with well-defined capacity constraints
- Calibrated models for specific urban areas

Limitations:

- Restricted parameter range ($\alpha <= 1.0$)
- Non-differentiable at V/C = 1
- Less widely used outside of Europe
- May require local calibration
- Dramatic change in behavior 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 modeling contexts (where it's more common)

Limitations:

- Less intuitive than BPR for general highway modeling
- 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:

- Can use similar values to BPR as starting points
- $\alpha = 0.15$ and $\beta = 4.0$ provide comparable behavior to BPR
- Fine-tuning may require understanding of the specific mathematical properties

For INRETS:

- $\alpha = 1.0$ is standard
- Must satisfy $\alpha <= 1.0$
- 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.paths import TrafficAssignment

assig = TrafficAssignment()
assig.set_vdf('BPR')
assig.set_vdf_parameters({"alpha": 0.15, "beta": 4.0})
```

Link-specific parameters:

```
# Parameters can reference field names in your network
assig.set_vdf_parameters({"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 behavior 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 behavior, 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.paths import TrafficAssignment, TrafficClass, VDF

# Check available VDF functions
vdf = VDF()
print(vdf.functions_available())
# Output: ['bpr', 'bpr2', 'conical', 'inrets', 'akcelik']

# Setup traffic assignment with BPR
assig = TrafficAssignment()
assig.set_vdf('BPR')
assig.set_vdf_parameters({"alpha": 0.15, "beta": 4.0})
```

Using Different VDFs

```
# Highway network - standard BPR
assig.set_vdf('BPR')
assig.set_vdf_parameters({"alpha": 0.15, "beta": 4.0})

# Urban network with signals - Akcelik
assig.set_vdf('AKCELIK')
assig.set_vdf_parameters({"alpha": 0.25, "tau": 0.8, "legnth": "distance"})

# Network with strong capacity constraints - BPR2
assig.set_vdf('BPR2')
assig.set_vdf_parameters({"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
assig.set_vdf('BPR')
assig.set_vdf_parameters({"alpha": "alpha_field", "beta": "beta_field"})
```

7.2.8 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.

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
- Value-of-Time (VoT)

[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

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].

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.

[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

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, and although the literature on this is obvious, we took some time to realize that we should re-compute the travel costs only **AFTER** checking for convergence.
- 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 super-network consists in having all classes with the same links in their sub-graphs, but assigning *b_node* identical to *a_node* for all links whenever a link is not available for a certain user class.

This approach is slightly less efficient when we are computing shortest paths, but it gets eliminated when topologically compressing the network for centroid-to-centroid path computation and it is a LOT more efficient when we are aggregating flows.

The use of the AequilibraE project and its built-in methods to build graphs ensure that all graph will be built in a consistent manner and multi-class assignment is possible.

[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

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.

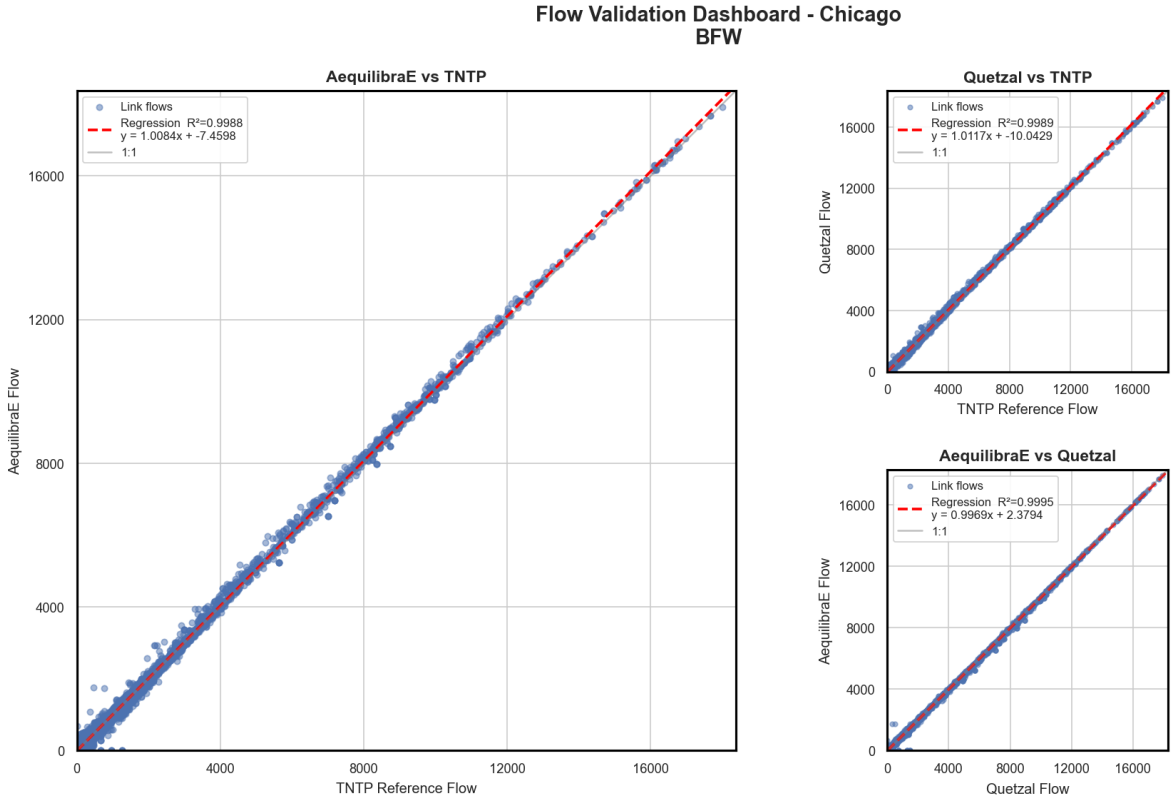
As shown below, the results produced by AequilibraE are within expected, although some differences have been found, particularly for Anaheim. We suspect that there are issues with the reference results and welcome further investigations.

Chicago

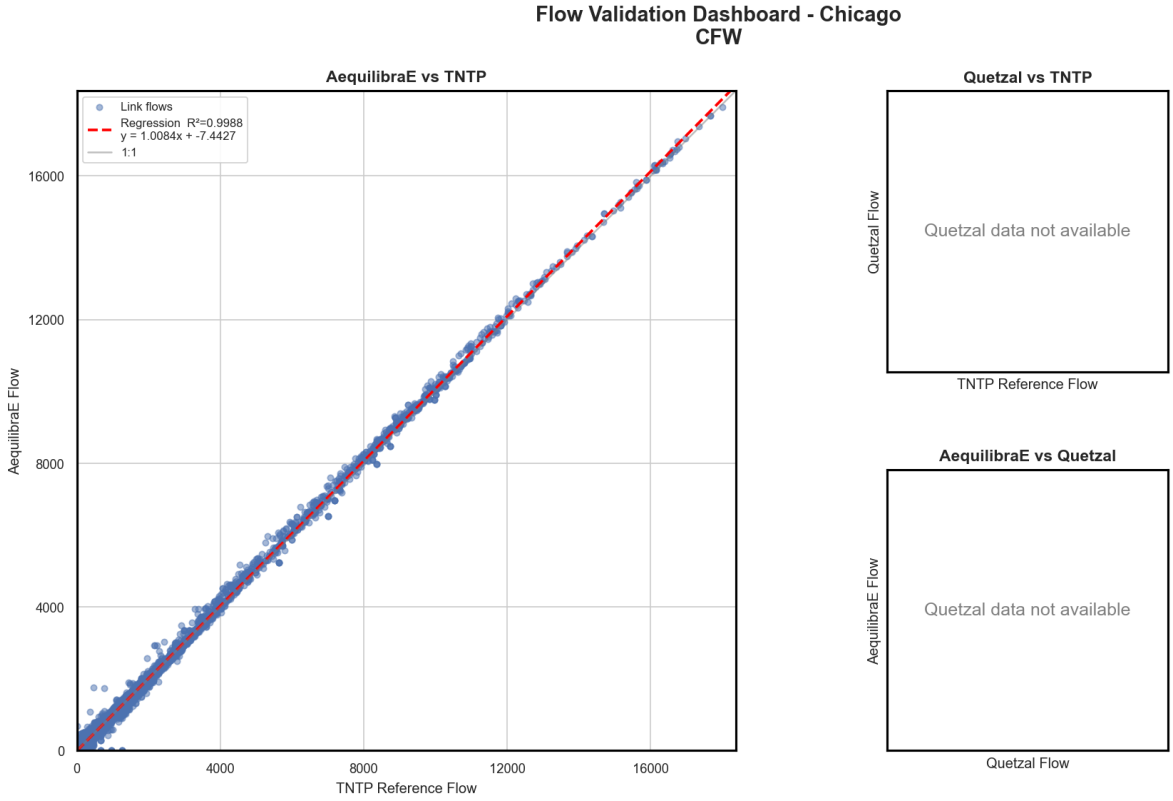
Network stats

- Links: 39,018
- Nodes: 12,982
- Zones: 1,790

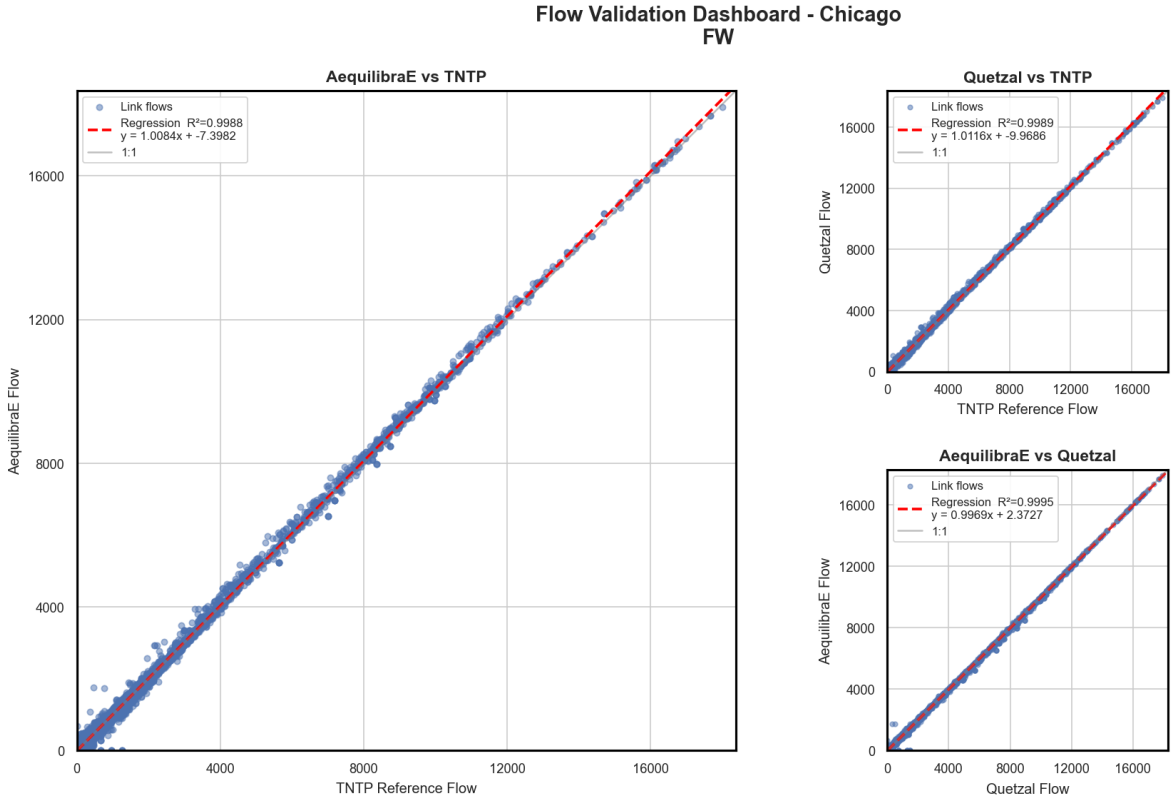
biconjugate Frank-Wolfe



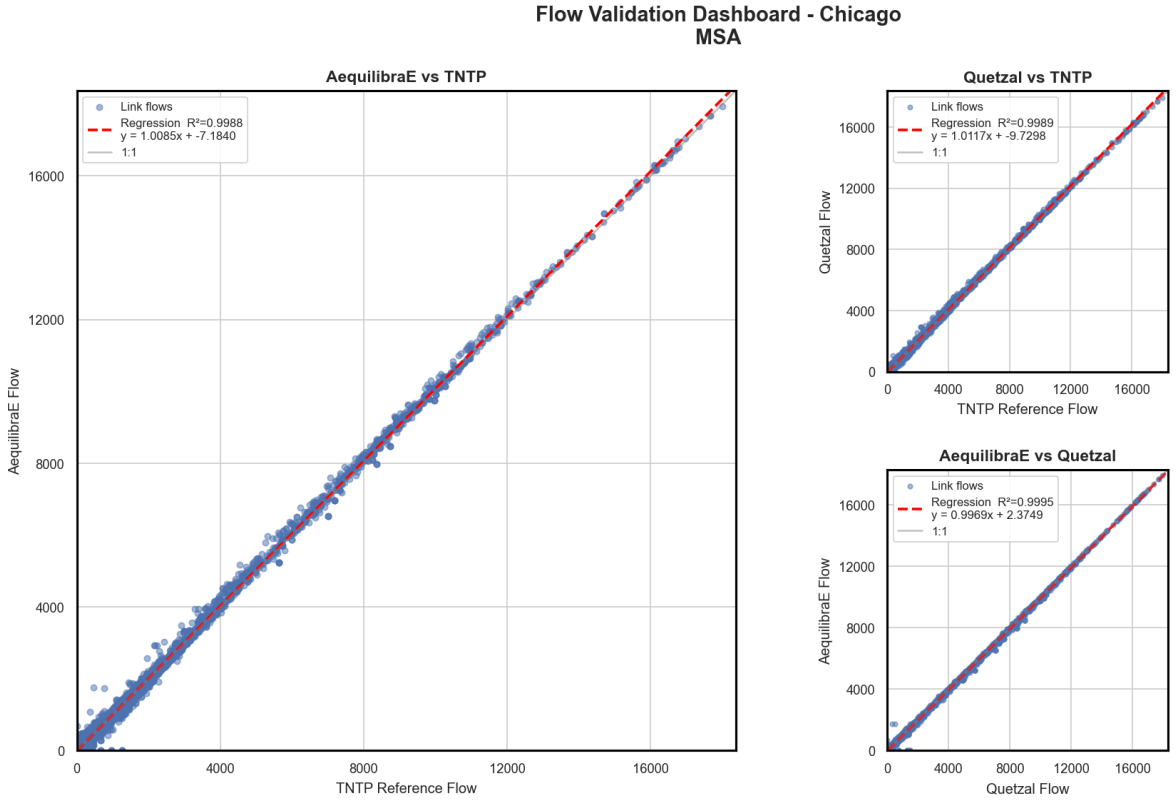
Conjugate Frank-Wolfe



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

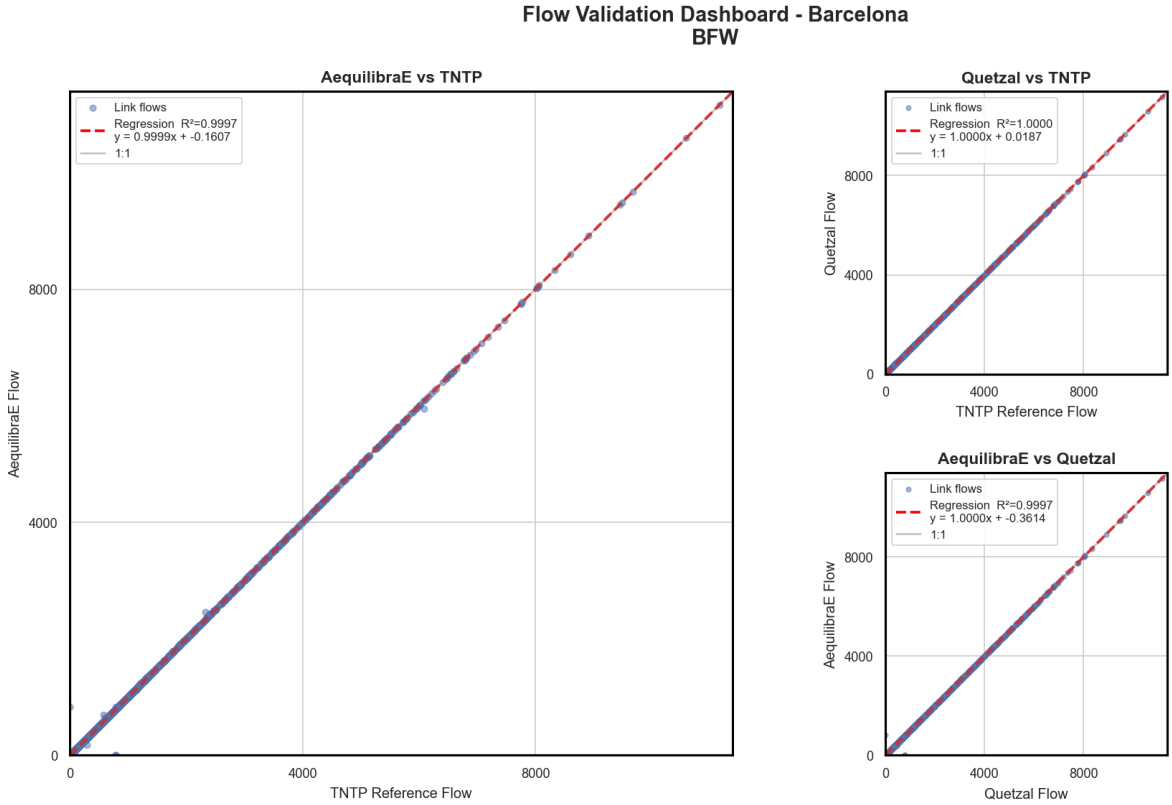


Barcelona

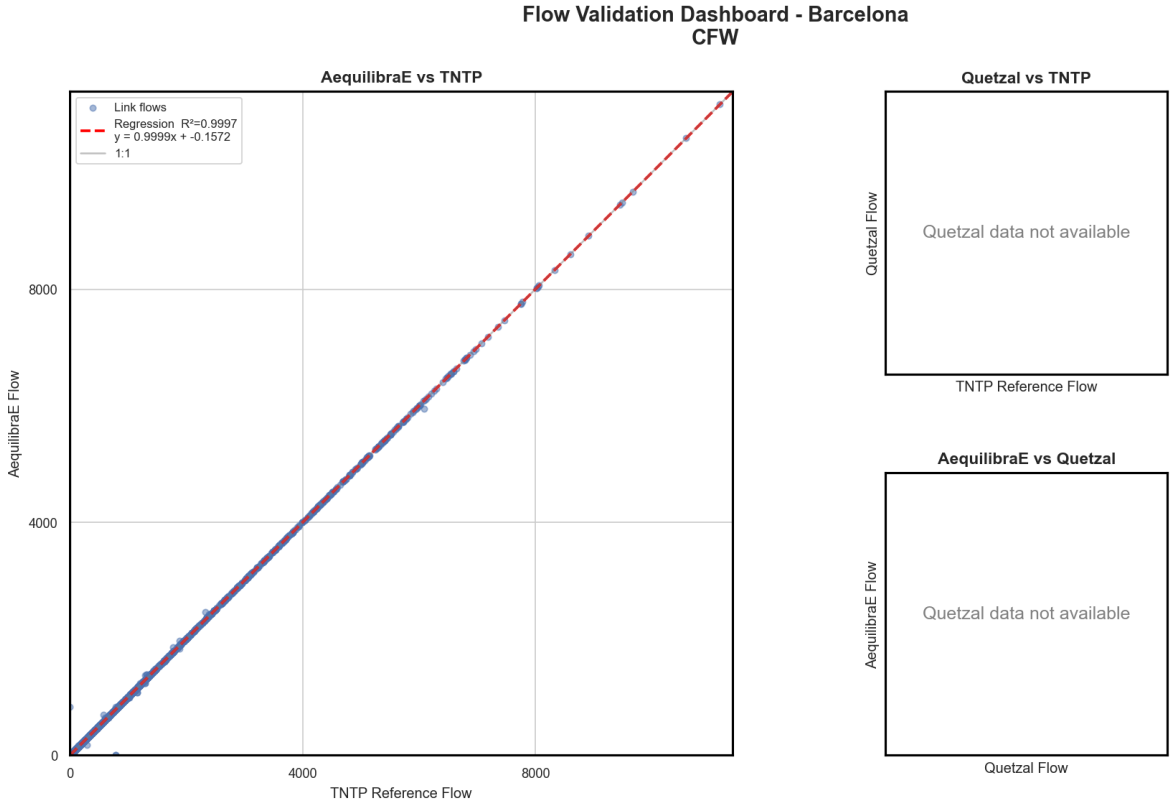
Network stats

- Links: 2,522
- Nodes: 1,020
- Zones: 110

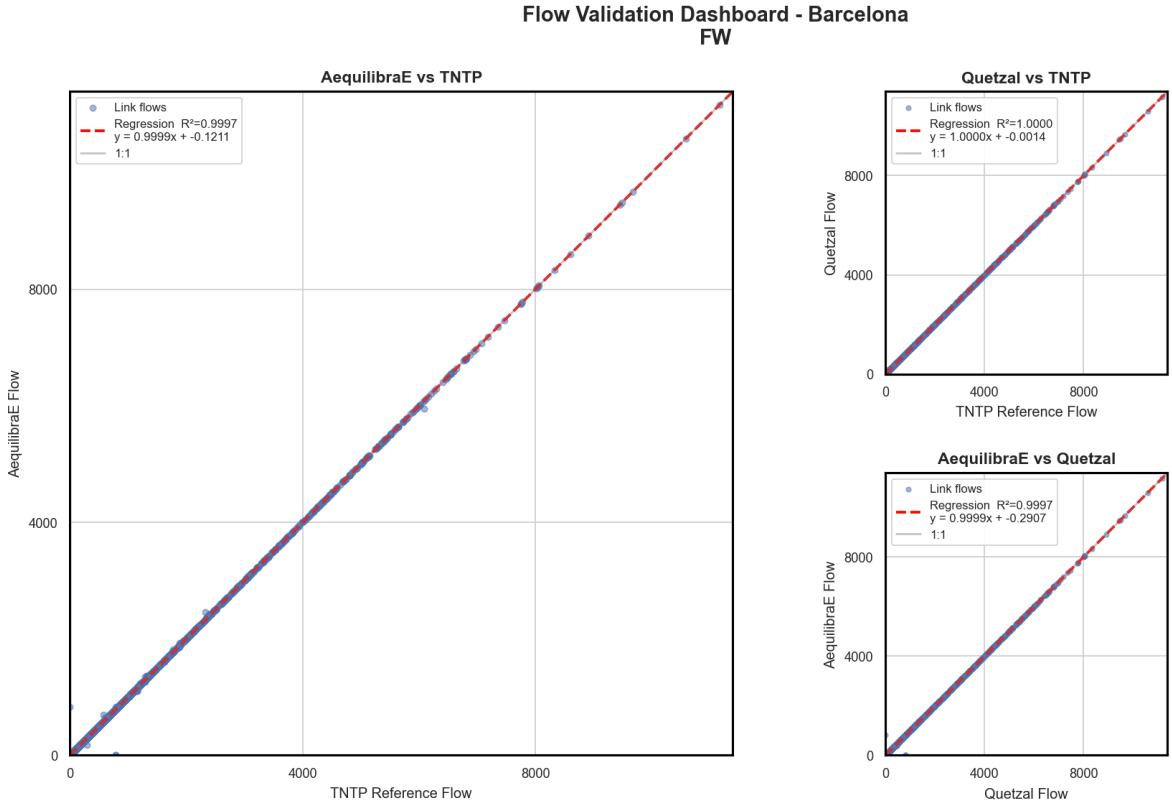
biconjugate Frank-Wolfe



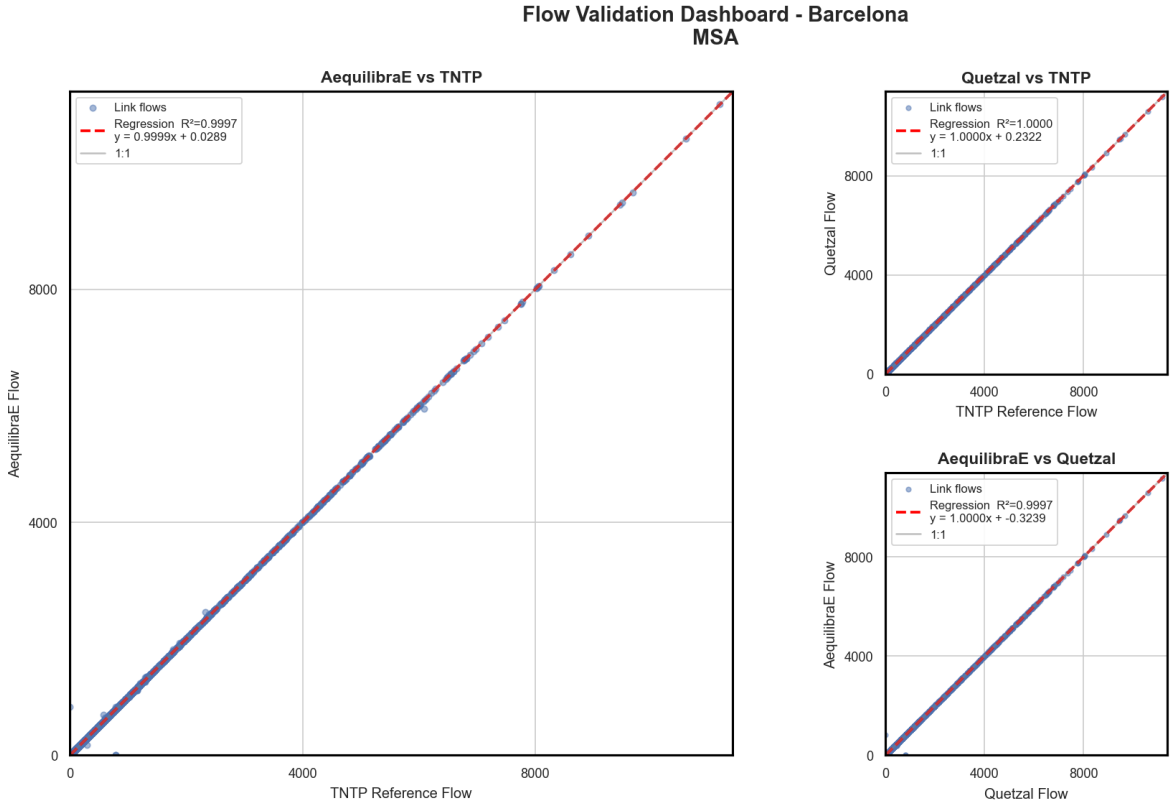
Conjugate Frank-Wolfe



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
Link flows
Regression R²=1.0000
y = 1.0000x + -0.0014
1:1
Quetzal Flow
TNTP Reference Flow
AequilibraE vs Quetzal
Link flows
Regression R²=0.9997
y = 0.9999x + -0.2907
1:1
AequilibraE Flow
Quetzal Flow

MSA

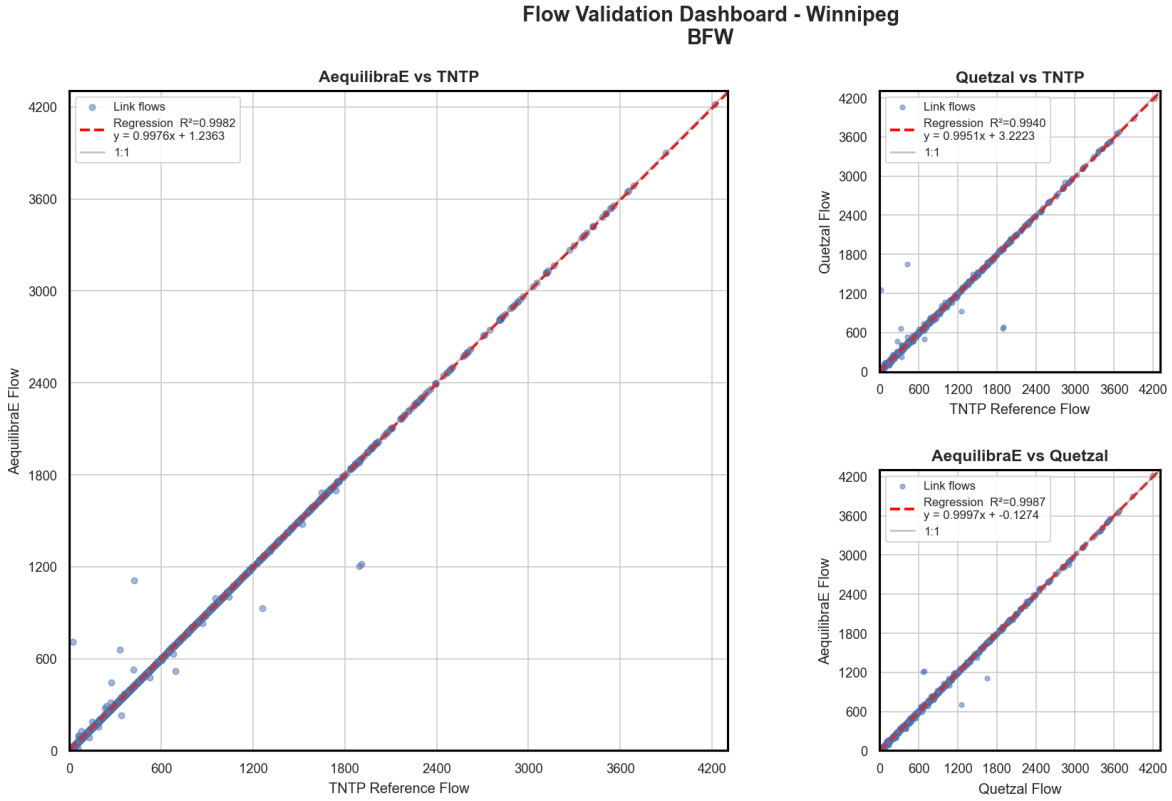


Winnipeg

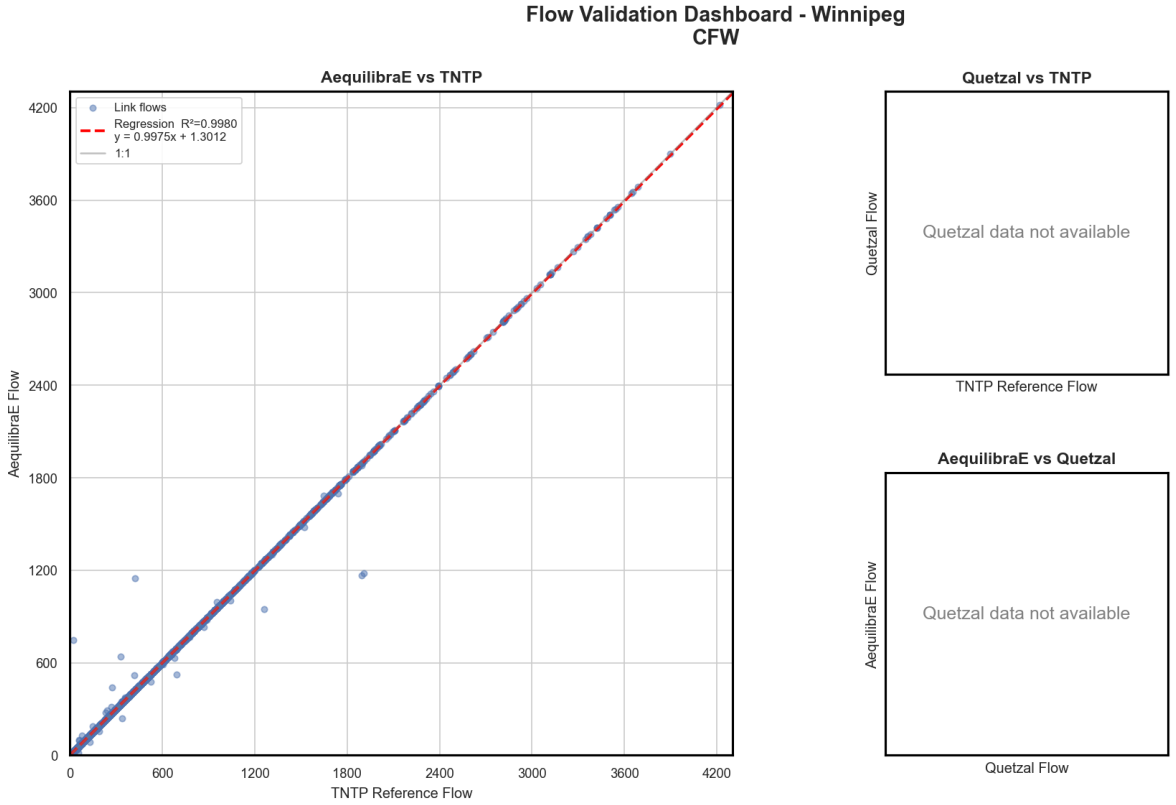
Network stats

- Links: 914
- Nodes: 416
- Zones: 38

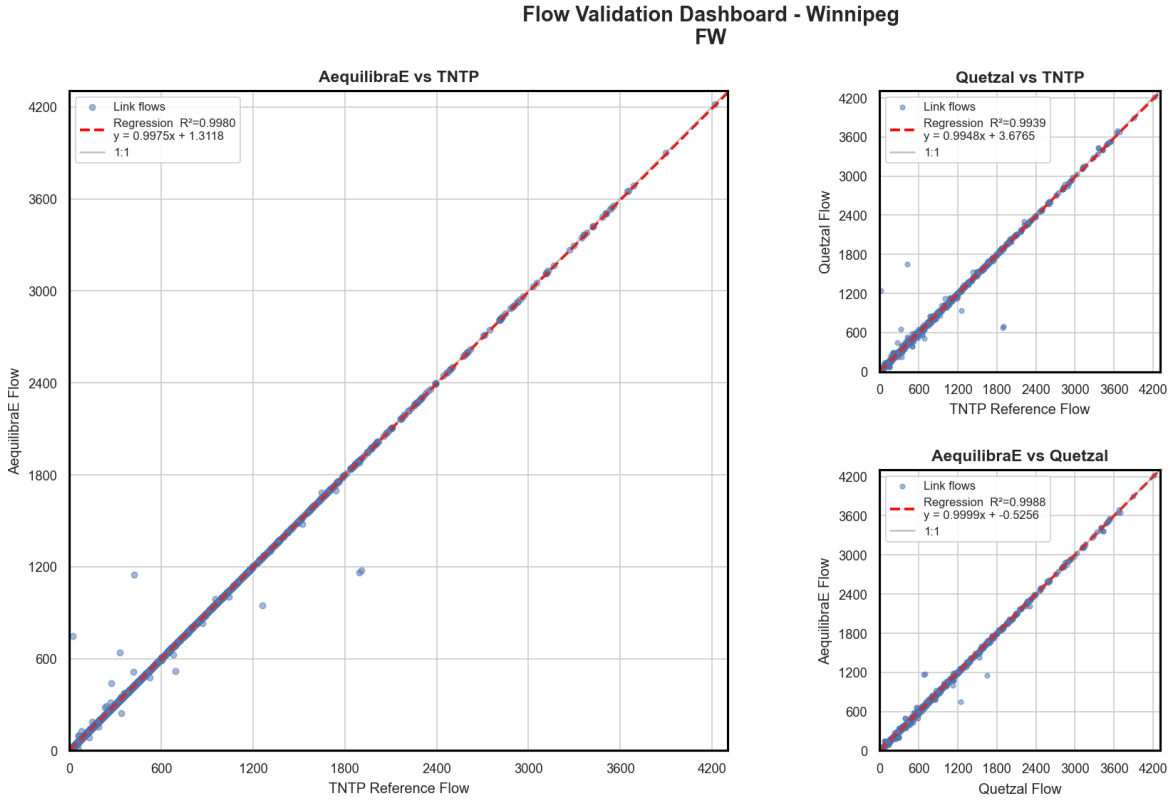
biconjugate Frank-Wolfe



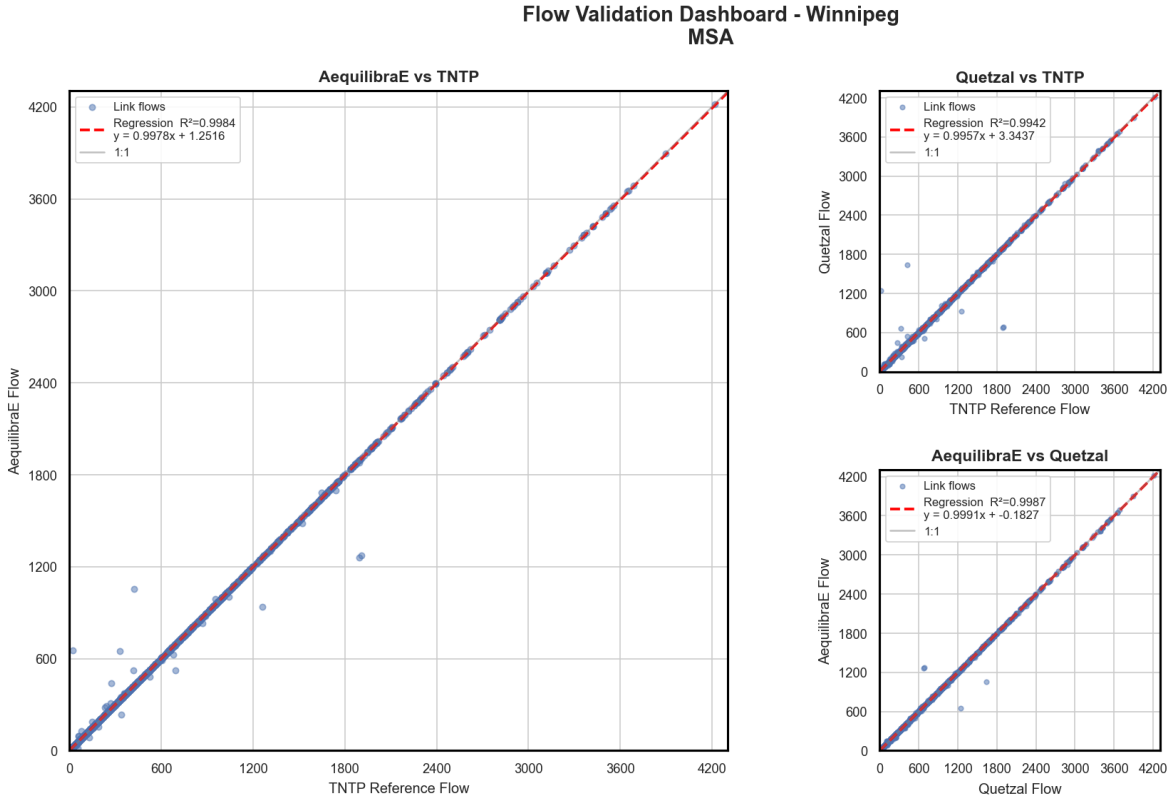
Conjugate Frank-Wolfe



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
Regression R²=0.9939
y = 0.9948x + 3.6765
Quetzal Flow
AequilibraE vs Quetzal
Regression R²=0.9988
y = 0.9999x + -0.5256

MSA

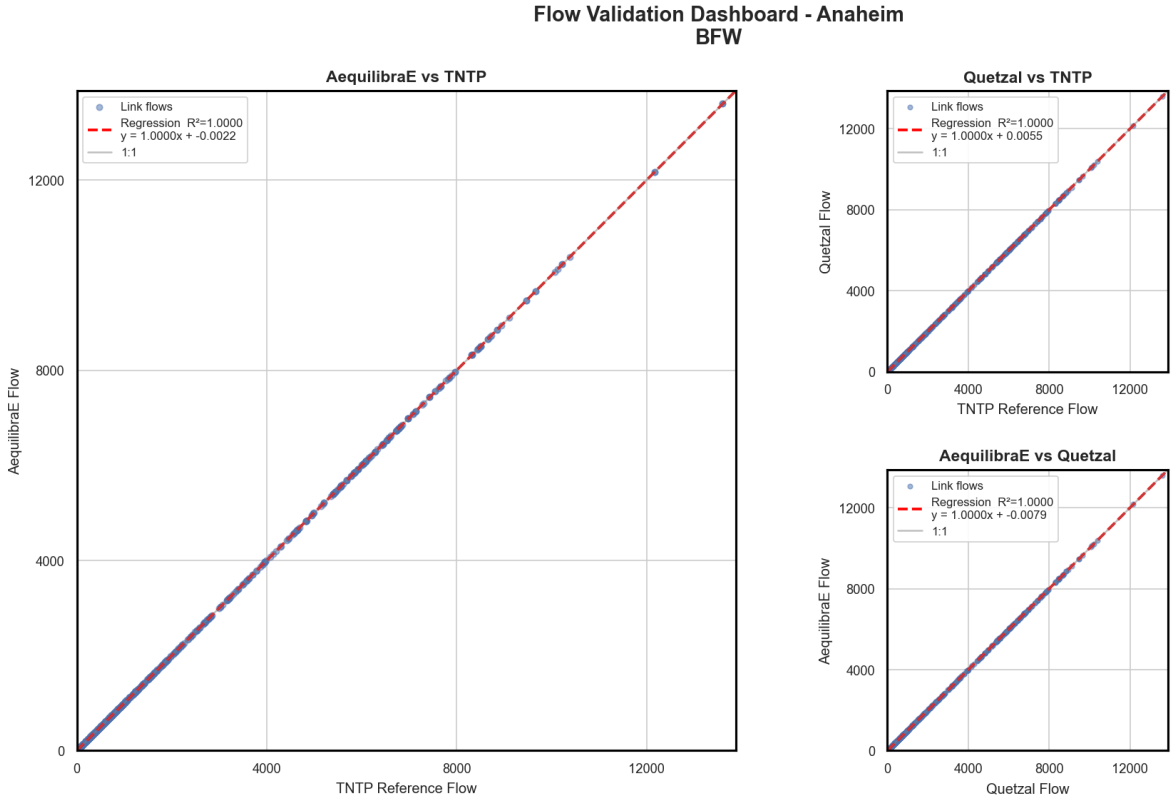


Anaheim

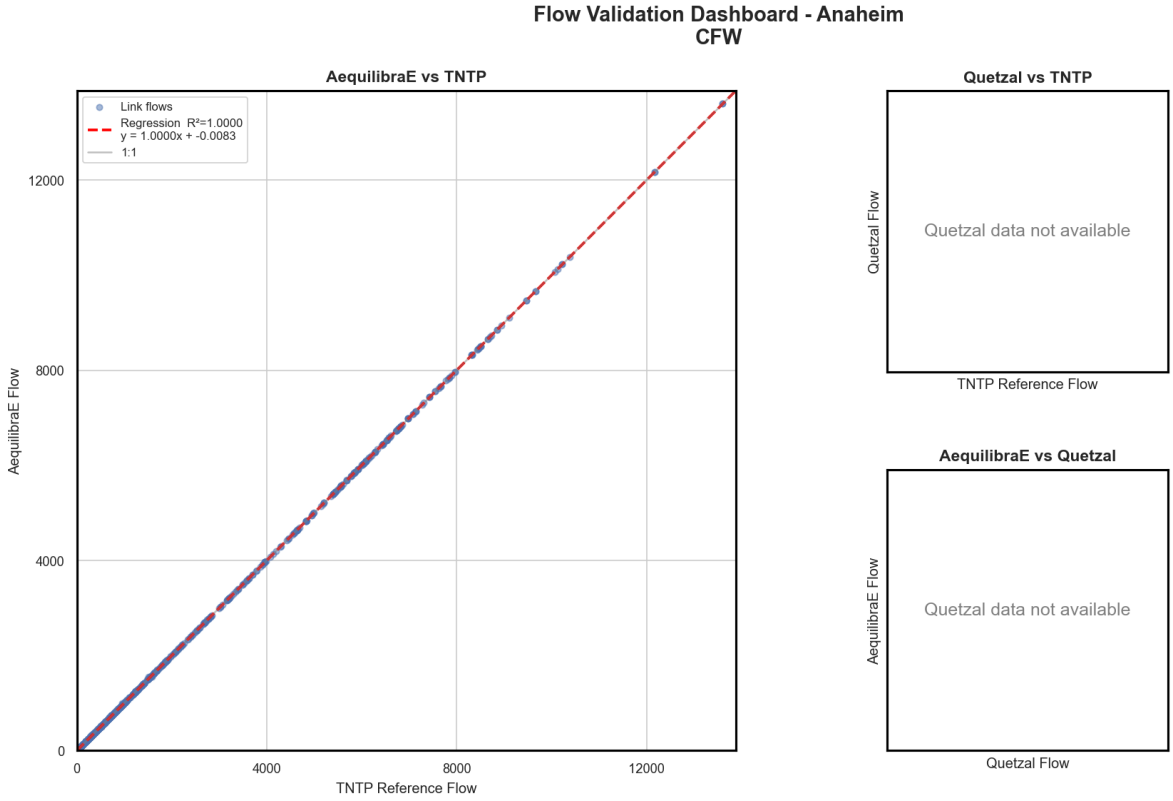
Network stats

- Links: 914
- Nodes: 416
- Zones: 38

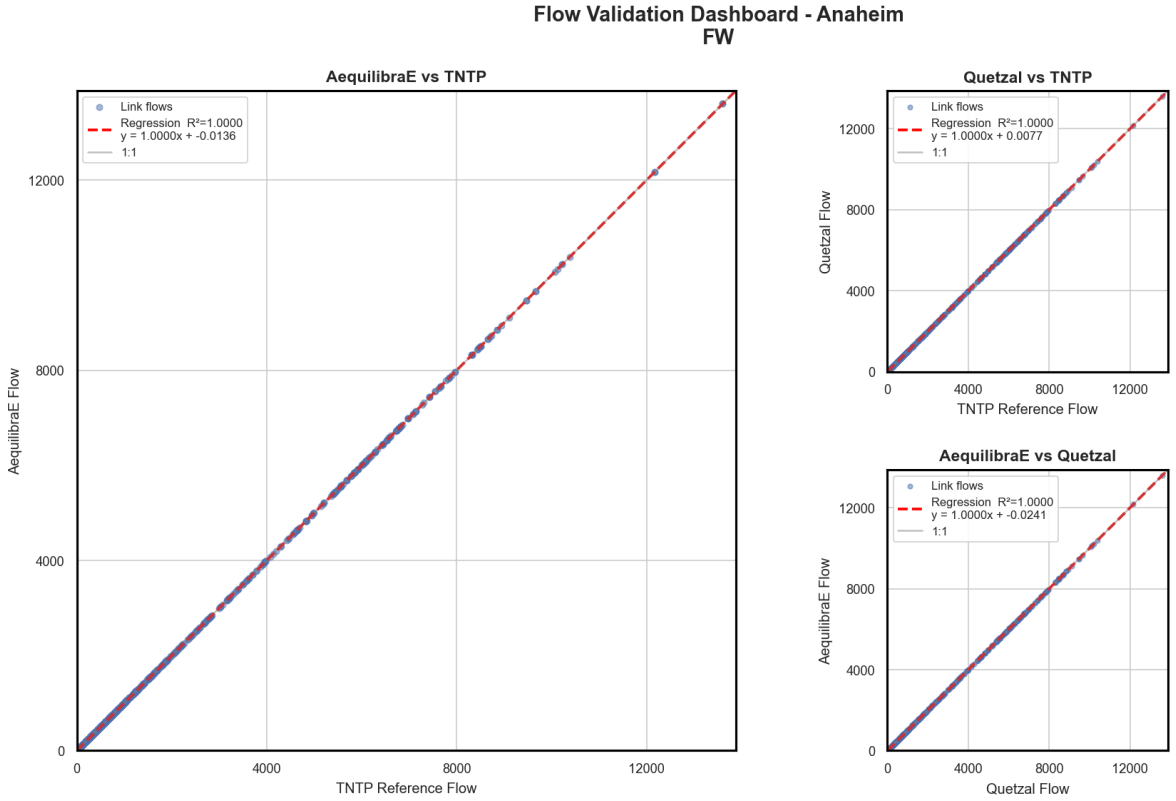
biconjugate Frank-Wolfe



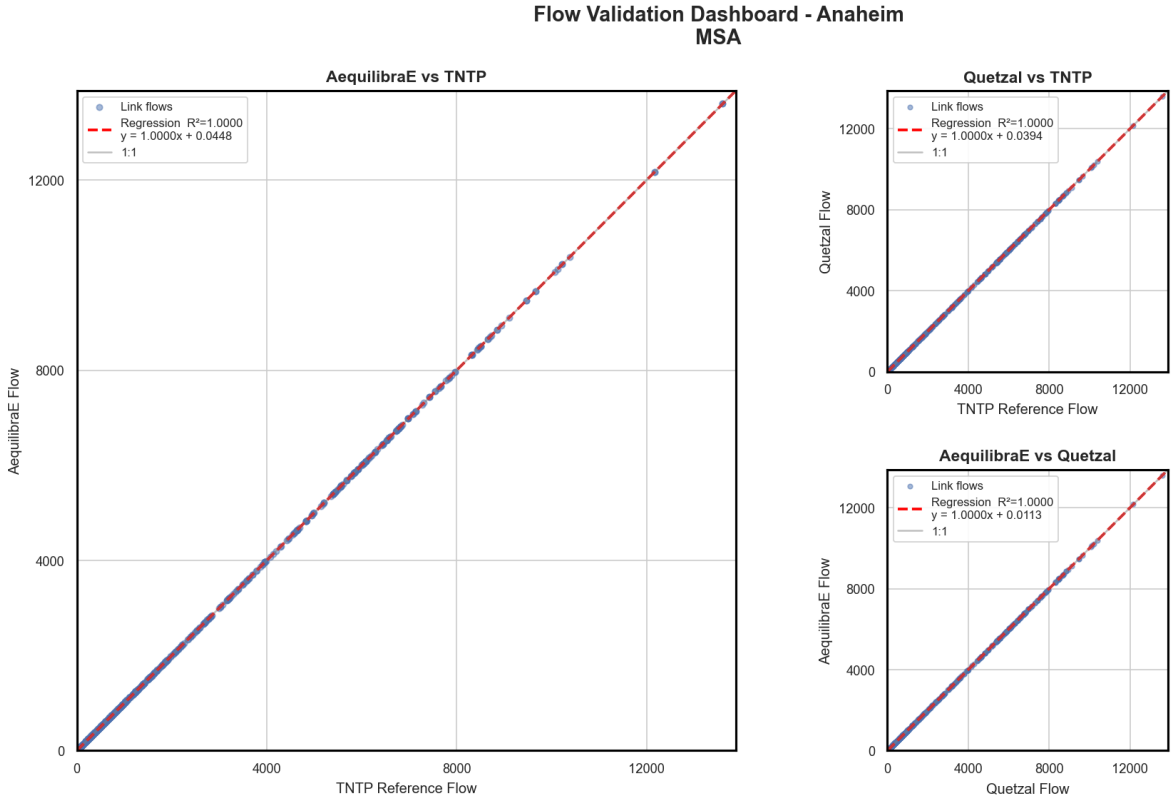
Conjugate Frank-Wolfe



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

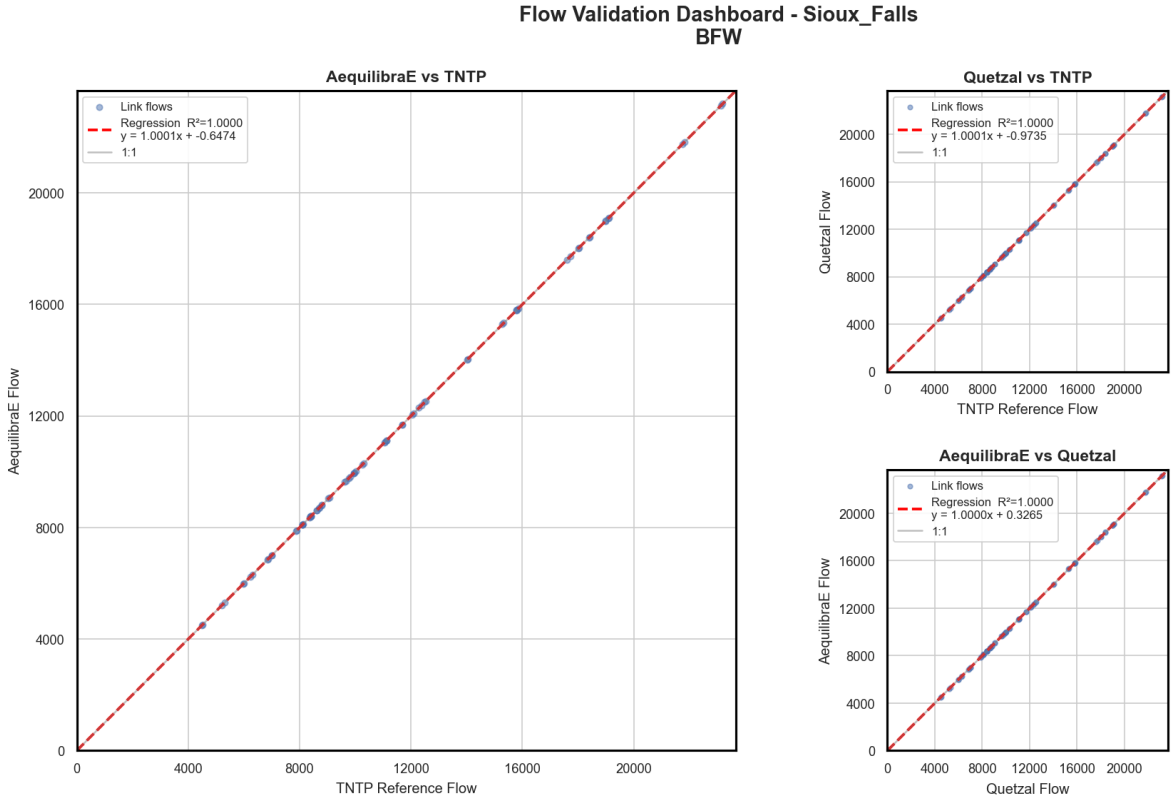


Sioux Falls

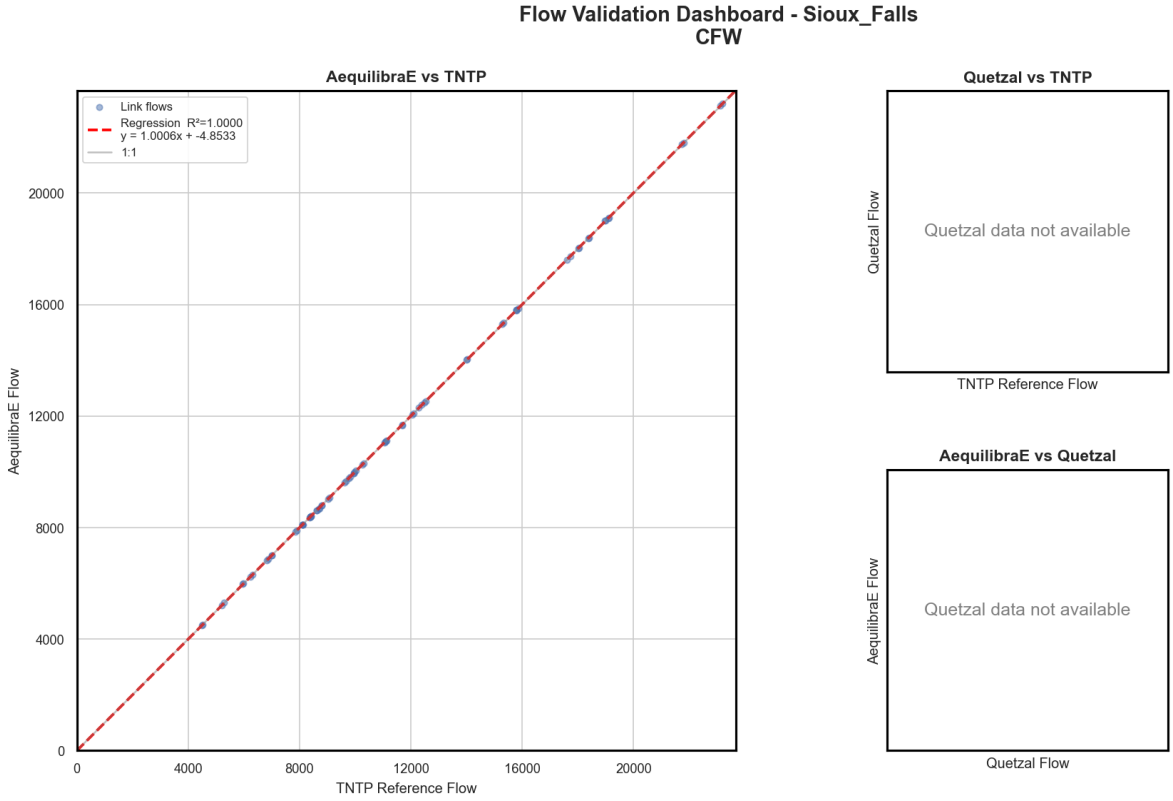
Network stats

- Links: 76
- Nodes: 24
- Zones: 24

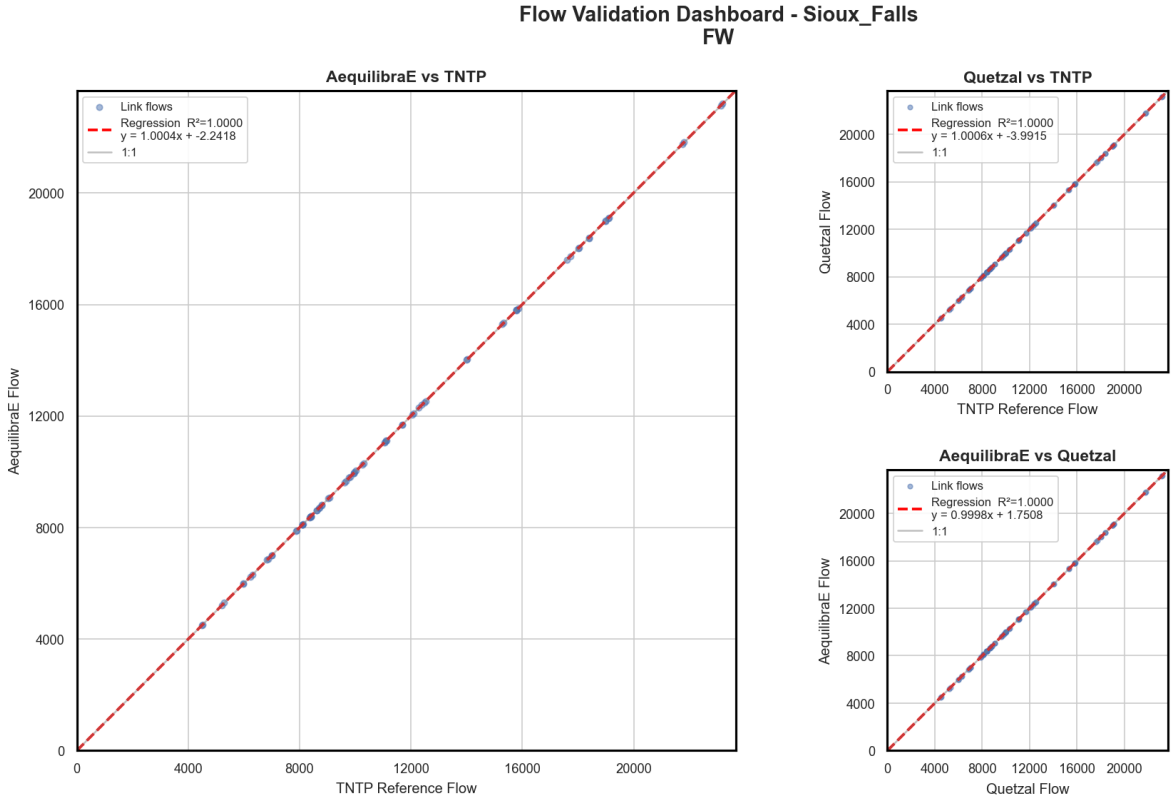
biconjugate Frank-Wolfe



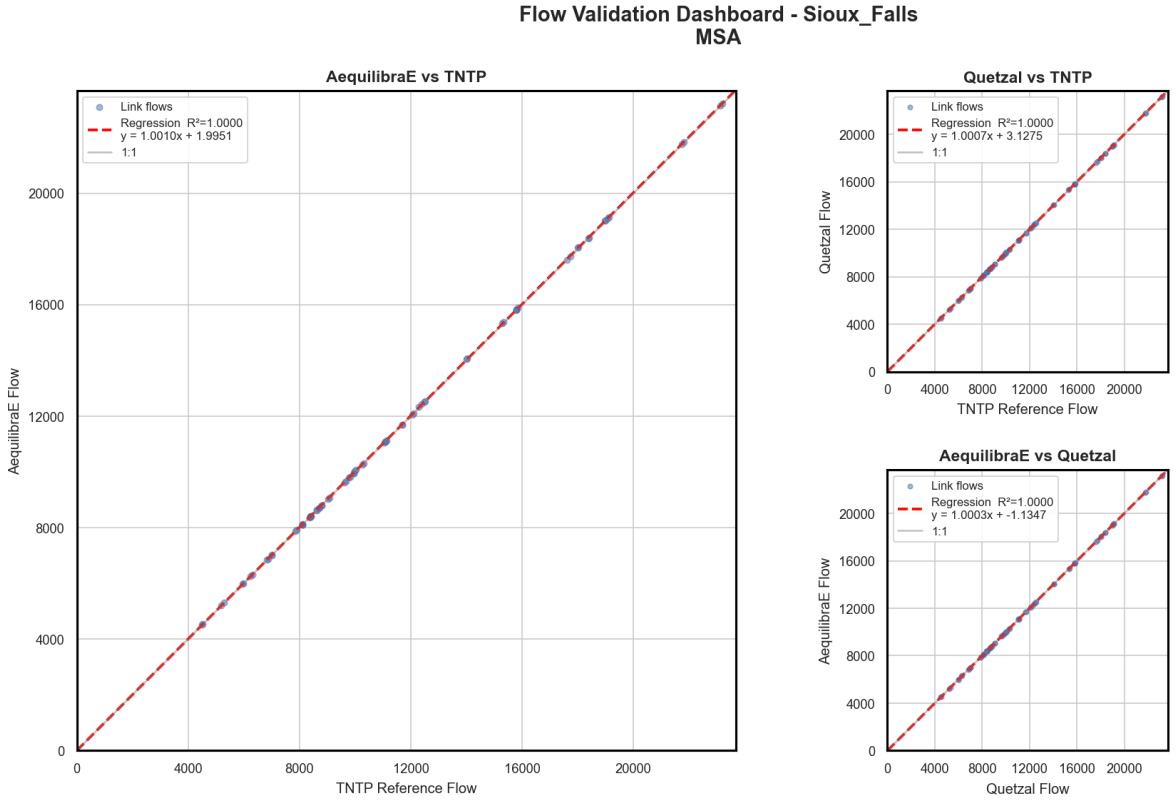
Conjugate Frank-Wolfe



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

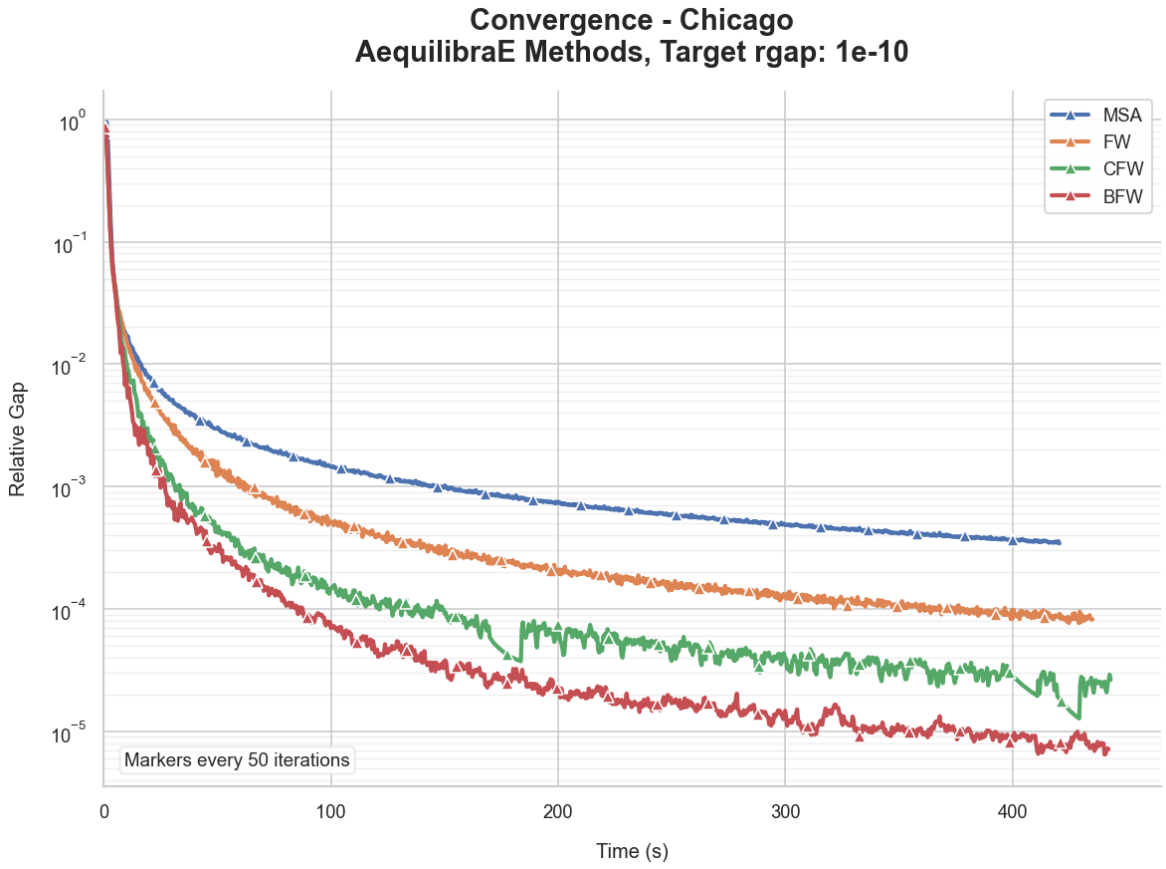
MSA



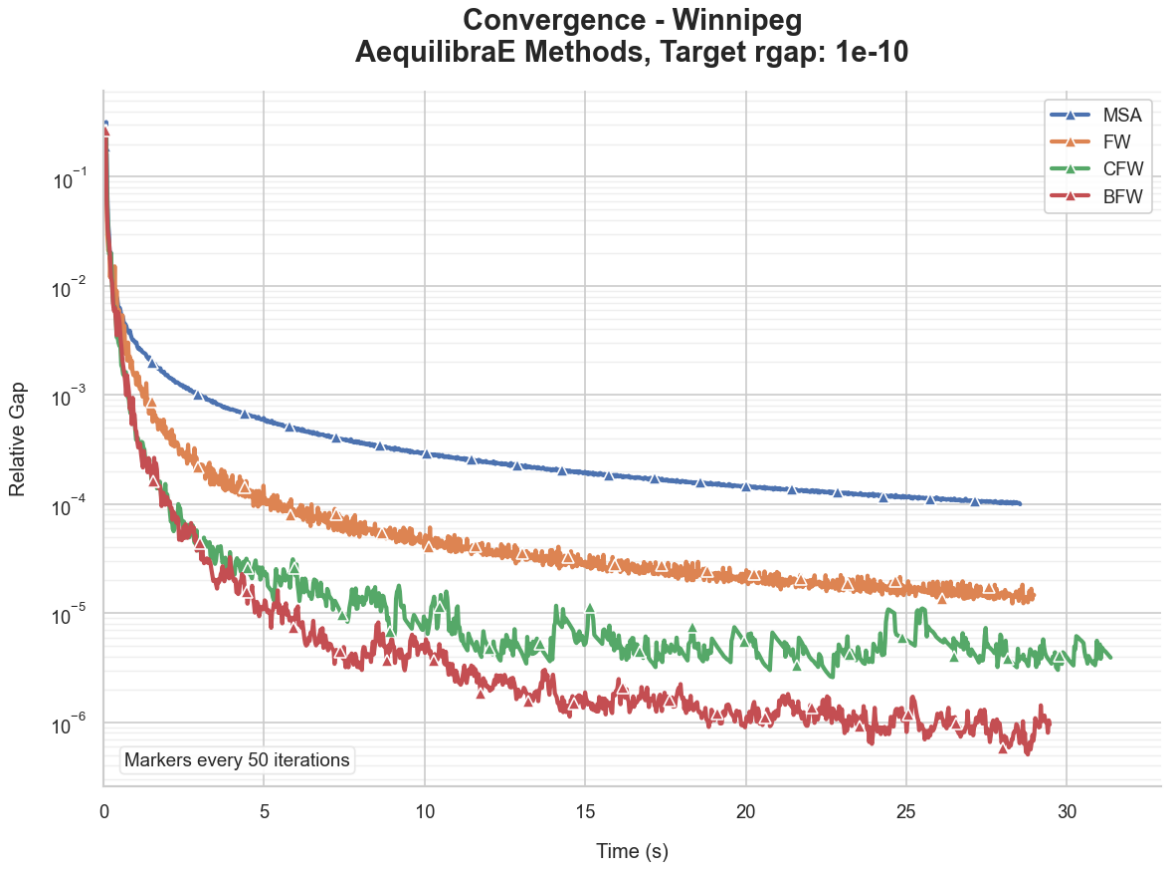
7.4.1 Convergence Study

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.

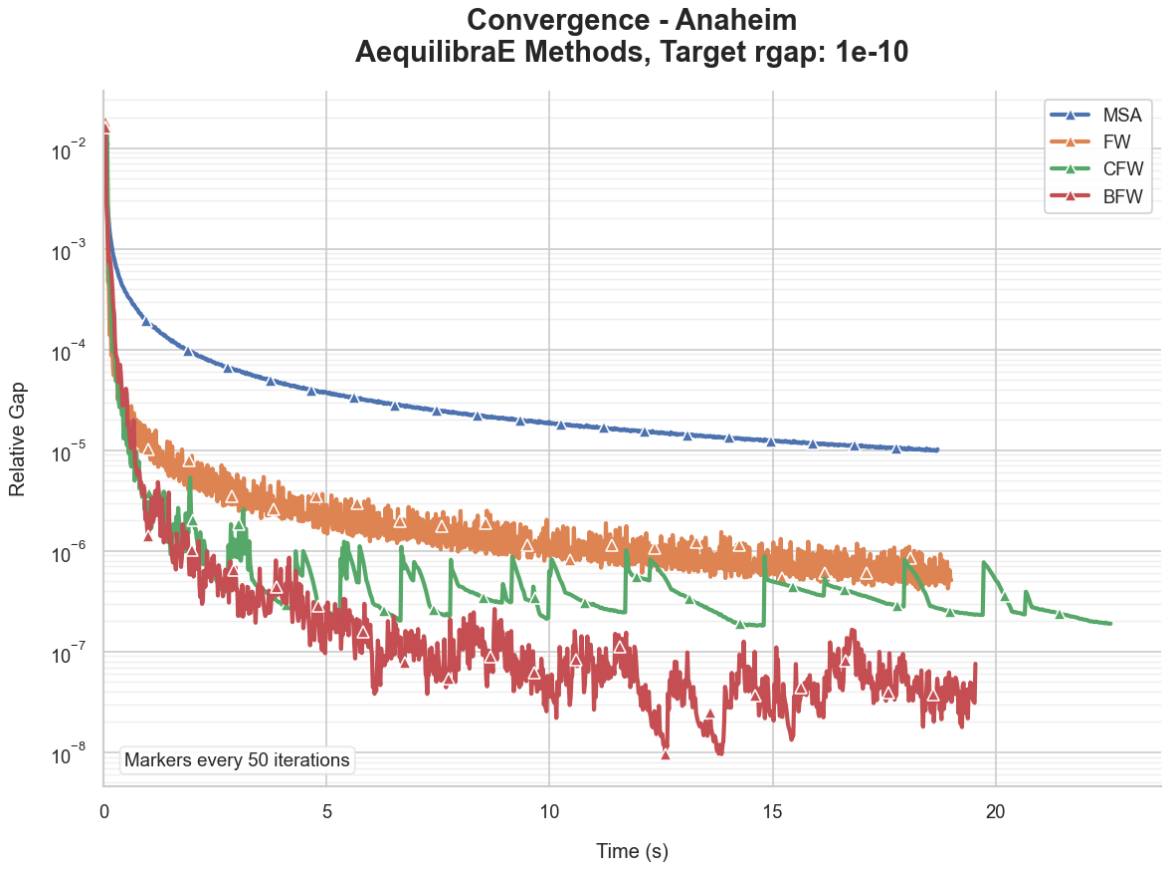
Chicago


Convergence - Chicago
AequilibraE Methods, Target rgap: 1e-10
MSA
FW
CFW
BFW
Relative Gap
Time (s)
Markers every 50 iterations
0
100
200
300
400

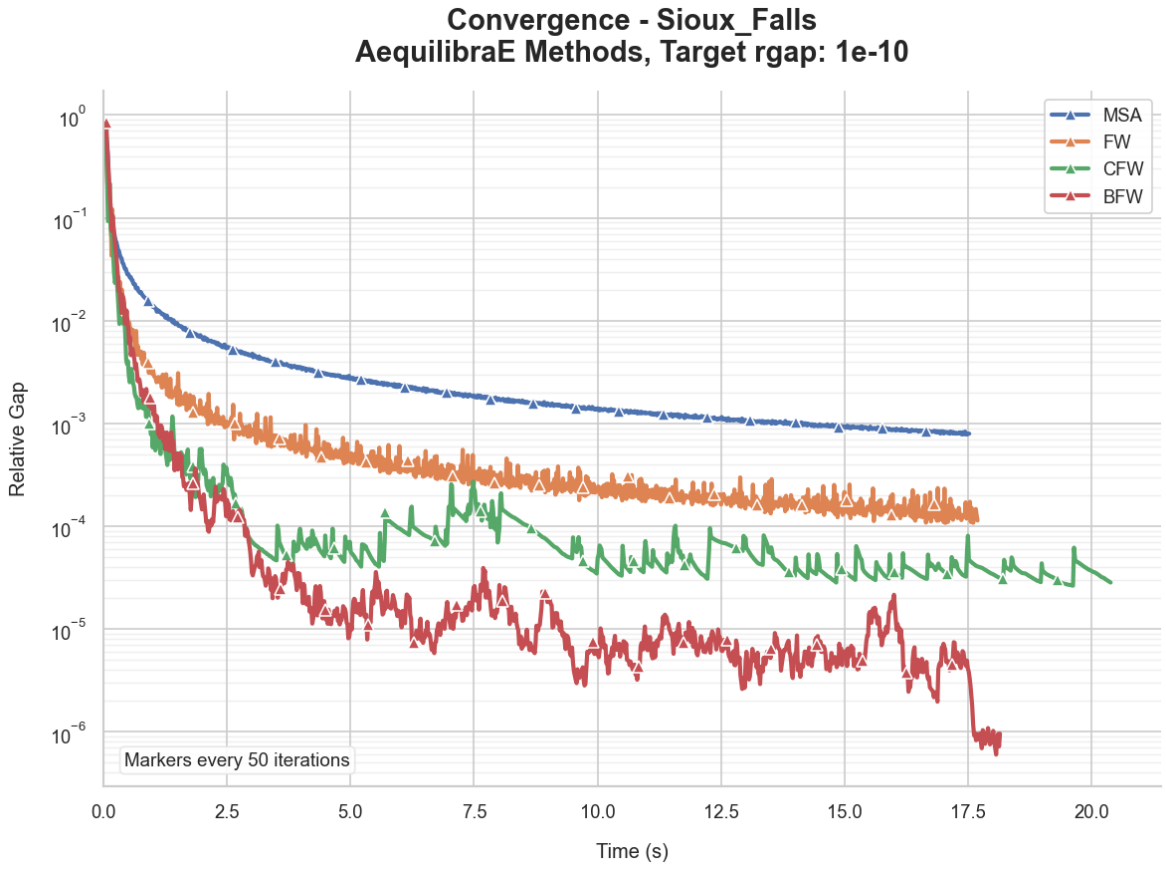
Winnipeg


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 substantially better than seen on previous versions and 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.paths.graph()*
- *aequilibrae.paths.traffic_class.TrafficClass()*
- *aequilibrae.paths.traffic_assignment.TrafficAssignment()*

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

from aequilibrae.utils.create_example import create_example
from aequilibrae.paths import TrafficAssignment, TrafficClass
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
logger = project.logger
```

Traffic assignment

We build all graphs

```
project.network.build_graphs()
# 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
graph = project.network.graphs["c"]

# Let's say we want to minimize the free_flow_time
graph.set_graph("free_flow_time")

# And we will allow paths to be computed going through other centroids/centroid␣
↪connectors
# required for the Sioux Falls network, as all nodes are centroids
graph.set_blocked_centroid_flows(False)
```

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.list()
```

We get the demand matrix, and prepare it for computation

```
demand = proj_matrices.get_matrix("demand_omx")
demand.computational_view(["matrix"])
```

Let's perform the traffic assignment

```
# Create the assignment class
assigclass = TrafficClass(name="car", graph=graph, matrix=demand)

assig = TrafficAssignment()

# We start by adding the list of traffic classes to be assigned
assig.add_class(assigclass)

# Then we set these parameters, which an only be configured after adding one class to␣
↪the assignment
assig.set_vdf("BPR")  # This is not case-sensitive

# Then we set the volume delay function and its parameters
assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# The capacity and free flow travel times as they exist in the graph
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
assig.set_algorithm("bfw")

# Let's set parameters that make this example run very fast
```

(continues on next page)

(continued from previous page)

```
assig.max_iter = 10
assig.rgap_target = 0.01

# we then execute the assignment
assig.execute()
```

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"].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.paths.graph()*
- *aequilibrae.paths.traffic_class.TrafficClass()*
- *aequilibrae.paths.traffic_assignment.TrafficAssignment()*
- *aequilibrae.matrix.aequilibrae_matrix()*

```
# Imports
import os
import pandas as pd
import numpy as np
from uuid import uuid4
from tempfile import gettempdir

from aequilibrae.matrix import AequilibraeMatrix
```

(continues on next page)

(continued from previous page)

```
from aequilibrae.paths import Graph
from aequilibrae.paths import TrafficAssignment
from aequilibrae.paths.traffic_class 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 AequilibraE Matrix with out data

```
aemfile = os.path.join(folder, "demand.aem")
aem = AequilibraeMatrix()
kwargs = {'file_name': aemfile,
          'zones': zones,
          'matrix_names': ['matrix']}

aem.create_empty(**kwargs)
aem.matrix['matrix'][:,:] = mtx[:,:]
aem.index[:] = index[:]
```

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, this is required if you plan to use the *A** path finding, otherwise it can safely 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")
```

Let's build our Graph! In case you're in doubt about AequilibraE Graph, *click here* to read more about it.

```
g = Graph()
g.cost = network['free_flow_time'].values
g.capacity = network['capacity'].values
g.free_flow_time = network['free_flow_time'].values

g.network = network
g.prepare_graph(index)
g.set_graph("free_flow_time")
g.cost = np.array(g.cost, copy=True)
g.set_skimming(["free_flow_time"])
g.set_blocked_centroid_flows(False)
g.network["id"] = g.network.link_id
g.lonlat_index = geom.loc[g.all_nodes]
```

Let's prepare our matrix for computation

```
aem.computational_view(["matrix"])
```

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("car", g, aem)

assig = TrafficAssignment()

assig.set_classes([assigclass])
assig.set_vdf("BPR")
assig.set_vdf_parameters({"alpha": "b", "beta": "power"})
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")
assig.set_algorithm("fw")
assig.max_iter = 100
assig.rgap_target = 1e-6
assig.execute()
```

Now let's take a look at the Assignment results

```
assig.results()
```

And at the Assignment report

```
assig.report()
```

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 assignnment with select link analysis.

See also

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

- *aequilibrae.paths.graph()*
- *aequilibrae.paths.traffic_class.TrafficClass()*
- *aequilibrae.paths.traffic_assignment.TrafficAssignment()*
- *aequilibrae.distribution.ipf()*
- *aequilibrae.distribution.gravity_calibration()*
- *aequilibrae.distribution.gravity_application()*
- *aequilibrae.distribution.synthetic_gravity_model()*

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

import pandas as pd

from aequilibrae.utils.create_example import create_example
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr)
logger = project.logger
```

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 aequilibrae.paths import TrafficAssignment, TrafficClass
```

```
# We build all graphs
project.network.build_graphs()
# 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
graph = project.network.graphs["c"]

# Let's say we want to minimize the free_flow_time
graph.set_graph("free_flow_time")

# And will skim time and distance while we are at it
graph.set_skimming(["free_flow_time", "distance"])

# And we will allow paths to be computed going through other centroids/centroid␣
↪connectors
# required for the Sioux Falls network, as all nodes are centroids
graph.set_blocked_centroid_flows(False)
```

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.list()
```

We get the demand matrix, and prepare it for computation

```
demand = proj_matrices.get_matrix("demand_omx")
demand.computational_view(["matrix"])
```

Let's perform the traffic assignment

```
# Create the assignment class
assigclass = TrafficClass(name="car", graph=graph, matrix=demand)

assig = TrafficAssignment()

# We start by adding the list of traffic classes to be assigned
assig.add_class(assigclass)

# Then we set these parameters, which an only be configured after adding one class to␣
↪the assignment
assig.set_vdf("BPR")  # This is not case-sensitive

# Then we set the volume delay function and its parameters
assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# The capacity and free flow travel times as they exist in the graph
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
assig.set_algorithm("bfw")
```

(continues on next page)

(continued from previous page)

```
# Since we haven't checked the parameters file, let's make sure convergence criteria␣
↪is good
assig.max_iter = 1000
assig.rgap_target = 0.001

# we then execute the assignment
assig.execute()
```

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
```

We need the demand matrix and to prepare it for computation

```
demand = proj_matrices.get_matrix("demand_omx")
demand.computational_view(["matrix"])
```

We also need the skims we just saved into our project

```
imped = proj_matrices.get_matrix("base_year_assignment_skims_car")

# We can check which matrix cores were created for our skims to decide which one to␣
↪use
imped.names
```

Where `free_flow_time_final` is actually the congested time for the last iteration

But before using the data, let's get some impedance for the intrazonals. Let's assume it is 75% of the closest zone.

```
imped_core = "free_flow_time_final"
imped.computational_view([imped_core])

# If we run the code below more than once, we will be overwriting the diagonal values␣
↪with non-sensical data
# so let's zero it first
np.fill_diagonal(imped.matrix_view, 0)

# We compute it with a little bit of NumPy magic
intrazonals = np.amin(imped.matrix_view, where=imped.matrix_view > 0, initial=imped.
↪matrix_view.max(), axis=1)
intrazonals *= 0.75

# Then we fill in the impedance matrix
np.fill_diagonal(imped.matrix_view, intrazonals)
```

Since we are working with an OMX file, we cannot overwrite a matrix on disk. So let's give it a new name to save.

```
imped.save(names=["final_time_with_intrazonals"])
```

This also updates these new matrices as those being used for computation

```
imped.view_names
```

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
```

(continues on next page)

(continued from previous page)

```
    # We save the model
    model.save(join(fldr, f"{function}_model.mod"))

    _ = plot_tlfd(gc.result_matrix.matrix_view, imped.matrix_view, 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_view, imped.matrix_view, "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_view, axis=1)
destinations = np.sum(demand.matrix_view, 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(**args)
ipf.fit()
```

When saving our vector into the project, we'll get an output that it was recorded

```
ipf.save_to_project(name="demand_ipfd_omx", 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")
imped.computational_view(["final_time_with_intrazonals"])
```

If we wanted the main diagonal to not be considered…

```
# np.fill_diagonal(imped.matrix_view, 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(**args)
    gravity.apply()

    # We get the output matrix and save it to OMX too,
    gravity.save_to_project(name=f"demand_{function}_modeled", file_name=f"demand_
↪{function}_modeled.omx")
```

We update the matrices table/records and verify that the new matrices are indeed there

```
proj_matrices.update_database()
proj_matrices.list()
```

Traffic assignment with Select Link Analysis

We'll perform traffic assignment for the future year.

```
logger.info("\n\n\n TRAFFIC ASSIGNMENT FOR FUTURE YEAR WITH SELECT LINK ANALYSIS")
```

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")
demand.names
```

Let's prepare our data for computation

```
demand.computational_view("matrix")
```

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", graph=graph, matrix=demand)

assig = TrafficAssignment()

# Add at a list of traffic classes to be assigned
assig.add_class(assigclass)

assig.set_vdf("BPR")

# Set the volume delay function and its parameters
assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# Set the capacity and free flow travel times as they exist in the graph
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
assig.set_algorithm("bfw")

# Once again we haven't checked the parameters file, so let's make sure convergence␣
↪criteria is good
assig.max_iter = 500
assig.rgap_target = 0.00001
```

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.set_select_links(select_links)
```

(continues on next page)

(continued from previous page)

```
# we then execute the assignment
assig.execute()
```

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()
```

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/

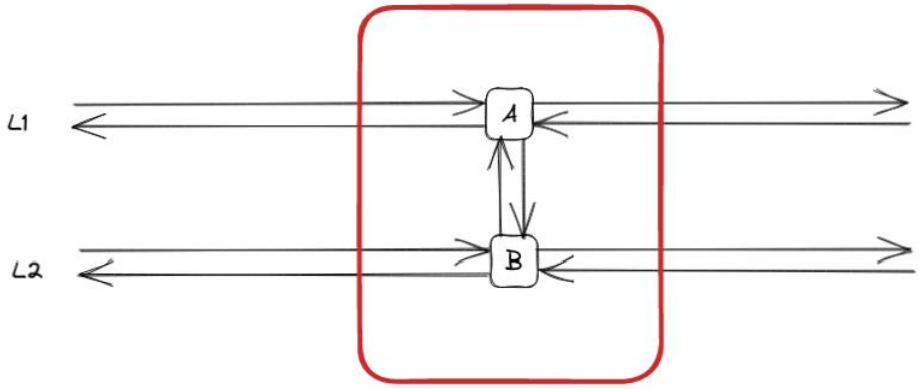
[1] Spiess, H. and Florian, M. (1989) "Optimal strategies: A new assignment model for transit networks". Transportation Research Part B: Methodological, 23(2), 83-102. Available in: https://doi.org/10.1016/0191-2615(89)90034-9

- **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 ($f = 1/s$).

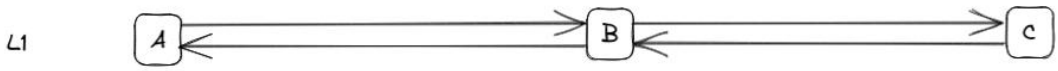
8.1.1 Elements of a transit network

These are the elements required to describe an assignment graph.

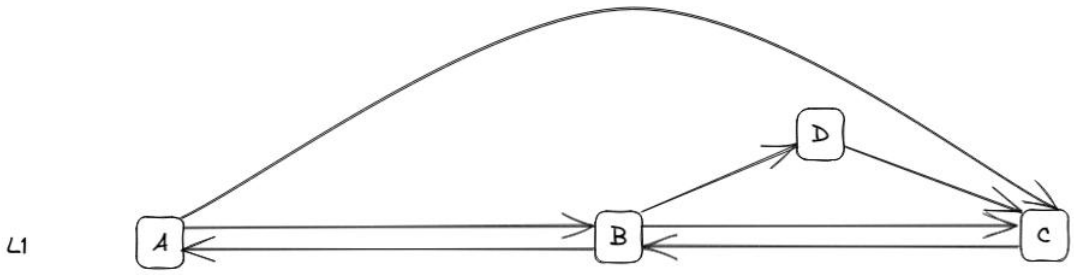
- **Transit stops and stations**: transit stops are points where passenger can board, alight or change vehicles. Also, they can be part of larger stations. In the illustratation below, two distinct stops ('A' and 'B') are highlighted, and they are both affiliated with the same station (depicted in red).



- **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

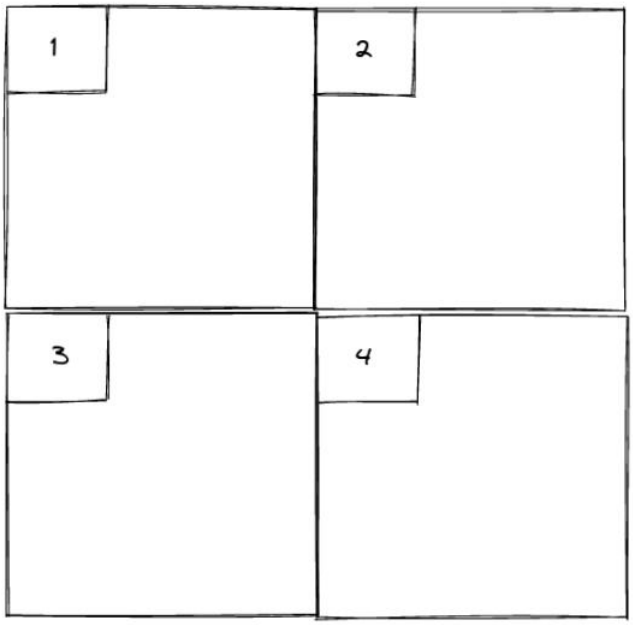
Associated with each sub-line, the headway corresponds to the mean time range between consecutive vehicles — the inverse of the line frequency used as a link attribute in the assignment algorithm.

- **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:

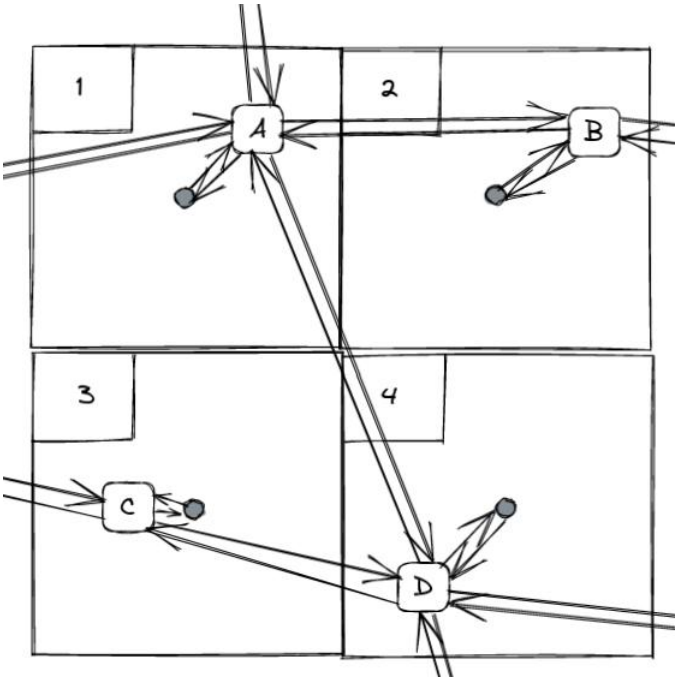
Line ID	Segment Index	Origin Stop	Destination Stop	Travel Time
L1_a1	1	A	B	300
L1_a1	2	B	C	600

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.

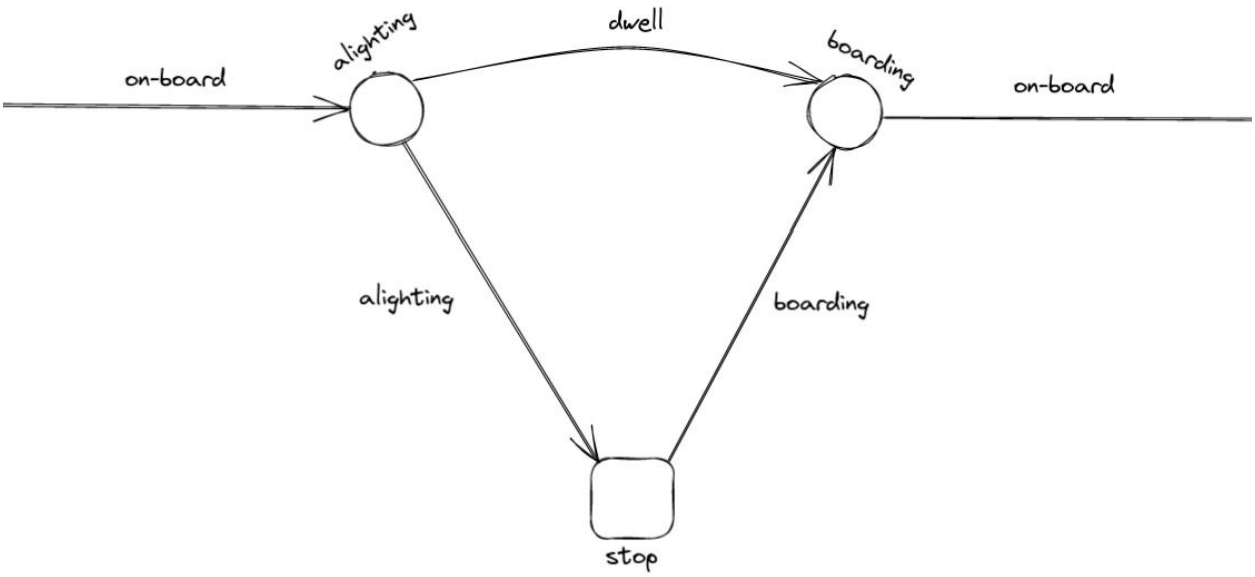


8.1.2 The assignment graph

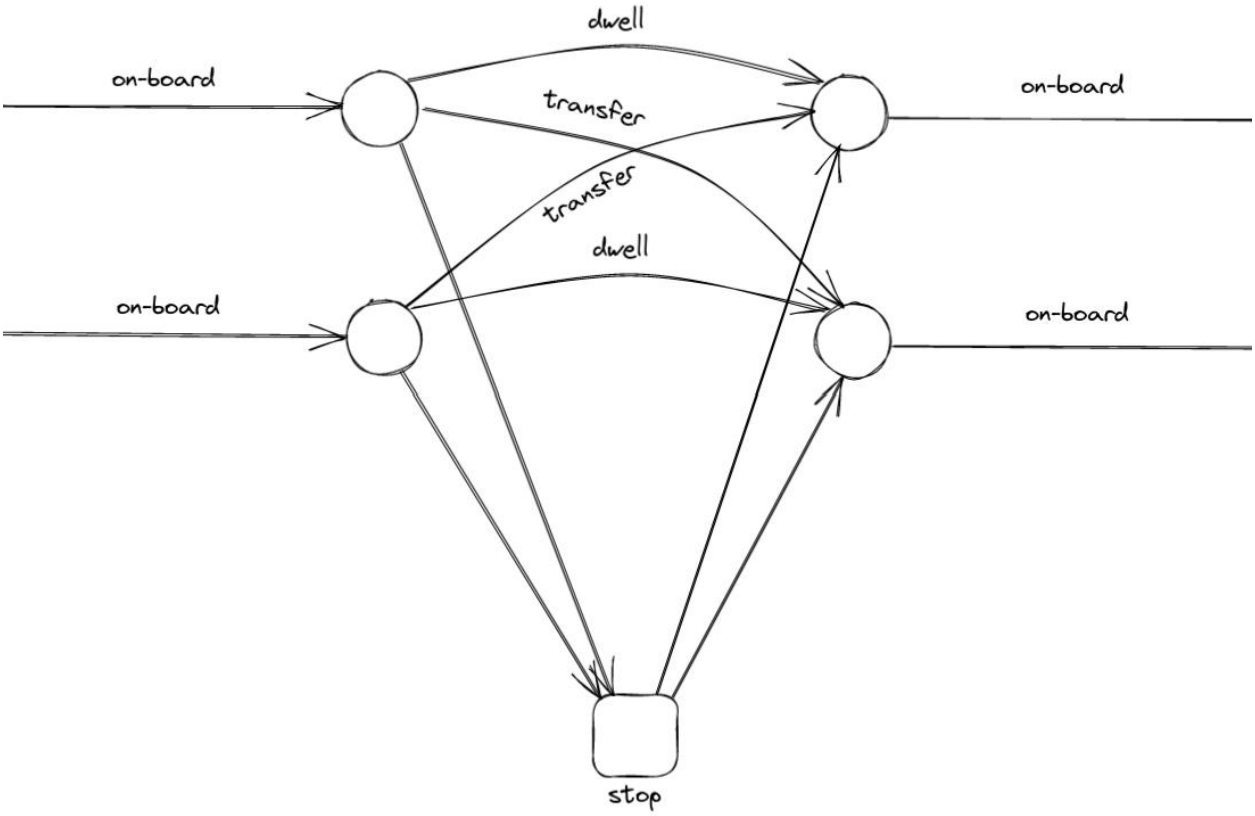
The transit network is used to generate a graph with specific nodes and links used to model the transit process. Various link types and node categories play crucial roles in this representation.

Link types	Node types
On-board	Stop
Boarding	Boarding
Alighting	Alighting
Dwell	OD
Transfer	Walking
Connector	
Walking	

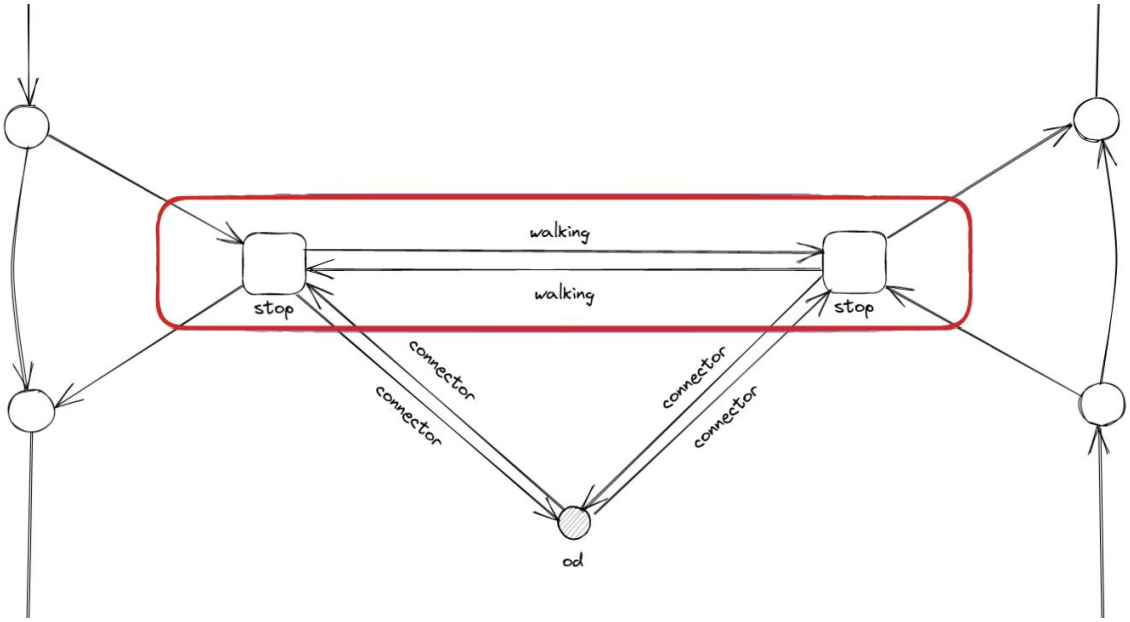
To illustrate, consider the anatomy of a simple stop (figure below). Waiting links encompass boarding and transfer links. Each line segment is associated with a boarding, an on-board and an alighting link.



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.

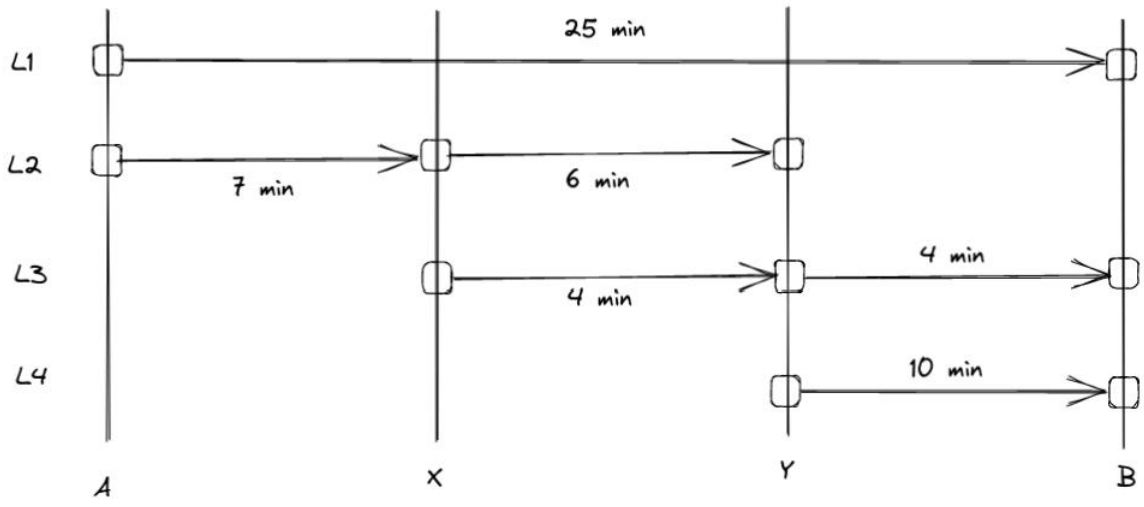
Additional attributes can be introduced for specific link types, such as:

- `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 187, 1].

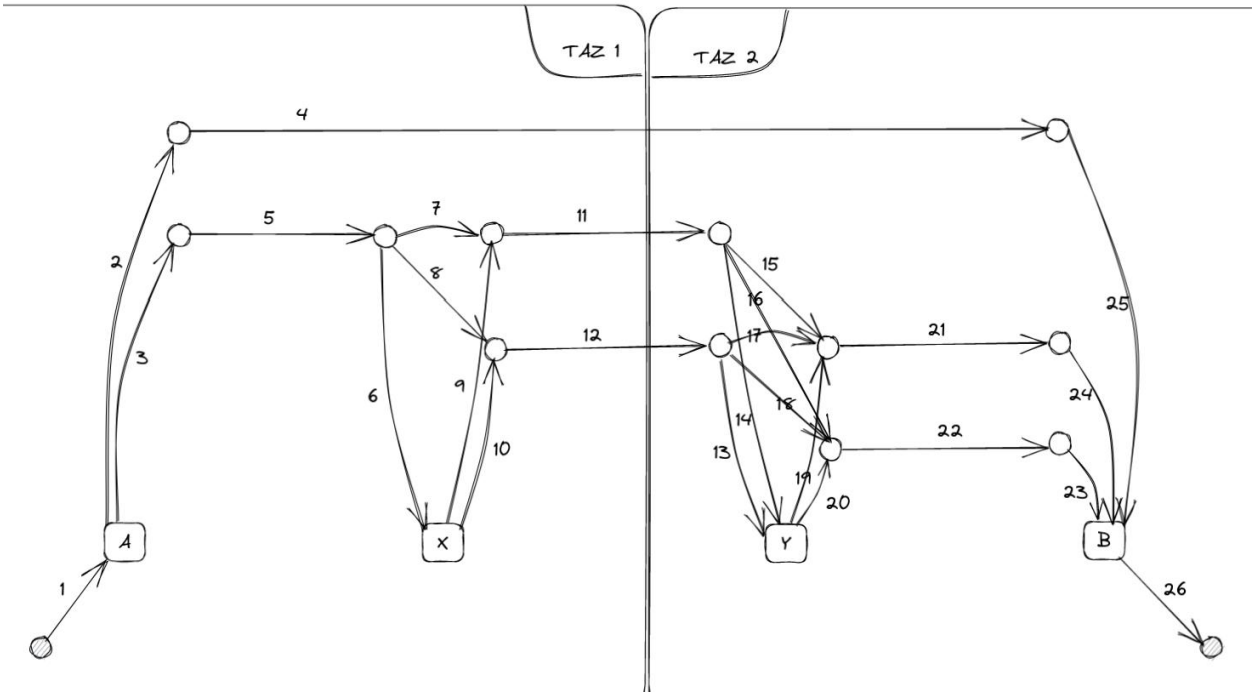
The following figure presents the travel times for each line.



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_flow`: duplicate OD nodes into unconnected origin and destination nodes in order to block centroid flows. Flows starts from an origin node and ends 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 196, 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.

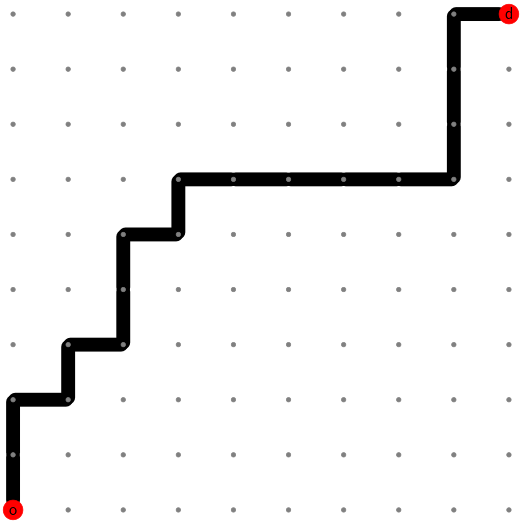
Listing 0: Hyperpath computation

```
# Let's import some packages
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
import pandas as pd

from aequilibrae.paths.cython.public_transport import HyperpathGenerating
from numba import jit

RANDOM_SEED = 124  # random seed
```

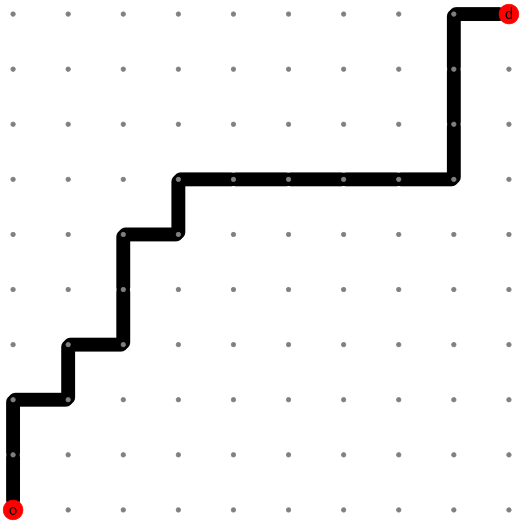
(continues on next page)

[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.

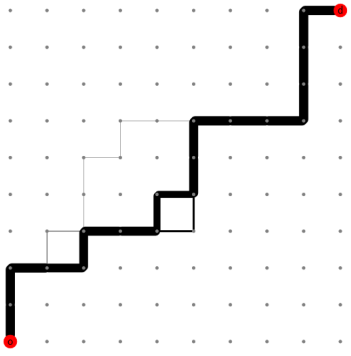
Shortest hyperpath - Bell's network α=0.0



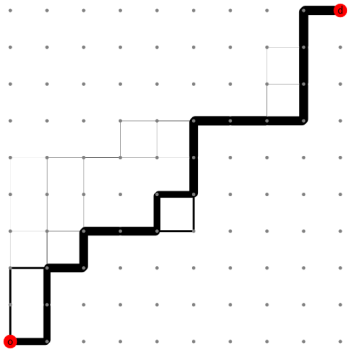
Shortest path - Bell's network



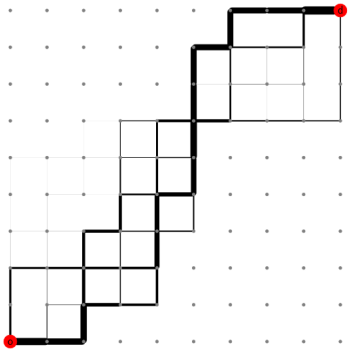
Shortest hyperpath - Bell's network α=0.5



Shortest hyperpath - Bell's network α=1.0



Shortest hyperpath - Bell's network α=100.0



(continued from previous page)

```
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
    else:
        edges["freq"] = edges.freq_base / alpha
```

(continues on next page)

(continued from previous page)

```
    # Spiess & Florian
    sf = HyperpathGenerating(
        edges, tail="tail", head="head", trav_time="trav_time", freq="freq"
    )
    sf.run(origin=0, destination=n * n - 1, volume=1.0)

    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
    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,
```

(continues on next page)

(continued from previous page)

```
        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'*.

```
>>> from aequilibrae.paths import TransitAssignment, TransitClass

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

>>> graph = data.create_graph(
...     with_outer_stop_transfers=False,
...     with_walking_edges=False,
...     blocking_centroid_flows=False,
...     connector_method="overlapping_regions",
... )
>>> project.network.build_graphs()

>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=num_zones, matrix_names=["pt"], memory_only=True)
>>> mat.index = transit_graph.centroids[:]
>>> mat.matrices[:, :, 0] = np.full((num_zones, num_zones), 1.0)
>>> mat.computational_view()

>>> # 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")
>>> assigclass.set_demand_matrix_core("pt")

>>> 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.paths import TransitAssignment, TransitClass

>>> project = create_example(f"{project_path}v2", "coquimbo")
>>> data = Transit(project)

>>> graph = data.create_graph(
...     with_outer_stop_transfers=False,
...     with_walking_edges=False,
...     blocking_centroid_flows=False,
...     connector_method="overlapping_regions",
... )
>>> project.network.build_graphs()

>>> 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(
```

(continues on next page)

(continued from previous page)

```
...       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()
>>> mat.create_empty(zones=num_zones, matrix_names=["pt"], memory_only=True)
>>> mat.index = transit_graph.centroids[:]
>>> mat.matrices[:, :, 0] = np.full((num_zones, num_zones), 1.0)
>>> mat.computational_view()

>>> # 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")
>>> assigclass.set_demand_matrix_core("pt")

>>> 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.transit.transit()*
- *aequilibrae.transit.lib_gtfs()*

```
# 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.transit import Transit
from aequilibrae.utils.create_example 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.

```
remove(join(fldr, "public_transport.sqlite"))
```

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 = Transit(project)

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()
```

(continues on next page)

(continued from previous page)

```
# 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 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.transit.transit()*
- *aequilibrae.transit.transit_graph_builder()*
- *aequilibrae.paths.traffic_class.TransitClass()*

- *aequilibrae.paths.traffic_assignment.TransitAssignment()*
- *aequilibrae.matrix.aequilibrae_matrix()*

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.paths import TransitAssignment, TransitClass
from aequilibrae.transit import Transit
from aequilibrae.transit.transit_graph_builder import TransitGraphBuilder
from aequilibrae.utils.create_example 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 create our `Transit` object.

```
data = Transit(project)
```

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"
)
```

Connector project matching

```
project.network.build_graphs()
graph.create_line_geometry(method="connector project match", graph="c")
data.save_graphs()
data.load()

# Reading back into AequilibraE
graph_db = TransitGraphBuilder.from_db(project, project.network.periods.default_
```

(continues on next page)

(continued from previous page)

```
↪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()
```

```
# Mock demand matrix
zones = len(transit_graph.centroids)
mat = AequilibraeMatrix()
mat.create_empty(zones=zones, matrix_names=['pt'], memory_only=True)
mat.index = transit_graph.centroids[:]
mat.matrices[:, :, 0] = np.full((zones, zones), 1.0)
mat.computational_view()
```

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()

assig.add_class(assigclass)

# Set assignment
assig.set_time_field("trav_time")
assig.set_frequency_field("freq")
assig.set_skimming_fields(["trav_time", "boardings", "freq"])
assig.set_algorithm("os")
assigclass.set_demand_matrix_core("pt")

# 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"].matrix["boardings"].sum()
```

Saving results

We'll be saving the skimming results.

```
assig.save_results(table_name='hyperpath example')
```

Wrapping up

```
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.transit.transit()*
- *aequilibrae.transit.transit_graph_builder()*
- *aequilibrae.paths.traffic_class.TransitClass()*
- *aequilibrae.paths.traffic_assignment.TransitAssignment()*
- *aequilibrae.matrix.aequilibrae_matrix()*

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

from aequilibrae.transit import Transit
from aequilibrae.utils.create_example 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 create our `Transit` object.

```
data = Transit(project)
```

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")
```

(continues on next page)

(continued from previous page)

```
# We drop geometry here for the sake of display.
graph.vertices.drop(columns="geometry")
```

```
graph.edges
```

The graphs also also stored in the `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 `Transit.create_graph` call. You can view previously created periods with the `Periods` object.

```
periods = project.network.periods
periods.data
```

Connector project matching

```
project.network.build_graphs()
```

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 create back in a particular graph via it's 'period_id'.

```
from aequilibrae.transit.transit_graph_builder import TransitGraphBuilder
```

```
graph_db = TransitGraphBuilder.from_db(project, periods.default_period.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)

names_list = ['pt']

mat = AequilibraeMatrix()
mat.create_empty(zones=zones_in_the_model,
                 matrix_names=names_list,
                 memory_only=True)
mat.index = transit_graph.centroids[:]
mat.matrices[:, :, 0] = np.full((zones_in_the_model, zones_in_the_model), 1.0)
mat.computational_view()
```

Hyperpath generation/assignment

We'll create a `TransitAssignment` object as well as a `TransitClass`

```
from aequilibrae.paths import TransitAssignment, TransitClass
```

Create the assignment class

```
assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)

assig = TransitAssignment()

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")
```

When there's multiple matrix cores we'll also need to set the core to use for the demand as we can only assign one at a time.

```
assigclass.set_demand_matrix_core("pt")
```

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()
```

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 Link-Penalization

The link Penalization (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.

[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.

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.

9.1.2 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.

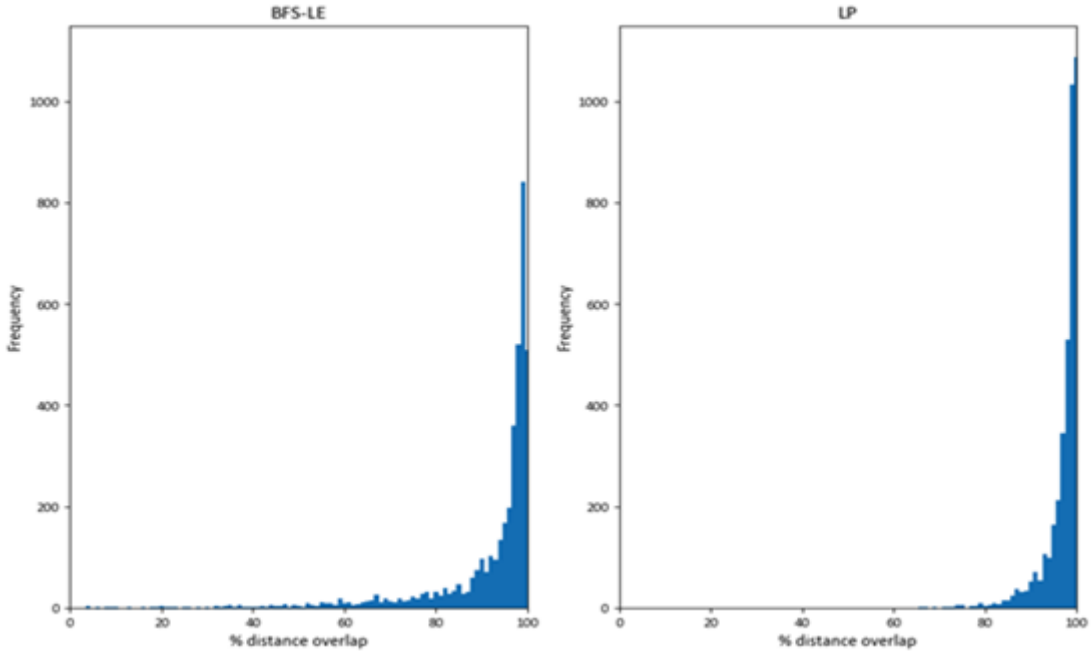
To efficiently store and determine the uniqueness of a new route or removed link sets, we used modified hash functions with properties that allowed us to store and nest them within standard C++ data structures. We used a commutative hash function for the removed link sets to allow for amortised $O(1)$ 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.

9.1.3 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.



[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 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.

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.

9.2.1 Binary logit filter

A binary logit filter is available to remove unfavourable routes from the route set before applying the path-sized logit assignment. This filters accepts a numerical parameter for the minimum demand share acceptable for any path, which is approximated by the binary logit considering the shortest path and each subsequent path.

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].

[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

[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.

9.2.3 References

9.3 Examples

9.3.1 Route Choice

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.paths.route_choice()*

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

import folium
import numpy as np
from aequilibrae.utils.create_example 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)
```

Let's build all graphs

```
project.network.build_graphs()
# 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

```
graph = project.network.graphs["c"]

# we also see what graphs are available
project.network.graphs.keys()

graph.set_graph("distance")

# We set the nodes of interest as centroids to make sure they are not simplified away␣
↪when we create the network
graph.prepare_graph(np.array(nodes_of_interest))
```

Route Choice class

Here we'll construct and use the Route Choice class to generate our route sets

```
from aequilibrae.paths import RouteChoice
```

This object construct might take a minute depending on the size of the graph due to the construction of the compressed link to network link mapping that's required. This is a one time operation per graph and is cached.

```
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

links = project.network.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.network.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.paths.graph()*
- *aequilibrae.paths.route_choice()*
- *aequilibrae.matrix.aequilibrae_matrix()*

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

(continues on next page)

(continued from previous page)

```
from os.path import join

from aequilibrae.utils.create_example import create_example
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
import logging
import sys
```

```
# When the project opens, we can tell the logger to direct all messages to the␣
↪terminal as well
logger = project.logger
stdout_handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s;%(levelname)s ; %(message)s")
stdout_handler.setFormatter(formatter)
logger.addHandler(stdout_handler)
```

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 build all graphs

```
project.network.build_graphs()
# We get warnings that several fields in the project are filled with NaNs.
# This is true, but we won't use those fields.
```

We also see what graphs are available

```
project.network.graphs.keys()
```

We grab the graph for cars

```
graph = project.network.graphs["c"]
```

(continues on next page)

(continued from previous page)

```
# Let's say that utility is just a function of distance, so we build our 'utility'␣
↪field as distance * theta
graph.network = graph.network.assign(utility=graph.network.distance * theta)

# Prepare the graph with all nodes of interest as centroids
graph.prepare_graph(np.array(nodes_of_interest))

# And set the cost of the graph the as the utility field just created
graph.set_graph("utility")
```

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

names_list = ["demand", "5x demand"]

mat = AequilibraeMatrix()
mat.create_empty(zones=graph.num_zones, matrix_names=names_list, memory_only=True)
mat.index = graph.centroids[:]
mat.matrices[:, :, 0] = np.full((graph.num_zones, graph.num_zones), 10.0)
mat.matrices[:, :, 1] = np.full((graph.num_zones, graph.num_zones), 50.0)
mat.computational_view()
```

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.network.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,
            }
        },
```

(continues on next page)

(continued from previous page)

```
    )
```

Route Choice class

Here we'll construct and use the Route Choice class to generate our route sets

```
from aequilibrae.paths import RouteChoice
```

This object construct might take a minute depending on the size of the graph due to the construction of the compressed link to network link mapping that's required. This is a one time operation per graph and is cached.

```
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.add_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 matrices are sparse and can be converted to SciPy sparse matrices for ease of use. They're stored in a dictionary where the key is the matrix name concatenated with the select link set name via an underscore.

```
rc.get_select_link_od_matrix_results()
```

```
od_matrix = rc.get_select_link_od_matrix_results()["sl1"]["demand"]
od_matrix.to_scipy().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.paths.graph()*
- *aequilibrae.paths.route_choice()*
- *aequilibrae.paths.sub_area()*
- *aequilibrae.matrix.aequilibrae_matrix()*

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

import pandas as pd
import numpy as np
import folium

from aequilibrae.utils.create_example import create_example
```

```
# We create the example project inside our temp folder
fldr = join(gettempdir(), uuid4().hex)

project = create_example(fldr, "coquimbo")
```

```
import logging
import sys
```

```
# We the project opens, we can tell the logger to direct all messages to the terminal
↪as well
logger = project.logger
stdout_handler = logging.StreamHandler(sys.stdout)
formatter = logging.Formatter("%(asctime)s;%(levelname)s ; %(message)s")
stdout_handler.setFormatter(formatter)
logger.addHandler(stdout_handler)
```

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

```
project.network.build_graphs()
# 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

```
graph = project.network.graphs["c"]
```

We also see what graphs are available

```
project.network.graphs.keys()
```

Let's say that utility is just a function of distance. So we build our *utility* field as the $distance * theta$.

```
graph.network = graph.network.assign(utility=graph.network.distance * theta)
```

Prepare the graph with all nodes of interest as centroids

```
graph.prepare_graph(graph.centroids)
```

And set the cost of the graph the as the utility field just created

```
graph.set_graph("utility")
```

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

names_list = ["demand"]

mat = AequilibraeMatrix()
mat.create_empty(zones=graph.num_zones, matrix_names=names_list, memory_only=True)
mat.index = graph.centroids[:]
mat.matrices[:, :, 0] = np.full((graph.num_zones, graph.num_zones), 10.0)
mat.computational_view()
```

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.zoning.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.paths import SubAreaAnalysis

subarea = SubAreaAnalysis(graph, zones, mat)
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.prepare_graph(new_centroids)
graph.set_graph("utility")
new_centroids
```

We can then perform an assignment using our new demand matrix on the limited graph

```
from aequilibrae.paths import RouteChoice

rc = RouteChoice(graph)
rc.add_demand(demand)
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],
    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.network.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={
```

(continues on next page)

(continued from previous page)

```
            "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.network.links.data
inner_links = links[links.crosses(poly.boundary)].sort_index()
inner_links.head()
```

As well as which nodes are interior.

```
nodes = project.network.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 graph links, dropping any dead ends and creating a *link_id*, *dir* multi-index.

```
g = (
    graph.graph.set_index("link_id")
    .loc[inner_links.link_id]
    .drop(graph.dead_end_links, errors="ignore")
    .reset_index()
    .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.

```
edge_pairs = {x: (x,) for x in itertools.permutations(g.index, r=2)}
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

```
project.network.build_graphs()
graph = project.network.graphs["c"]
graph.network = graph.network.assign(utility=graph.network.distance * theta)
graph.prepare_graph(graph.centroids)
graph.set_graph("utility")
graph.set_blocked_centroid_flows(False)
```

This object construction might take a minute depending on the size of the graph due to the construction of the compressed link to network link mapping that's required. This is a one time operation per graph and is cached. We need to supply a Graph and an AequilibraeMatrix or DataFrame via the `add_demand` method, if demand is not provided link loading cannot be performed.

```
rc = RouteChoice(graph)
rc.add_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. This filter is also possible with the automated approach, just edit the `subarea.rc.demand.df` DataFrame, however make sure the index remains intact.

```
ods_pairs_of_interest = [
    (4, 39),
    (92, 37),
    (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]
rc.demand.df = rc.demand.df.loc[ods_pairs_of_interest].sort_index().astype(np.float32)
rc.demand.df
```

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 convert the sparse matrices to SciPy COO matrices.

```
sl_od = rc.get_select_link_od_matrix_results()
edge_totals = {k: sl_od[k]["demand"].to_scipy() for k in single_edges}
edge_pair_values = {k: sl_od[k]["demand"].to_scipy() 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 = graph.all_nodes[o]
        d = graph.all_nodes[d]

        o_inside = o in inside_nodes.index
        d_inside = d in inside_nodes.index

        if o_inside and not d_inside:
            exited[o, graph.all_nodes[link.b_node]] += load
        elif not o_inside and d_inside:
            entered[graph.all_nodes[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_inside = o in inside_nodes.index
        d_inside = d in inside_nodes.index

        if not o_inside and not d_inside:
            through[graph.all_nodes[link1.a_node], graph.all_nodes[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.

```
demand = pd.DataFrame(
    list(entered.values()) + list(exited.values()) + list(through.values()),
    index=pd.MultiIndex.from_tuples(
        list(entered.keys()) + list(exited.keys()) + list(through.keys()), names=[
↪"origin id", "destination id"]
    ),
    columns=["demand"],
).sort_index()
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().reshape(-1))
graph.prepare_graph(new_centroids)
graph.set_graph("utility")
new_centroids
```

Re-perform our assignment

```
rc = RouteChoice(graph)
rc.add_demand(demand)
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.project import Project
from aequilibrae.utils.simwrapper.generate_simwrapper_config import␣
↪SimwrapperConfigGenerator

prj = Project()
prj.open('/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.utils.create_example 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")
demand.computational_view(["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.zoning()*
- *aequilibrae.project.network.nodes()*

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from math import sqrt
from shapely.geometry import Point
import shapely.wkb

from aequilibrae.utils.create_example 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.network
```

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(?)))"
```

(continues on next page)

(continued from previous page)

```
with project.db_connection as conn:
    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):
    nd = nodes.get(i)
    nd.renumber(i + 1300)
```

```
# Now we can add them to the model and add centroids to them while we are at it.
zoning = project.zoning
for i, zone_geo in enumerate(simple_progress(grid, s, "Add zone centroids")):
    zone = zoning.new(i + 1)
    zone.geometry = zone_geo
    zone.save()
    # None means that the centroid will be added in the geometric point of the zone
    # But we could provide a Shapely point as an alternative
    zone.add_centroid(None)
```

Centroid connectors

Let's connect our zone centroids to the network.

```
for zone_id, zone in zoning.all_zones().items():
    # We will connect for walk, with 1 connector per zone
    zone.connect_mode(mode_id="w", connectors=1)

    # 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)
    zone.connect_mode(mode_id="c", link_types="ytrusP", connectors=2)

    # This takes a few minutes to compute, so we will break after processing the␣
↪first 10 zones
    if 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 = nodes.new_centroid(10000)
airport.geometry = Point(166.91749582, -0.54472590)
airport.save()
```

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.

```
airport.connect_mode(mode_id="c", link_types="ytrusP", connectors=1)
```

```
project.close()
```

CHAPTER

ELEVEN

API REFERENCE

aequilibrae

11.1 aequilibrae

class aequilibrae.**AequilibraeMatrix**

Matrix class

static random_name() → Path

Returns a random name for a matrix with root in the temp directory of the user

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> mat = AequilibraeMatrix()
>>> str(mat.random_name())
'/tmp/aequilibrae/Aequilibrae_matrix_...'
```

close()

Removes matrix from memory and flushes all data to disk, or closes the OMX file if that is the case

columns() → ndarray

Returns column vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (np.ndarray): the column totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.columns()
array([357.54256811, 357.45109051, 310.88655449, 276.6783439 ,
       266.70388637, 270.62976319, 266.32888632, 279.6897402 ,
       285.89821842, 242.79743295, 252.34085912, 301.78116548,
       302.97058146, 270.61855294, 264.59944248, 257.83842251,
```

(continues on next page)

(continued from previous page)

```
       276.63310578, 257.74513863, 281.15724257, 271.63886077,
       264.62215032, 252.79791125, 273.18139747, 282.7636574 ])

>>> project.close()
```

computational_view(*core_list: List[str] = None*)

Creates a memory view for a list of matrices that is compatible with Cython memory buffers

It allows for AequilibraE matrices to be used in all parallelized algorithms within AequilibraE

In case of OMX matrices, the computational view is held only in memory

Arguments

core_list (`list`): List with the names of all matrices that need to be in the buffer

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.computational_view(['bike trips', 'walk trips'])
>>> mat.view_names
['bike trips', 'walk trips']
```

copy(*output_name: str = None*, *cores: List[str] = None*, *names: List[str] = None*, *compress: bool = None*, *memory_only: bool = True*)

Copies a list of cores (or all cores) from one matrix file to another one

Arguments

output_name (`str`): Name of the new matrix file. If none is provided, returns a copy in memory only

cores (`list`): List of the matrix cores to be copied

names (`list`, *Optional*): List with the new names for the cores. Defaults to current names

compress (`bool`, *Optional*): Whether you want to compress the matrix or not. Defaults to `False`. Not yet implemented

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
```

(continues on next page)

(continued from previous page)

```
>>> mat.copy(Path(my_folder_path) / 'copy_of_my_matrix.aem',
...          cores=['bike trips', 'walk trips'],
...          names=['bicycle', 'walking'],
...          memory_only=False)
<aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix object at 0x...>

>>> mat2 = AequilibraeMatrix()
>>> mat2.load(Path(my_folder_path) / 'copy_of_my_matrix.aem')
>>> mat2.cores
2
```

create_empty(*file_name: Path = None*, *zones: int = None*, *matrix_names: List[str] = None*, *data_type: dtype = <class 'numpy.float64'>*, *index_names: List[str] = None*, *compressed: bool = False*, *memory_only: bool = True*)

Creates an empty matrix in the AequilibraE format

Arguments

file_name (`Path`): Local path to the matrix file

zones (`int`): Number of zones in the model (Integer). Maximum number of zones in a matrix is 4,294,967,296

matrix_names (`list`): A regular Python list of names of the matrix. Limit is 50 characters each. Maximum number of cores per matrix is 256

data_type (`np.dtype`, *Optional*): Data type of the matrix as NUMPY data types (`np.int32`, `np.int64`, `np.float32`, `np.float64`). Defaults to `np.float64`

index_names (`list`, *Optional*): A regular Python list of names for indices. Limit is 20 characters each. Maximum number of indices per matrix is 256

compressed (`bool`, *Optional*): Whether it is a flat matrix or a compressed one (Boolean - Not yet implemented)

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.num_indices
1
>>> mat.zones
3317
```

create_from_omx(*omx_path: str*, *file_path: str = None*, *cores: List[str] = None*, *mappings: List[str] = None*, *robust: bool = True*, *compressed: bool = False*, *memory_only: bool = True*) → None

Creates an AequilibraeMatrix from an original OpenMatrix

Arguments

omx_path (`Path`): Path to the OMX file one wants to import

file_path (`Path`, *Optional*): Path for the output AequilibraeMatrix

cores (`list`, *Optional*): List of matrix cores to be imported

mappings (`list`, *Optional*): List of the matrix mappings (i.e. indices, centroid numbers) to be imported

robust (`bool`, *Optional*): Boolean for whether AequilibraE should try to adjust the names for cores and indices in case they are too long. Defaults to `True`

compressed (`bool`, *Optional*): Boolean for whether we should compress the output matrix. Not yet implemented

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

`create_from_trip_list`(*path_to_file: str*, *from_column: str*, *to_column: str*, *list_cores: List[str]*) → str

Creates an AequilibraeMatrix from a trip list csv file The output is saved in the same folder as the trip list file

Arguments

path_to_file (`str`): Path for the trip list csv file

from_column (`str`): trip list file column containing the origin zones numbers

to_column (`str`): trip list file column containing the destination zones numbers

list_cores (`list`): list of core columns in the trip list file

`export`(*output_name: Path*, *cores: List[str] = None*)

Exports the matrix to other formats, rather than AEM. Formats currently supported: CSV, OMX

When exporting to AEM or OMX, the user can chose to export only a set of cores, but all indices are exported

When exporting to CSV, the active index will be used, and all cores will be exported as separate columns in the output file

Arguments

output_name (`Path`): Path to the output file

cores (`list`): Names of the cores to be exported.

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.export(Path(my_folder_path) / 'my_new_path.omx', ['Car trips', 'bike␣
↪trips'])
```

`get_matrix`(*core: str*, *copy=False*) → ndarray

Returns the data for a matrix core

Arguments

core (`str`): name of the matrix core to be returned

copy (`bool`, *Optional*): return a copy of the data. Defaults to False

Returns

object (`np.ndarray`): NumPy array

is_omx()

Returns `True` if matrix data source is OMX, `False` otherwise

load(*file_path: Path*)

Loads matrix from disk. All cores and indices are load. First index is default.

Arguments

file_path (`str`): Path to AEM or OMX file on disk

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view()
>>> mat.names
['distance_blended', 'time_final']

>>> project.close()
```

nan_to_num()

Converts all `NaN` values in all cores in the computational view to zeros

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> nan_matrix = np.empty((3,3))
>>> nan_matrix[:] = np.nan

>>> index = np.arange(1, 4, dtype=np.int32)

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / "matrices/nan_matrix.aem
↪",
...                  zones=3,
...                  matrix_names=["only_nan"])
>>> mat.index[:] = index[:]
>>> mat.matrix["only_nan"][:, :] = nan_matrix[:, :]
>>> mat.computational_view()
>>> mat.nan_to_num()
>>> mat.get_matrix("only_nan")
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

rows() → ndarray

Returns row vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (`np.ndarray`): the row totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.rows()
array([357.68202084, 358.68778868, 310.68285491, 275.87964738,
       265.91709918, 268.60184371, 267.32264726, 281.3793747 ,
       286.15085073, 242.60308705, 252.1776242 , 305.56774194,
       303.58100777, 270.48841269, 263.20417379, 253.92665702,
       277.1655432 , 258.84368258, 280.65697316, 272.7651157 ,
       264.06806038, 252.87533845, 273.45639965, 281.61102767])

>>> project.close()
```

save (*names=()*, *file_name=None*) → None

Saves matrix data back to file.

If working with AEM file, it flushes data to disk. If working with OMX, requires new names.

Arguments

names (`tuple(str)`, *Optional*): New names for the matrices. Required if working with OMX files

file_name (`str`, *Optional*): Local path to the matrix file

setDescription (*matrix_description: str*)

Sets description for the matrix

Arguments

matrix_description (`str`): Text with matrix description. Maximum length is 144 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.aem',
...                  zones=zones_in_the_model,
...                  memory_only=False)
>>> mat.setDescription('This is a text')
>>> mat.save()
>>> mat.close()

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(my_folder_path) / 'my_matrix.aem')
>>> mat.description.decode('utf-8')
'This is a text'
```

setName (*matrix_name: str*)

Sets the name for the matrix itself. Only works for matrices in disk.

Arguments

matrix_name (`str`): matrix name. Maximum length is 50 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.omx',
...                  zones=zones_in_the_model,
...                  memory_only=False)
>>> mat.setName('This is my example')
>>> mat.save()
>>> mat.close()
```

set_index(*index_to_set: str*) → None

Sets the standard index to be the one the user wants to have be the one being used in all operations during run time. The first index is ALWAYS the default one every time the matrix is instantiated

Arguments

index_to_set (str): Name of the index to be used. The default index name is 'main_index'

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']
>>> index_list = ['tazs', 'census']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list,
...                  index_names=index_list )
>>> mat.num_indices
2
>>> mat.current_index
'tazs'
>>> mat.set_index('census')
>>> mat.current_index
'census'
```

class aequilibrae.**AssignmentPaths**(*table_name: str*, *project=None*)

Class for accessing path files optionally generated during assignment.

static get_path_for_destination_from_files(*path_o: DataFrame*, *path_o_index: DataFrame*, *destination: int*)

for a given path file and path index file, and a given destination, return the path links in o-d order

get_path_for_destination(*origin: int*, *destination: int*, *iteration: int*, *traffic_class_id: str*)

Return all link ids, i.e. the full path, for a given destination

read_path_file(*origin: int*, *iteration: int*, *traffic_class_id: str) -> (<class 'pandas.DataFrame'>*, *<class 'pandas.DataFrame'>*)

class aequilibrae.**AssignmentResults**

Assignment result holder for a single *TrafficClass* with multiple user classes

get_graph_to_network_mapping()

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (pd.DataFrame): Pandas DataFrame data with the traffic class assignment results

get_sl_results() → DataFrame

prepare(*graph:* Graph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (*Graph*): Needs to have been set with number of centroids and list of skims (if any)

matrix (*AequilibraeMatrix*): Matrix properly set for computation with matrix.computational_view(:obj:`list`)

reset() → None

Resets object to prepared and pre-computation state

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

total_flows() → None

Totals all link flows for this class into a single link load

Results are placed into *total_link_loads* class member

class aequilibrae.**Graph**(**args*, ***kwargs*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (str): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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): heuristic to use if a_star is enabled. Default is None.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()

>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk(*filename: str*) → None

Loads graph from disk

Arguments

filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments

filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments

block_centroid_flows (`bool`): Blocking or not paths to go 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

class aequilibrae.**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
```

(continues on next page)

(continued from previous page)

```
# 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")
>>> matrix.computational_view(["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.export(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(*name: str*, *file_name: str*, *project=None*) → None

Saves the matrix output to the project file

Arguments

name (str): Name of the desired matrix record

file_name (`str`): Name for the matrix file name. AEM and OMX supported

project (*Project*, *Optional*): Project we want to save the results to. Defaults to the active project

class aequilibrae.**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")
>>> demand.computational_view()

# We load the skim matrix
>>> skim = project.matrices.get_matrix("skims")
>>> skim.computational_view(["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.**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")
>>> matrix.computational_view()

>>> vectors = pd.DataFrame(
...     {"productions": np.zeros(matrix.zones), "attractions": np.zeros(matrix.
↪zones)},
...     index=matrix.index,
... )
```

(continues on next page)

(continued from previous page)

```
>>> vectors["productions"] = matrix.rows()
>>> vectors["attractions"] = matrix.columns()

>>> 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.export(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(*name: str*, *file_name: str*, *project=None*) → *MatrixRecord*

Saves the matrix output to the project file

Arguments

name (`str`): Name of the desired matrix record

file_name (`str`): Name for the matrix file name. AEM and OMX supported

project (*Project*, *Optional*): Project we want to save the results to. Defaults to the active project

class aequilibrae.**Log**(*project_base_path: Path*)

API entry point to the log file contents

```
>>> project = Project()
>>> project.new(project_path)

>>> log = project.log()

# We get all entries for the log file
>>> entries = log.contents()

# Or clear everything (NO UN-DOs)
>>> log.clear()

>>> project.close()
```

clear()

Clears the log file. Use it wisely

contents() → list

Returns contents of log file

Returns
log_contents (`list`): List with all entries in the log file

class aequilibrae.**Matrices**(*project*)

Gateway into the matrices available/recorded in the model

check_exists(*name: str*) → bool

Checks whether a matrix with a given name exists

Returns
exists (`bool`): Does the matrix exist?

clear_database() → None

Removes records from the matrices database that do not exist in disk

delete_record(*matrix_name: str*) → None

Deletes a Matrix Record from the model and attempts to remove from disk

get_matrix(*matrix_name: str*) → *AequilibraeMatrix*

Returns an AequilibraE matrix available in the project

Raises an error if matrix does not exist

Arguments
matrix_name (`str`): Name of the matrix to be loaded

Returns
matrix (*AequilibraeMatrix*): Matrix object

get_record(*matrix_name: str*) → *MatrixRecord*

Returns a model Matrix Record for manipulation in memory

list() → DataFrame

List of all matrices available

Returns
df (`pd.DataFrame`): Pandas DataFrame listing all matrices available in the model

new_record(*name: str*, *file_name: str*, *matrix=None*) → *MatrixRecord*

Creates a new record for a matrix in disk, but does not save it

If the matrix file is not already on disk, it will fail

Arguments
name (`str`): Name of the matrix

file_name (`str`): Name of the file on disk

Returns
matrix_record (`MatrixRecord`): A matrix record that can be manipulated in memory before saving

reload()

Discards all memory matrices in memory and loads recreate them

update_database() → None

Adds records to the matrices database for matrix files found on disk

class aequilibrae.**NetworkSkimming**(*graph*, *origins=None*, *project=None*)

```
>>> from aequilibrae.paths.network_skimming import NetworkSkimming

>>> project = create_example(project_path)
>>> project.network.build_graphs(modes=["c"])

>>> graph = project.network.graphs['c']
>>> graph.set_graph("distance")
>>> graph.set_skimming("distance")

>>> skm = NetworkSkimming(graph)
>>> skm.execute()

# The skim report (if any error generated) is available here
>>> skm.report
[]

# To access the skim matrix directly from its temporary file
>>> matrix = skm.results.skims

# Or you can save the results to disk
>>> skm.save_to_project('skimming_result_omx', 'omx')

>>> project.close()
```

doWork()

execute()

Runs the skimming process as specified in the graph

save_to_project(*name: str*, *format='omx'*, *project=None*) → None

Saves skim results to the project folder and creates record in the database

Arguments

name (`str`): Name of the matrix. Same value for matrix record name and file (plus extension)

format (`str`, *Optional*): File format ('aem' or 'omx'). Default is 'omx'

project (*Project*, *Optional*): Project we want to save the results to. Defaults to the active project

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.**Parameters**(*path: Path | None = None*)

Global parameters module.

Parameters are used in many procedures, and are often defined in the `parameters.yml` file ONLY.

Parameters are organized 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.parameters import Parameters

>>> project = Project()
>>> project.new(project_path)

>>> p = 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']['max_query_area_size'] = 10000000000
>>> p.parameters['osm']['sleeptime'] = 0
>>> p.write_back()

>>> # You can also restore the software default values
>>> p.restore_default()

>>> project.close()
```

classmethod load_default()

restore_default()

Restores parameters to generic default

write_back()

Writes the parameters back to file

file_default: Path =
PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/parameters.yml')

class aequilibrae.**PathResults**

Path computation result holder

```
>>> from aequilibrae.paths.results import PathResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize distance
>>> car_graph.set_graph('distance')
```

(continues on next page)

(continued from previous page)

```
# If you want to compute skims
# It does increase path computation time substantially
>>> car_graph.set_skimming(['distance', 'free_flow_time'])

>>> res = PathResults()
>>> res.prepare(car_graph)
>>> res.compute_path(1, 17)

# Update all the outputs mentioned above for destination 9. Same origin: 1
>>> res.update_trace(9)

# clears all computation results
>>> res.reset()

>>> project.close()
```

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*) → None

Computes the path between two nodes in the network.

*A** heuristics are currently only valid distance cost fields.

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`): Heuristic to use if `a_star` is enabled. Default is `None`.

get_heuristics() → List[str]

Return the available heuristics.

prepare(*graph:* Graph) → None

Prepares the object with dimensions corresponding to the graph object

Arguments

graph (*Graph*): Needs to have been set with number of centroids and list of skims (if any)

reset() → None

Resets object to prepared and pre-computation state

set_heuristic(*heuristic: str*) → None

Set the heuristics to be used in A*. Must be one of *get_heuristics()*.

Arguments

heuristic (`str`): Heuristic to use in A*.

update_trace(*destination: int*) → None

Updates the path's nodes, links, skims and mileposts

If the previously computed path had *early_exit* enabled, *update_trace* will check if the *destination* has already been found, if not the shortest path tree will be recomputed with the *early_exit* argument passed on.

If the previously computed path had *a_star* enabled, *update_trace* always recompute the path.

Arguments

destination (`int`): ID of the node we are computing the path too

class aequilibrae.**Project**

AequilibraE project class

Listing 1: Create Project

```
>>> new_project = Project()
>>> new_project.new(project_path)

# Safely closes the project
>>> new_project.close()
```

Listing 2: Open Project

```
>>> existing_project = Project()
>>> existing_project.open(project_path)

>>> existing_project.close()
```

classmethod from_path(*project_folder*)

activate()

check_file_indices() → None

Makes results_database.sqlite and the matrices folder compatible with project database

clone_scenario(*scenario_name: str*, *description: str = ''*)

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 = ''*)

Creates an empty scenario, without any links, nodes, and zones.

Arguments

scenario_name (`str`): scenario name

description (`str`): useful scenario description

deactivate()

list_scenarios()

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

new(*project_path: str*) → None

Creates a new project

Arguments

project_path (`str`): Full path to the project data folder. If folder exists, it will fail

open(*project_path: str*) → None

Loads project from disk

Arguments

project_path (`str`): Full path to the project data folder. If the project inside does not exist, it will fail.

upgrade(*ignore_project: bool = False*, *ignore_transit: bool = False*, *ignore_results: bool = False*)

Find and apply all applicable migrations.

All database upgrades are 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*)

Switch the active scenario.

Arguments

scenario_name (`str`): name of the scenario to be activated

property about: *About*

property db_connection

property db_connection_spatial

Deprecated alias for `db_connection`, which is now a spatial connection.

property logger: Logger

property matrices: *Matrices*

property network: *Network*

property parameters: dict

property path_to_file

property project_base_path

property project_parameters: *Parameters*

property results: *Results*

property results_connection

property run

Load and return the AequilibraE run module with the default arguments from `parameters.yml` partially applied.

Refer to `run/__init__.py` file within the project folder for documentation.

property transit: *Transit*

property transit_connection

property zoning

class aequilibrae.**SkimResults**

Network skimming result holder.

```
>>> from aequilibrae.paths.results import SkimResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize travel time
>>> car_graph.set_graph('free_flow_time')

# Skims travel time and distance
>>> car_graph.set_skimming(['free_flow_time', 'distance'])

>>> res = SkimResults()
>>> res.prepare(car_graph)

>>> res.skims.export(project_path / "skim_matrices.omx")

>>> project.close()
```

prepare(*graph:* Graph)

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)

class aequilibrae.**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

class aequilibrae.**TrafficAssignment**(*project=None*)

Traffic assignment class.

For a comprehensive example on use, see the Use examples page.

```
>>> from aequilibrae.paths import TrafficAssignment, TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(True)

>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")

# We will only assign one user class stored as 'matrix' inside the OMX file
>>> demand.computational_view(['matrix'])

# Creates the assignment class
>>> assigclass = TrafficClass("car", graph, demand)

>>> assig = TrafficAssignment()

# The first thing to do is to add at list of traffic classes to be assigned
>>> assig.set_classes([assigclass])

# Then we set the volume delay function
>>> assig.set_vdf("BPR")  # This is not case-sensitive

# And its parameters
>>> assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# The capacity and free flow travel times as they exist in the graph
>>> assig.set_capacity_field("capacity")
>>> assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
>>> assig.set_algorithm('bfw')

>>> assig.max_iter = 10
>>> assig.rgap_target = 0.00001

>>> assig.execute() # we then execute the assignment
```

(continues on next page)

(continued from previous page)

```
# 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 = assigclass.results.skims # blended ones
>>> last_skims = assigclass._aon_results.skims # those for the last iteration

>>> project.close()
```

add_class(*traffic_class:* TrafficClass) → None

Adds a traffic class to the assignment

Arguments

traffic_class (*TrafficClass*): Traffic class

add_preload(*preload: DataFrame*, *name: str = None*) → None

Given a dataframe of 'link_id', 'direction' and 'preload', merge into current preloads dataframe.

Arguments

preload (`pd.DataFrame`): dataframe mapping 'link_id' & 'direction' to 'preload' **name** (`str`): Name for particular preload (optional - default name will be chosen if not specified)

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns

skim list (`list`): Lists of all skims with the results for each class

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

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 active 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 active 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 active 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*) → None

Saves the skims (if any) to the skim folder and registers in the matrix list

Arguments
name (str): Name of the matrix record to hold this matrix (same name used for file name)

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 ('aem' or 'omx'). Default is 'omx'

project (*Project*, *Optional*): Project we want to save the results to.
Defaults to the active project

select_link_flows() → Dict[str, DataFrame]

Returns a dataframe of the select link flows for each class

set_algorithm(*algorithm: str*)

Chooses the assignment algorithm. e.g. 'frank-wolfe', 'bfw', 'msa'

'fw' is also accepted as an alternative to 'frank-wolfe'

Arguments
algorithm (`str`): Algorithm to be used

set_capacity_field(*capacity_field: str*) → None

Sets the graph field that contains link capacity for the assignment period -> e.g. 'capacity1h'

Arguments
capacity_field (`str`): Field name

set_classes(*classes: List[*TrafficClass*]*) → None

Sets Traffic classes to be assigned

Arguments
classes (`List[TrafficClass]`): List of Traffic classes for assignment

set_cores(*cores: int*) → None

Allows one to set the number of cores to be used AFTER traffic classes have been added

Inherited from `AssignmentResultsBase`

Arguments
cores (`int`): Number of CPU cores to use

set_path_file_format(*file_format: str*) → None

Specify path saving format. Either parquet or feather.

Arguments
file_format (`str`): Name of file format to use for path files

set_save_path_files(*save_it: bool*) → None

Turn path saving on or off.

Arguments
save_it (`bool`): Boolean to indicate whether paths should be saved

set_time_field(*time_field: str*) → None

Sets the graph field that contains free flow travel time -> e.g. 'fftime'

Arguments
time_field (`str`): Field name

set_vdf(*vdf_function: str*) → None

Sets the Volume-delay function to be used

Arguments
vdf_function (`str`): Name of the VDF to be used

set_vdf_parameters (*par: dict*) → None

Sets the parameters for the Volume-delay function.

Parameter values can be scalars (same values for the entire network) or network field names (link-specific values) - Examples: {'alpha': 0.15, 'beta': 4.0} or {'alpha': 'alpha', 'beta': 'beta'}

The Akcelik VDF parameter 'tau' value has typical `8` factor absorbed into it. Users should supply `8 * tau` to match other common usages. Additionally the standard `0.25` factor can be overridden by supplying the 'alpha' parameter.

Arguments

par (`dict`): Dictionary with all parameters for the chosen VDF

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']

bpr_parameters = ['alpha', 'beta']

class aequilibrae.**TrafficClass** (*name: str*, *graph:* Graph, *matrix:* AequilibraeMatrix)

Traffic class for equilibrium traffic assignment

```
>>> from aequilibrae.paths import TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(False)

>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")
>>> demand.computational_view()

>>> tc = TrafficClass("car", graph, demand)
>>> tc.set_pce(1.3)

>>> project.close()
```

set_fixed_cost (*field_name: str*, *multiplier=1*)

Sets value of time

Arguments

field_name (`str`): Name of the graph field with fixed costs for this class

multiplier (`Union[float, int]`): Multiplier for the fixed cost. Defaults to 1 if not set

set_pce(*pce: float | int*) → None

Sets Passenger Car equivalent

Arguments

pce (`Union[float, int]`): PCE. Defaults to 1 if not set

set_select_links(*links: Dict[str, List[Tuple[int, int]]]*)

Set the selected links. Checks if the links and directions are valid. Translates link_id and direction into unique link id used in compact graph. Supply links=None to disable select link analysis.

Arguments

links (`Union[None, Dict[str, List[Tuple[int, int]]]]`): name of link set and Link IDs and directions to be used in select link analysis

set_vot(*value_of_time: float*) → None

Sets value of time

Arguments

value_of_time (`Union[float, int]`): Value of time. Defaults to 1 if not set

skim_congested(*skim_fields=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.

Arguments

skim_fields (`Union[None, str]`): Name of the skims to use. If None, uses default only

property info: dict

class aequilibrae.**VDF**

Volume-Delay function

```
>>> from aequilibrae.paths import VDF

>>> vdf = VDF()
>>> vdf.functions_available()
['bpr', 'bpr2', 'conical', 'inrets', 'akcelik']
```

functions_available() → list

returns a list of all functions available

class aequilibrae.**allOrNothing**(*class_name: str*, *matrix:* AequilibraeMatrix, *graph:* Graph, *results:* AssignmentResults)

doWork()

execute()

func_assig_thread(*origin*, *all_threads*)

signal = <aequilibrae.utils.python_signal.PythonSignal object>

Modules

context
distribution

continues on next page

Table 2 – continued from previous page

log
matrix
parameters
paths
project
reference_files
transit
utils

11.1.1 aequilibrae.context

Functions

activate_project(project)
get_active_project([must_exist])
get_logger()

aequilibrae.context.**activate_project**(*project*)

aequilibrae.context.**get_active_project**(*must_exist=True*)

aequilibrae.context.**get_logger**()

11.1.2 aequilibrae.distribution

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
```

(continues on next page)

(continued from previous page)

```
# We load the impedance matrix
>>> matrix = project.matrices.get_matrix("skims")
>>> matrix.computational_view(["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.export(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(*name: str*, *file_name: str*, *project=None*) → None

Saves the matrix output to the project file

Arguments

name (`str`): Name of the desired matrix record

file_name (`str`): Name for the matrix file name. AEM and OMX supported

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active 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")
>>> demand.computational_view()

# We load the skim matrix
>>> skim = project.matrices.get_matrix("skims")
>>> skim.computational_view(["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")
>>> matrix.computational_view()

>>> vectors = pd.DataFrame(
...     {"productions": np.zeros(matrix.zones), "attractions": np.zeros(matrix.
↪zones)},
...     index=matrix.index,
... )

>>> vectors["productions"] = matrix.rows()
>>> vectors["attractions"] = matrix.columns()

>>> ipf_args = {"matrix": matrix,
...             "vectors": vectors,
...             "row_field": "productions",
...             "column_field": "attractions",
```

(continues on next page)

(continued from previous page)

```
...             "nan_as_zero": False}

>>> fratar = Ipf(**ipf_args)
>>> fratar.fit()

# We can get back to our OMX matrix in the end
>>> fratar.output.export(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(*name: str*, *file_name: str*, *project=None*) → *MatrixRecord*

Saves the matrix output to the project file

Arguments

name (`str`): Name of the desired matrix record

file_name (`str`): Name for the matrix file name. AEM and OMX supported

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active 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

Modules

gravity_application	
gravity_calibration	Algorithms to **calibrate** synthetic gravity models with power and exponential functions
ipf	
ipf_core	
synthetic_gravity_model	

aequilibrae.distribution.gravity_application

Classes

GravityApplication([project])	Applies a synthetic gravity model.

class aequilibrae.distribution.gravity_application.**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")
>>> matrix.computational_view(["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
...         }
```

(continues on next page)

(continued from previous page)

```
>>> gravity = GravityApplication(**args)

# Solve and save the outputs
>>> gravity.apply()
>>> gravity.output.export(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(*name: str*, *file_name: str*, *project=None*) → None

Saves the matrix output to the project file

Arguments

name (`str`): Name of the desired matrix record

file_name (`str`): Name for the matrix file name. AEM and OMX supported

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active project

aequilibrae.distribution.gravity_calibration

Algorithms to **calibrate** synthetic gravity models with power and exponential functions

The procedures implemented in this code are some of those suggested in Modelling Transport, 4th Edition, Ortuzar and Willumsen, Wiley 2011

Classes

GravityCalibration([project])	Calibrate a traditional gravity model

class aequilibrae.distribution.gravity_calibration.**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")
>>> demand.computational_view()

# We load the skim matrix
>>> skim = project.matrices.get_matrix("skims")
>>> skim.computational_view(["time_final"])

>>> args = {"matrix": demand,
```

(continues on next page)

(continued from previous page)

```
...         "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

aequilibrae.distribution.ipf

Classes

Ipf([project])	Iterative proportional fitting procedure

class aequilibrae.distribution.ipf.**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")
>>> matrix.computational_view()

>>> vectors = pd.DataFrame(
...     {"productions": np.zeros(matrix.zones), "attractions": np.zeros(matrix.
↪zones)},
...     index=matrix.index,
... )

>>> vectors["productions"] = matrix.rows()
>>> vectors["attractions"] = matrix.columns()

>>> ipf_args = {"matrix": matrix,
...             "vectors": vectors,
...             "row_field": "productions",
...             "column_field": "attractions",
...             "nan_as_zero": False}

>>> fratar = Ipf(**ipf_args)
>>> fratar.fit()
```

(continues on next page)

(continued from previous page)

```
# We can get back to our OMX matrix in the end
>>> fratar.output.export(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(*name: str*, *file_name: str*, *project=None*) → *MatrixRecord*

Saves the matrix output to the project file

Arguments

name (str): Name of the desired matrix record

file_name (str): Name for the matrix file name. AEM and OMX supported

project (Project, *Optional*): Project we want to save the results to. Defaults to the active project

aequilibrae.distribution.ipf_core

aequilibrae.distribution.synthetic_gravity_model

Classes

SyntheticGravityModel()	Simple class object to represent synthetic gravity models

class aequilibrae.distribution.synthetic_gravity_model.**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

11.1.3 aequilibrae.log

Functions

get_log_handler(log_file[, ensure_file_exists])	Return a log handler that writes to the given log_file

Classes

Log(project_base_path)	API entry point to the log file contents

class aequilibrae.log.**Log**(*project_base_path: Path*)

API entry point to the log file contents

```
>>> project = Project()
>>> project.new(project_path)

>>> log = project.log()

# We get all entries for the log file
>>> entries = log.contents()

# Or clear everything (NO UN-DOs)
>>> log.clear()

>>> project.close()
```

clear()

Clears the log file. Use it wisely

contents() → list

Returns contents of log file

Returns

log_contents (list): List with all entries in the log file

aequilibrae.log.**get_log_handler**(*log_file: Path*, *ensure_file_exists=True*)

Return a log handler that writes to the given log_file

11.1.4 aequilibrae.matrix

class aequilibrae.matrix.**AequilibraeMatrix**

Matrix class

static random_name() → Path

Returns a random name for a matrix with root in the temp directory of the user

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> mat = AequilibraeMatrix()
>>> str(mat.random_name())
'/tmp/aequilibrae/Aequilibrae_matrix_...'
```

close()

Removes matrix from memory and flushes all data to disk, or closes the OMX file if that is the case

columns() → ndarray

Returns column vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (np.ndarray): the column totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
```

(continues on next page)

(continued from previous page)

```
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.columns()
array([357.54256811, 357.45109051, 310.88655449, 276.6783439 ,
       266.70388637, 270.62976319, 266.32888632, 279.6897402 ,
       285.89821842, 242.79743295, 252.34085912, 301.78116548,
       302.97058146, 270.61855294, 264.59944248, 257.83842251,
       276.63310578, 257.74513863, 281.15724257, 271.63886077,
       264.62215032, 252.79791125, 273.18139747, 282.7636574 ])

>>> project.close()
```

computational_view(*core_list: List[str] = None*)

Creates a memory view for a list of matrices that is compatible with Cython memory buffers

It allows for AequilibraE matrices to be used in all parallelized algorithms within AequilibraE

In case of OMX matrices, the computational view is held only in memory

Arguments

core_list (`list`): List with the names of all matrices that need to be in the buffer

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.computational_view(['bike trips', 'walk trips'])
>>> mat.view_names
['bike trips', 'walk trips']
```

copy(*output_name: str = None*, *cores: List[str] = None*, *names: List[str] = None*, *compress: bool = None*, *memory_only: bool = True*)

Copies a list of cores (or all cores) from one matrix file to another one

Arguments

output_name (`str`): Name of the new matrix file. If none is provided, returns a copy in memory only

cores (`list`): List of the matrix cores to be copied

names (`list`, *Optional*): List with the new names for the cores. Defaults to current names

compress (`bool`, *Optional*): Whether you want to compress the matrix or not. Defaults to `False`. Not yet implemented

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
```

(continues on next page)

(continued from previous page)

```
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.copy(Path(my_folder_path) / 'copy_of_my_matrix.aem',
...          cores=['bike trips', 'walk trips'],
...          names=['bicycle', 'walking'],
...          memory_only=False)
<aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix object at 0x...>

>>> mat2 = AequilibraeMatrix()
>>> mat2.load(Path(my_folder_path) / 'copy_of_my_matrix.aem')
>>> mat2.cores
2
```

create_empty(*file_name: Path = None*, *zones: int = None*, *matrix_names: List[str] = None*, *data_type: dtype = <class 'numpy.float64'>*, *index_names: List[str] = None*, *compressed: bool = False*, *memory_only: bool = True*)

Creates an empty matrix in the AequilibraE format

Arguments

file_name (`Path`): Local path to the matrix file

zones (`int`): Number of zones in the model (Integer). Maximum number of zones in a matrix is 4,294,967,296

matrix_names (`list`): A regular Python list of names of the matrix. Limit is 50 characters each. Maximum number of cores per matrix is 256

data_type (`np.dtype`, *Optional*): Data type of the matrix as NUMPY data types (`np.int32`, `np.int64`, `np.float32`, `np.float64`). Defaults to `np.float64`

index_names (`list`, *Optional*): A regular Python list of names for indices. Limit is 20 characters each. Maximum number of indices per matrix is 256

compressed (`bool`, *Optional*): Whether it is a flat matrix or a compressed one (Boolean - Not yet implemented)

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.num_indices
1
```

(continues on next page)

(continued from previous page)

```
>>> mat.zones
3317
```

create_from_omx(*omx_path: str*, *file_path: str = None*, *cores: List[str] = None*, *mappings: List[str] = None*, *robust: bool = True*, *compressed: bool = False*, *memory_only: bool = True*) → None

Creates an AequilibraeMatrix from an original OpenMatrix

Arguments

omx_path (`Path`): Path to the OMX file one wants to import

file_path (`Path`, *Optional*): Path for the output AequilibraeMatrix

cores (`list`, *Optional*): List of matrix cores to be imported

mappings (`list`, *Optional*): List of the matrix mappings (i.e. indices, centroid numbers) to be imported

robust (`bool`, *Optional*): Boolean for whether AequilibraE should try to adjust the names for cores and indices in case they are too long. Defaults to `True`

compressed (`bool`, *Optional*): Boolean for whether we should compress the output matrix. Not yet implemented

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

create_from_trip_list(*path_to_file: str*, *from_column: str*, *to_column: str*, *list_cores: List[str]*) → str

Creates an AequilibraeMatrix from a trip list csv file The output is saved in the same folder as the trip list file

Arguments

path_to_file (`str`): Path for the trip list csv file

from_column (`str`): trip list file column containing the origin zones numbers

to_column (`str`): trip list file column containing the destination zones numbers

list_cores (`list`): list of core columns in the trip list file

export(*output_name: Path*, *cores: List[str] = None*)

Exports the matrix to other formats, rather than AEM. Formats currently supported: CSV, OMX

When exporting to AEM or OMX, the user can chose to export only a set of cores, but all indices are exported

When exporting to CSV, the active index will be used, and all cores will be exported as separate columns in the output file

Arguments

output_name (`Path`): Path to the output file

cores (`list`): Names of the cores to be exported.

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
```

(continues on next page)

(continued from previous page)

```
>>> mat.export(Path(my_folder_path) / 'my_new_path.omx', ['Car trips', 'bike␣
↪trips'])
```

get_matrix(*core: str*, *copy=False*) → ndarray

Returns the data for a matrix core

Arguments

core (`str`): name of the matrix core to be returned

copy (`bool`, *Optional*): return a copy of the data. Defaults to False

Returns

object (`np.ndarray`): NumPy array

is_omx()

Returns `True` if matrix data source is OMX, `False` otherwise

load(*file_path: Path*)

Loads matrix from disk. All cores and indices are load. First index is default.

Arguments

file_path (`str`): Path to AEM or OMX file on disk

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view()
>>> mat.names
['distance_blended', 'time_final']

>>> project.close()
```

nan_to_num()

Converts all `NaN` values in all cores in the computational view to zeros

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> nan_matrix = np.empty((3,3))
>>> nan_matrix[:] = np.nan

>>> index = np.arange(1, 4, dtype=np.int32)

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / "matrices/nan_matrix.aem
↪",
...                  zones=3,
...                  matrix_names=["only_nan"])
>>> mat.index[:] = index[:]
>>> mat.matrix["only_nan"][:, :] = nan_matrix[:, :]
>>> mat.computational_view()
```

(continues on next page)

(continued from previous page)

```
>>> mat.nan_to_num()
>>> mat.get_matrix("only_nan")
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

rows() → ndarray

Returns row vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (np.ndarray): the row totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.rows()
array([357.68202084, 358.68778868, 310.68285491, 275.87964738,
       265.91709918, 268.60184371, 267.32264726, 281.3793747 ,
       286.15085073, 242.60308705, 252.1776242 , 305.56774194,
       303.58100777, 270.48841269, 263.20417379, 253.92665702,
       277.1655432 , 258.84368258, 280.65697316, 272.7651157 ,
       264.06806038, 252.87533845, 273.45639965, 281.61102767])

>>> project.close()
```

save(*names=()*, *file_name=None*) → None

Saves matrix data back to file.

If working with AEM file, it flushes data to disk. If working with OMX, requires new names.

Arguments

names (tuple(str), *Optional*): New names for the matrices. Required if working with OMX files

file_name (str, *Optional*): Local path to the matrix file

setDescription(*matrix_description: str*)

Sets description for the matrix

Arguments

matrix_description (str): Text with matrix description. Maximum length is 144 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.aem',
...                  zones=zones_in_the_model,
```

(continues on next page)

(continued from previous page)

```
...                     memory_only=False)
>>> mat.setDescription('This is a text')
>>> mat.save()
>>> mat.close()

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(my_folder_path) / 'my_matrix.aem')
>>> mat.description.decode('utf-8')
'This is a text'
```

setName(*matrix_name: str*)

Sets the name for the matrix itself. Only works for matrices in disk.

Arguments

matrix_name (`str`): matrix name. Maximum length is 50 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.omx',
...                  zones=zones_in_the_model,
...                  memory_only=False)
>>> mat.setName('This is my example')
>>> mat.save()
>>> mat.close()
```

set_index(*index_to_set: str*) → None

Sets the standard index to be the one the user wants to have be the one being used in all operations during run time. The first index is ALWAYS the default one every time the matrix is instantiated

Arguments

index_to_set (`str`): Name of the index to be used. The default index name is 'main_index'

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']
>>> index_list = ['tazs', 'census']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list,
...                  index_names=index_list )
>>> mat.num_indices
2
>>> mat.current_index
'tazs'
>>> mat.set_index('census')
>>> mat.current_index
'census'
```

class aequilibrae.matrix.**COO**

A class to implement sparse matrix operations such as reading, writing, and indexing

classmethod from_disk(*path*, *names=None*, *aeq=False*)

Read a OMX file and return a dictionary of matrix names to a scipy.sparse matrix, or aequilibrae.matrix.sparse matrix.

classmethod from_matrix(*m*)

Create COO matrix from an dense or scipy-like matrix.

to_disk(*path*, *name: str*)

to_scipy(*shape=None*)

Create scipy.sparse.coo_matrix from this COO matrix.

shape

class aequilibrae.matrix.**GeneralisedCOODemand**

add_df(*dfs: pd.DataFrame | List[pd.DataFrame]*, *shape=None*, *fill: float = 0.0*)

Add a DataFrame to the existing ones.

Expects a DataFrame with a multi-index of (o, d).

add_matrix(*matrix:* AequilibraeMatrix, *shape=None*, *fill: float = 0.0*)

Add an AequilibraE matrix to the existing demand in a sparse manner.

batches()

is_empty() → bool

no_demand() → bool

df

f32_names

f64_names

nodes_to_indices

shape

class aequilibrae.matrix.**Sparse**

A class to implement sparse matrix operations such as reading, writing, and indexing

classmethod from_disk(*path*, *names=None*, *aeq=False*)

Read a OMX file and return a dictionary of matrix names to a scipy.sparse matrix, or aequilibrae.matrix.sparse matrix.

to_disk(*path*, *name: str*)

Modules

aequilibrae_matrix
coo_demand
sparse_matrix

aequilibrae.matrix.aequilibrae_matrix

Classes

AequilibraeMatrix()	Matrix class

class aequilibrae.matrix.aequilibrae_matrix.**AequilibraeMatrix**

Matrix class

static random_name() → Path

Returns a random name for a matrix with root in the temp directory of the user

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> mat = AequilibraeMatrix()
>>> str(mat.random_name())
'/tmp/aequilibrae/Aequilibrae_matrix_...'
```

close()

Removes matrix from memory and flushes all data to disk, or closes the OMX file if that is the case

columns() → ndarray

Returns column vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (np.ndarray): the column totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.columns()
array([357.54256811, 357.45109051, 310.88655449, 276.6783439 ,
       266.70388637, 270.62976319, 266.32888632, 279.6897402 ,
       285.89821842, 242.79743295, 252.34085912, 301.78116548,
       302.97058146, 270.61855294, 264.59944248, 257.83842251,
       276.63310578, 257.74513863, 281.15724257, 271.63886077,
       264.62215032, 252.79791125, 273.18139747, 282.7636574 ])

>>> project.close()
```

computational_view(*core_list: List[str] = None*)

Creates a memory view for a list of matrices that is compatible with Cython memory buffers

It allows for AequilibraE matrices to be used in all parallelized algorithms within AequilibraE

In case of OMX matrices, the computational view is held only in memory

Arguments

core_list (list): List with the names of all matrices that need to be in the buffer

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.computational_view(['bike trips', 'walk trips'])
>>> mat.view_names
['bike trips', 'walk trips']
```

copy(*output_name: str = None*, *cores: List[str] = None*, *names: List[str] = None*, *compress: bool = None*, *memory_only: bool = True*)

Copies a list of cores (or all cores) from one matrix file to another one

Arguments

output_name (`str`): Name of the new matrix file. If none is provided, returns a copy in memory only

cores (`list`): List of the matrix cores to be copied

names (`list`, *Optional*): List with the new names for the cores. Defaults to current names

compress (`bool`, *Optional*): Whether you want to compress the matrix or not. Defaults to `False`. Not yet implemented

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.copy(Path(my_folder_path) / 'copy_of_my_matrix.aem',
...          cores=['bike trips', 'walk trips'],
...          names=['bicycle', 'walking'],
...          memory_only=False)
<aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix object at 0x...>

>>> mat2 = AequilibraeMatrix()
>>> mat2.load(Path(my_folder_path) / 'copy_of_my_matrix.aem')
>>> mat2.cores
2
```

create_empty(*file_name: Path = None*, *zones: int = None*, *matrix_names: List[str] = None*, *data_type: dtype = <class 'numpy.float64'>*, *index_names: List[str] = None*, *compressed: bool = False*, *memory_only: bool = True*)

Creates an empty matrix in the AequilibraE format

Arguments

file_name (`Path`): Local path to the matrix file

zones (`int`): Number of zones in the model (Integer). Maximum number of zones in a matrix is 4,294,967,296

matrix_names (`list`): A regular Python list of names of the matrix. Limit is 50 characters each. Maximum number of cores per matrix is 256

data_type (`np.dtype`, *Optional*): Data type of the matrix as NUMPY data types (`np.int32`, `np.int64`, `np.float32`, `np.float64`). Defaults to `np.float64`

index_names (`list`, *Optional*): A regular Python list of names for indices. Limit is 20 characters each. Maximum number of indices per matrix is 256

compressed (`bool`, *Optional*): Whether it is a flat matrix or a compressed one (Boolean - Not yet implemented)

memory_only (`bool`, *Optional*): Whether you want to keep the matrix copy in memory only. Defaults to `True`

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.num_indices
1
>>> mat.zones
3317
```

create_from_omx(*omx_path: str*, *file_path: str = None*, *cores: List[str] = None*, *mappings: List[str] = None*, *robust: bool = True*, *compressed: bool = False*, *memory_only: bool = True*) → None

Creates an AequilibraeMatrix from an original OpenMatrix

Arguments

omx_path (`Path`): Path to the OMX file one wants to import

file_path (`Path`, *Optional*): Path for the output AequilibraeMatrix

cores (`list`, *Optional*): List of matrix cores to be imported

mappings (`list`, *Optional*): List of the matrix mappings (i.e. indices, centroid numbers) to be imported

robust (`bool`, *Optional*): Boolean for whether AequilibraE should try to adjust the names for cores and indices in case they are too long. Defaults to `True`

compressed (`bool`, *Optional*): Boolean for whether we should compress the output matrix. Not yet implemented

memory_only (**`bool`, *Optional*): Whether you want to keep the matrix copy in** memory only. Defaults to `True`

create_from_trip_list(*path_to_file: str*, *from_column: str*, *to_column: str*, *list_cores: List[str]*) → str

Creates an AequilibraeMatrix from a trip list csv file The output is saved in the same folder as the trip list file

Arguments

path_to_file (`str`): Path for the trip list csv file

from_column (`str`): trip list file column containing the origin zones numbers

to_column (`str`): trip list file column containing the destination zones numbers

list_cores (`list`): list of core columns in the trip list file

export(*output_name: Path*, *cores: List[str] = None*)

Exports the matrix to other formats, rather than AEM. Formats currently supported: CSV, OMX

When exporting to AEM or OMX, the user can chose to export only a set of cores, but all indices are exported

When exporting to CSV, the active index will be used, and all cores will be exported as separate columns in the output file

Arguments

output_name (`Path`): Path to the output file

cores (`list`): Names of the cores to be exported.

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.export(Path(my_folder_path) / 'my_new_path.omx', ['Car trips', 'bike␣
↪trips'])
```

get_matrix(*core: str*, *copy=False*) → ndarray

Returns the data for a matrix core

Arguments

core (`str`): name of the matrix core to be returned

copy (`bool`, *Optional*): return a copy of the data. Defaults to False

Returns

object (`np.ndarray`): NumPy array

is_omx()

Returns `True` if matrix data source is OMX, `False` otherwise

load(*file_path: Path*)

Loads matrix from disk. All cores and indices are load. First index is default.

Arguments

file_path (`str`): Path to AEM or OMX file on disk

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
```

(continues on next page)

(continued from previous page)

```
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view()
>>> mat.names
['distance_blended', 'time_final']

>>> project.close()
```

nan_to_num()

Converts all NaN values in all cores in the computational view to zeros

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> nan_matrix = np.empty((3,3))
>>> nan_matrix[:] = np.nan

>>> index = np.arange(1, 4, dtype=np.int32)

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / "matrices/nan_matrix.aem
↪",
...                  zones=3,
...                  matrix_names=["only_nan"])
>>> mat.index[:] = index[:]
>>> mat.matrix["only_nan"][:, :] = nan_matrix[:, :]
>>> mat.computational_view()
>>> mat.nan_to_num()
>>> mat.get_matrix("only_nan")
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

rows() → ndarray

Returns row vector for the matrix in the computational view

Computational view needs to be set to a single matrix core

Returns

object (np.ndarray): the row totals for the matrix currently on the computational view

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> project = create_example(project_path)

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(project_path) / 'matrices/skims.omx')
>>> mat.computational_view(["distance_blended"])
>>> mat.rows()
array([357.68202084, 358.68778868, 310.68285491, 275.87964738,
       265.91709918, 268.60184371, 267.32264726, 281.3793747 ,
       286.15085073, 242.60308705, 252.1776242 , 305.56774194,
       303.58100777, 270.48841269, 263.20417379, 253.92665702,
       277.1655432 , 258.84368258, 280.65697316, 272.7651157 ,
       264.06806038, 252.87533845, 273.45639965, 281.61102767])
```

(continues on next page)

(continued from previous page)

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

save (*names=()*, *file_name=None*) → None

Saves matrix data back to file.

If working with AEM file, it flushes data to disk. If working with OMX, requires new names.

Arguments

names (`tuple(str)`, *Optional*): New names for the matrices. Required if working with OMX files

file_name (`str`, *Optional*): Local path to the matrix file

setDescription (*matrix_description: str*)

Sets description for the matrix

Arguments

matrix_description (`str`): Text with matrix description. Maximum length is 144 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.aem',
...                  zones=zones_in_the_model,
...                  memory_only=False)
>>> mat.setDescription('This is a text')
>>> mat.save()
>>> mat.close()

>>> mat = AequilibraeMatrix()
>>> mat.load(Path(my_folder_path) / 'my_matrix.aem')
>>> mat.description.decode('utf-8')
'This is a text'
```

setName (*matrix_name: str*)

Sets the name for the matrix itself. Only works for matrices in disk.

Arguments

matrix_name (`str`): matrix name. Maximum length is 50 characters

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=Path(my_folder_path) / 'my_matrix.omx',
...                  zones=zones_in_the_model,
...                  memory_only=False)
>>> mat.setName('This is my example')
>>> mat.save()
>>> mat.close()
```

set_index(*index_to_set: str*) → None

Sets the standard index to be the one the user wants to have be the one being used in all operations during run time. The first index is ALWAYS the default one every time the matrix is instantiated

Arguments

index_to_set (`str`): Name of the index to be used. The default index name is 'main_index'

```
>>> from aequilibrae.matrix import AequilibraeMatrix

>>> zones_in_the_model = 3317
>>> names_list = ['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk␣
↪trips']
>>> index_list = ['tazs', 'census']

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(zones=zones_in_the_model,
...                  matrix_names=names_list,
...                  index_names=index_list )
>>> mat.num_indices
2
>>> mat.current_index
'tazs'
>>> mat.set_index('census')
>>> mat.current_index
'census'
```

aequilibrae.matrix.coo_demand

Classes

GeneralisedCOODemand

class aequilibrae.matrix.coo_demand.**GeneralisedCOODemand**

add_df(*dfs: pd.DataFrame | List[pd.DataFrame]*, *shape=None*, *fill: float = 0.0*)

Add a DataFrame to the existing ones.

Expects a DataFrame with a multi-index of (o, d).

add_matrix(*matrix:* AequilibraeMatrix, *shape=None*, *fill: float = 0.0*)

Add an AequilibraE matrix to the existing demand in a sparse manner.

batches()

is_empty() → bool

no_demand() → bool

df

f32_names

f64_names

nodes_to_indices

shape

aequilibrae.matrix.sparse_matrix

Classes

COO	A class to implement sparse matrix operations such as reading, writing, and indexing
Sparse	A class to implement sparse matrix operations such as reading, writing, and indexing

class aequilibrae.matrix.sparse_matrix.**COO**

A class to implement sparse matrix operations such as reading, writing, and indexing

classmethod from_disk(*path*, *names=None*, *aeq=False*)

Read a OMX file and return a dictionary of matrix names to a scipy.sparse matrix, or aequilibrae.matrix.sparse matrix.

classmethod from_matrix(*m*)

Create COO matrix from an dense or scipy-like matrix.

to_disk(*path*, *name: str*)

to_scipy(*shape=None*)

Create scipy.sparse.coo_matrix from this COO matrix.

shape

class aequilibrae.matrix.sparse_matrix.**Sparse**

A class to implement sparse matrix operations such as reading, writing, and indexing

classmethod from_disk(*path*, *names=None*, *aeq=False*)

Read a OMX file and return a dictionary of matrix names to a scipy.sparse matrix, or aequilibrae.matrix.sparse matrix.

to_disk(*path*, *name: str*)

11.1.5 aequilibrae.parameters

Classes

Parameters([path])	Global parameters module.

class aequilibrae.parameters.**Parameters**(*path: Path | None = None*)

Global parameters module.

Parameters are used in many procedures, and are often defined in the `parameters.yml` file ONLY.

Parameters are organized 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.parameters import Parameters

>>> project = Project()
>>> project.new(project_path)

>>> p = 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']['max_query_area_size'] = 10000000000
>>> p.parameters['osm']['sleeptime'] = 0
>>> p.write_back()

>>> # You can also restore the software default values
>>> p.restore_default()

>>> project.close()
```

classmethod load_default()

restore_default()

Restores parameters to generic default

write_back()

Writes the parameters back to file

file_default: Path = PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/parameters.yml')

11.1.6 aequilibrae.paths

class aequilibrae.paths.**AssignmentPaths**(*table_name: str*, *project=None*)

Class for accessing path files optionally generated during assignment.

static get_path_for_destination_from_files(*path_o: DataFrame*, *path_o_index: DataFrame*, *destination: int*)

for a given path file and path index file, and a given destination, return the path links in o-d order

get_path_for_destination(*origin: int*, *destination: int*, *iteration: int*, *traffic_class_id: str*)

Return all link ids, i.e. the full path, for a given destination

read_path_file(*origin: int*, *iteration: int*, *traffic_class_id: str) -> (<class 'pandas.DataFrame'>*, *<class 'pandas.DataFrame'>*)

class aequilibrae.paths.**AssignmentResults**

Assignment result holder for a single *TrafficClass* with multiple user classes

get_graph_to_network_mapping()

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (pd.DataFrame): Pandas DataFrame data with the traffic class assignment results

get_sl_results() → DataFrame

prepare(*graph:* Graph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (*Graph*): Needs to have been set with number of centroids and list of skims (if any)

matrix (AequilibraeMatrix): Matrix properly set for computation with matrix.computational_view(:obj:`list`)

reset() → None

Resets object to prepared and pre-computation state

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

total_flows() → None

Totals all link flows for this class into a single link load

Results are placed into *total_link_loads* class member

class aequilibrae.paths.**ConnectivityAnalysis**(*graph*, *origins=None*, *project=None*)

```
>>> from aequilibrae.paths.connectivity_analysis import ConnectivityAnalysis

>>> project = create_example(project_path)

>>> network = project.network
>>> network.build_graphs()

>>> graph = network.graphs['c']
>>> graph.set_graph(cost_field="distance")
>>> graph.set_blocked_centroid_flows(False)

>>> conn_test = ConnectivityAnalysis(graph)
>>> conn_test.execute()

# The connectivity tester report as a Pandas DataFrame
>>> disconnected = conn_test.disconnected_pairs

>>> 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

connectivity = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.paths.**Graph**(**args*, ***kwargs*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (`str`): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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`): heuristic to use if `a_star` is enabled. Default is `None`.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()
```

(continues on next page)

(continued from previous page)

```
>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk(*filename: str*) → None

Loads graph from disk

Arguments

filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments

filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments

block_centroid_flows (`bool`): Blocking or not paths to go 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

class aequilibrae.paths.**HyperpathGenerating**(*edges*, *tail='tail'*, *head='head'*, *trav_time='trav_time'*, *freq='freq'*, *check_edges=False*, *skim_cols=None*, ** (Keyword-only parameters separator (PEP 3102))*, *o_vert_ids=array([], dtype=int64)*, *d_vert_ids=array([], dtype=int64)*, *nodes_to_indices*)

A class for hyperpath generation.

Arguments

edges (`pandas.DataFrame`): The edges of the graph.

tail (`str`, optional): The column name for the tail of the edge. Default is "tail".

head (`str`, optional): The column name for the head of the edge. Default is "head".

trav_time (`str`, optional): The column name for the travel time of the edge. Default is "trav_time".

freq (`str`, optional): The column name for the frequency of the edge. Default is "freq".

check_edges (`bool`, optional): If `True`, check the validity of the edges. Default is `False`.

assign(*origin_column*, *destination_column*, *demand_column*, *check_demand=False*, *threads=None*)

Assigns demand to the edges of the graph.

Assumes the **_column* arguments are provided as numpy arrays that form a COO sprase matrix.

Arguments

origin_column (`np.ndarray`, optional): The column for the origin vertices. Default is "orig_vert_idx".

destination_column (`np.ndarray`, optional): The column or the destination vertices. Default is "dest_vert_idx".

demand_column (`np.ndarray`, optional): The column for the demand values. Default is "demand".

check_demand (`bool`, optional): If `True`, check the validity of the demand data. Default is `False`.

threads (`int`, optional): The number of threads to use for computation. Default is `0` (using all available threads).

check_skim_cols(*skim_cols: Union(list[str], tuple[str], set(str))*)

compute_skim_cols(*skim_cols*, *edges: DataFrame*, *trav_time: str*)

info() → dict

run(*origin*, *destination*, *volume*)

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 active project

class aequilibrae.paths.**MultiThreadedAoN**

prepare(*graph*, *results*)

class aequilibrae.paths.**MultiThreadedNetworkSkimming**

prepare(*graph*, *cores*, *nodes*, *num_skims*)

prepare_(*graph*, *cores*, *nodes*)

class aequilibrae.paths.**NetworkSkimming**(*graph*, *origins=None*, *project=None*)

```
>>> from aequilibrae.paths.network_skimming import NetworkSkimming

>>> project = create_example(project_path)
>>> project.network.build_graphs(modes=["c"])

>>> graph = project.network.graphs['c']
>>> graph.set_graph("distance")
>>> graph.set_skimming("distance")

>>> skm = NetworkSkimming(graph)
>>> skm.execute()

# The skim report (if any error generated) is available here
>>> skm.report
[]

# To access the skim matrix directly from its temporary file
>>> matrix = skm.results.skims

# Or you can save the results to disk
>>> skm.save_to_project('skimming_result_omx', 'omx')

>>> project.close()
```

doWork()

execute()

Runs the skimming process as specified in the graph

save_to_project(*name: str*, *format='omx'*, *project=None*) → None

Saves skim results to the project folder and creates record in the database

Arguments

name (`str`): Name of the matrix. Same value for matrix record name and file (plus extension)

format (`str`, *Optional*): File format ('aem' or 'omx'). Default is 'omx'

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active project

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.paths.**OptimalStrategies**(*assig_spec*)

execute()

class aequilibrae.paths.**PathResults**

Path computation result holder

```
>>> from aequilibrae.paths.results import PathResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize distance
>>> car_graph.set_graph('distance')

# If you want to compute skims
# It does increase path computation time substantially
>>> car_graph.set_skimming(['distance', 'free_flow_time'])

>>> res = PathResults()
>>> res.prepare(car_graph)
>>> res.compute_path(1, 17)

# Update all the outputs mentioned above for destination 9. Same origin: 1
>>> res.update_trace(9)

# clears all computation results
```

(continues on next page)

(continued from previous page)

```
>>> res.reset()

>>> project.close()
```

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*) → None

Computes the path between two nodes in the network.

*A** heuristics are currently only valid distance cost fields.

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`): Heuristic to use if `a_star` is enabled. Default is `None`.

get_heuristics() → List[str]

Return the available heuristics.

prepare(*graph:* Graph) → None

Prepares the object with dimensions corresponding to the graph object

Arguments

graph (*Graph*): Needs to have been set with number of centroids and list of skims (if any)

reset() → None

Resets object to prepared and pre-computation state

set_heuristic(*heuristic: str*) → None

Set the heuristics to be used in A*. Must be one of *get_heuristics()*.

Arguments

heuristic (`str`): Heuristic to use in A*.

update_trace(*destination: int*) → None

Updates the path's nodes, links, skims and mileposts

If the previously computed path had *early_exit* enabled, *update_trace* will check if the *destination* has already been found, if not the shortest path tree will be recomputed with the *early_exit* argument passed on.

If the previously computed path had *a_star* enabled, *update_trace* always recompute the path.

Arguments

destination (`int`): ID of the node we are computing the path too

class aequilibrae.paths.**RouteChoice**(*graph:* Graph, *project=None*)

add_demand(*demand*, *fill: float = 0.0*)

Add demand DataFrame or matrix for the assignment.

Arguments

demand (`Union[pd.DataFrame, AequilibraeMatrix]`): Demand to add to assignment. If the supplied demand is a DataFrame, it should have a 2-level MultiIndex of Origin and

Destination node IDs. If an AequilibraE Matrix is supplied node IDs will be inferred from the index. Demand values should be either `float32` or `float64`.

fill (`float`): Value to fill any `NaN` with.

`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 a list of links IDs stored as integers with the direction encoded as the sign. 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 graphs current cost field and the `beta` and `cutoff_prob` parameters.

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.

If `recompute_psl` is `False` the table returned from `self.get_results()` will have all zeros for the cost and path overlap fields, and all True for the mask field. If `recompute_psl` is `True` these fields will be recalculated as required.

`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.

`execute_single`(*origin: int*, *destination: int*, *demand: float = 0.0*) → List[Tuple[int]]

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.

Returns

route set (`List[Tuple[int]]`): A list of routes as tuples of link IDs.

`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`. When the link id in the route set is positive it represents the ab direction, while negative represents 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, Dict[str, coo_matrix]]

Get the select link OD matrix results as a sparse matrix.

Returns

select link OD matrix results (`Dict[str, Dict[str, scipy.sparse.coo_matrix]]`): Returns a dict of select link set names to a dict of demand column names to a sparse OD matrix

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.

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 active 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_flows(*table_name: str*, *project=None*) → None

Saves the select link link flows for all classes into the results database. Additionally, it exports the OD matrices into OMX format.

Arguments

table_name (`str`): Name of the table being inserted to and the name of the OpenMatrix file used for OD matrices.

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active project

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.
- When performing an assignment, `cutoff_prob` can be provided to exclude routes from the path-sized logit model. The `cutoff_prob` is used to compute an inverse binary logit and obtain a max difference in utilities. If a paths total cost is greater than the minimum cost path in the route set plus the max difference, the route is excluded from the PSL calculations. The route is still returned, but with a probability of 0.0.
- The `cutoff_prob` should be in the range $[0, 1]$. It is then rescaled internally to $[0.5, 1]$ as probabilities below 0.5 produce negative differences in utilities because the choice is between two routes only, one of which is the shortest path. A higher `cutoff_prob` includes less routes. A value of 1.0 will only include the minimum cost route. A value of 0.0 includes all routes.

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_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]]]*, *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, dirirection)`.

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': {'beta': 1.0,
'cutoff_prob': 0.0, 'max_depth': 0, 'max_misses': 100, 'max_routes': 0,
'penalty': 1.01, 'seed': 0, 'store_results': True}, 'link-penalisation': {}}

demand_index_names = ['origin id', 'destination id']
```

class aequilibrae.paths.**SkimResults**

Network skimming result holder.

```
>>> from aequilibrae.paths.results import SkimResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize travel time
>>> car_graph.set_graph('free_flow_time')

# Skims travel time and distance
>>> car_graph.set_skimming(['free_flow_time', 'distance'])

>>> res = SkimResults()
>>> res.prepare(car_graph)

>>> res.skims.export(project_path / "skim_matrices.omx")

>>> project.close()
```

prepare(*graph:* Graph)

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)

class aequilibrae.paths.**SubAreaAnalysis**(*graph:* Graph, *subarea: GeoDataFrame*, *demand: DataFrame* | AequilibraeMatrix, *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.

class aequilibrae.paths.**TrafficAssignment**(*project=None*)

Traffic assignment class.

For a comprehensive example on use, see the Use examples page.

```
>>> from aequilibrae.paths import TrafficAssignment, TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(True)

>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")

# We will only assign one user class stored as 'matrix' inside the OMX file
>>> demand.computational_view(['matrix'])

# Creates the assignment class
>>> assigclass = TrafficClass("car", graph, demand)

>>> assig = TrafficAssignment()

# The first thing to do is to add at list of traffic classes to be assigned
>>> assig.set_classes([assigclass])

# Then we set the volume delay function
>>> assig.set_vdf("BPR")  # This is not case-sensitive

# And its parameters
>>> assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# The capacity and free flow travel times as they exist in the graph
>>> assig.set_capacity_field("capacity")
>>> assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
>>> assig.set_algorithm('bfw')

>>> assig.max_iter = 10
>>> assig.rgap_target = 0.00001

>>> assig.execute() # we then execute the assignment

# 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()
```

(continues on next page)

(continued from previous page)

```
# 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 = assigclass.results.skims # blended ones
>>> last_skims = assigclass._aon_results.skims # those for the last iteration

>>> project.close()
```

add_class(*traffic_class:* TrafficClass) → None

Adds a traffic class to the assignment

Arguments

traffic_class (*TrafficClass*): Traffic class

add_preload(*preload: DataFrame*, *name: str = None*) → None

Given a dataframe of 'link_id', 'direction' and 'preload', merge into current preloads dataframe.

Arguments

preload (`pd.DataFrame`): dataframe mapping 'link_id' & 'direction' to 'preload' **name** (`str`): Name for particular preload (optional - default name will be chosen if not specified)

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns

skim list (`list`): Lists of all skims with the results for each class

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

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 active 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 active 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 active 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*) → None

Saves the skims (if any) to the skim folder and registers in the matrix list

Arguments

name (str): Name of the matrix record to hold this matrix (same name used for file name)

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 ('aem' or 'omx'). Default is 'omx'

project (Project, *Optional*): Project we want to save the results to.
Defaults to the active project

select_link_flows() → Dict[str, DataFrame]

Returns a dataframe of the select link flows for each class

set_algorithm(*algorithm: str*)

Chooses the assignment algorithm. e.g. 'frank-wolfe', 'bfw', 'msa'

'fw' is also accepted as an alternative to 'frank-wolfe'

Arguments
algorithm (`str`): Algorithm to be used

set_capacity_field(*capacity_field: str*) → None

Sets the graph field that contains link capacity for the assignment period -> e.g. 'capacity1h'

Arguments
capacity_field (`str`): Field name

set_classes(*classes: List[TrafficClass]*) → None

Sets Traffic classes to be assigned

Arguments
classes (`List[TrafficClass]`): List of Traffic classes for assignment

set_cores(*cores: int*) → None

Allows one to set the number of cores to be used AFTER traffic classes have been added

Inherited from `AssignmentResultsBase`

Arguments
cores (`int`): Number of CPU cores to use

set_path_file_format(*file_format: str*) → None

Specify path saving format. Either parquet or feather.

Arguments
file_format (`str`): Name of file format to use for path files

set_save_path_files(*save_it: bool*) → None

Turn path saving on or off.

Arguments
save_it (`bool`): Boolean to indicate whether paths should be saved

set_time_field(*time_field: str*) → None

Sets the graph field that contains free flow travel time -> e.g. 'fftime'

Arguments
time_field (`str`): Field name

set_vdf(*vdf_function: str*) → None

Sets the Volume-delay function to be used

Arguments
vdf_function (`str`): Name of the VDF to be used

set_vdf_parameters(*par: dict*) → None

Sets the parameters for the Volume-delay function.

Parameter values can be scalars (same values for the entire network) or network field names (link-specific values) - Examples: {'alpha': 0.15, 'beta': 4.0} or {'alpha': 'alpha', 'beta': 'beta'}

The Akcelik VDF parameter 'tau' value has typical `8` factor absorbed into it. Users should supply `8 * tau` to match other common usages. Additionally the standard `0.25` factor can be overridden by supplying the 'alpha' parameter.

Arguments

par (`dict`): Dictionary with all parameters for the chosen VDF

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.

algorithm: str

all_algorithms = ['all-or-nothing', 'msa', 'frank-wolfe', 'fw', 'cfw', 'bfw']

assignment: *LinearApproximation* | *OptimalStrategies*

bpr_parameters = ['alpha', 'beta']

capacity: ndarray

capacity_field: str

classes: List[*TrafficClass*]

congested_time: ndarray

cores: int

description: str

free_flow_tt: ndarray

preloads: DataFrame

save_path_files: bool

time_field: str

total_flow: ndarray

vdf_parameters: list

class aequilibrae.paths.**TrafficClass**(*name: str*, *graph:* Graph, *matrix:* AequilibraeMatrix)

Traffic class for equilibrium traffic assignment

```
>>> from aequilibrae.paths import TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(False)
```

(continues on next page)

(continued from previous page)

```
>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")
>>> demand.computational_view()

>>> tc = TrafficClass("car", graph, demand)
>>> tc.set_pce(1.3)

>>> project.close()
```

set_fixed_cost(*field_name: str*, *multiplier=1*)

Sets value of time

Arguments

field_name (`str`): Name of the graph field with fixed costs for this class

multiplier (`Union[float, int]`): Multiplier for the fixed cost. Defaults to 1 if not set

set_pce(*pce: float | int*) → None

Sets Passenger Car equivalent

Arguments

pce (`Union[float, int]`): PCE. Defaults to 1 if not set

set_select_links(*links: Dict[str, List[Tuple[int, int]]]*)

Set the selected links. Checks if the links and directions are valid. Translates link_id and direction into unique link id used in compact graph. Supply links=None to disable select link analysis.

Arguments

links (`Union[None, Dict[str, List[Tuple[int, int]]]]`): name of link set and Link IDs and directions to be used in select link analysis

set_vot(*value_of_time: float*) → None

Sets value of time

Arguments

value_of_time (`Union[float, int]`): Value of time. Defaults to 1 if not set

skim_congested(*skim_fields=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.

Arguments

skim_fields (`Union[None, str]`): Name of the skims to use. If None, uses default only

property info: dict

class aequilibrae.paths.**TransitAssignment**(**args*, *project=None*, ***kwargs*)

add_class(*transport_class:* TransportClassBase) → None

Adds a Transport class to the assignment

Arguments

transport_class (`TransportClassBase`): Transport class

algorithms_available() → list

Returns all algorithms available for use

Returns
`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns
skim list (`list`): Lists of all skims with the results for each class

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 active 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[TransportClassBase]*) → None

Sets Transport classes to be assigned

Arguments
classes (`List[TransportClassBase]`): List of TransportClass's for assignment

set_cores(*cores: int*) → 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

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']

assignment: *LinearApproximation* | *OptimalStrategies*

classes: List[*TrafficClass*]

cores: int

description: str

free_flow_tt: ndarray

time_field: str

total_flow: ndarray

class aequilibrae.paths.**TransitAssignmentResults**

Assignment result holder for a single `Transit`

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (pd.DataFrame): DataFrame data with the transit class assignment results

prepare(*graph:* TransitGraph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (*TransitGraph*): Needs to have been set with number of centroids

matrix (AequilibraeMatrix): Matrix properly set for computation with matrix.computational_view(:obj:`list`)

reset() → None

Resets object to prepared and pre-computation state

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

class aequilibrae.paths.**TransitClass**(*name: str*, *graph:* TransitGraph, *matrix:* AequilibraeMatrix)

set_demand_matrix_core(*core: str*)

Set the matrix core to use for demand.

Arguments

core (str):

property info: dict

class aequilibrae.paths.**TransitGraph**(*config: dict = None*, *od_node_mapping: DataFrame = None*, **args*, ***kwargs*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (str): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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`): heuristic to use if `a_star` is enabled. Default is `None`.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()

>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk (*filename: str*) → None

Loads graph from disk

Arguments

filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments

filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments

block_centroid_flows (`bool`): Blocking or not paths to go 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

property config

class aequilibrae.paths.**VDF**

Volume-Delay function

```
>>> from aequilibrae.paths import VDF

>>> vdf = VDF()
>>> vdf.functions_available()
['bpr', 'bpr2', 'conical', 'inrets', 'akcelik']
```

functions_available() → list

returns a list of all functions available

class aequilibrae.paths.**allOrNothing**(*class_name: str*, *matrix:* AequilibraeMatrix, *graph:* Graph, *results:* AssignmentResults)

doWork()

execute()

func_assig_thread(*origin*, *all_threads*)

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.paths.**one_to_all**(*origin*, *matrix*, *graph*, *result*, *aux_result*, *curr_thread*)

aequilibrae.paths.**path_computation**(*origin*, *destination*, *graph*, *results*)

Parameters

- **graph** – AequilibraE graph. Needs to have been set with number of centroids and list of skims (if any)
- **results** – AequilibraE Matrix properly set for computation using matrix.computational_view([matrix list])
- **skimming** – if we will skim for all nodes or not

aequilibrae.paths.**skimming_single_origin**(*origin*, *graph*, *result*, *aux_result*, *curr_thread*)

Parameters

- **origin**
- **graph**
- **results**

Returns

aequilibrae.paths.**update_path_trace**(*results*, *destination*, *graph*)

If *results.early_exit* is *True*, early exit will be enabled if the path is to be recomputed. If *results.a_star* is *True*, A* will be used if the path is to be recomputed.

Parameters

- **graph** – AequilibraE graph. Needs to have been set with number of centroids and list of skims (if any)
- **results** – AequilibraE Matrix properly set for computation using matrix.computational_view([matrix list])
- **destination** – New destination for path computation

Modules

AoN
all_or_nothing
assignment_paths
connectivity_analysis
graph
graph_building
linear_approximation

continues on next page

Table 16 – continued from previous page

multi_threaded_aon	
multi_threaded_paths	
multi_threaded_skimming	
network_skimming	
optimal_strategies	
public_transport	
results	path computation related code :1: (WARNING/2) Title overline too short. ============ path computation related code ============
route_choice	
sub_area	
traffic_assignment	
traffic_class	
vdf	

aequilibrae.paths.AoN

aequilibrae.paths.all_or_nothing

Classes

allOrNothing(class_name, matrix, graph, results)	

class aequilibrae.paths.all_or_nothing.**allOrNothing**(*class_name: str*, *matrix:* AequilibraeMatrix, *graph:* Graph, *results:* AssignmentResults)

doWork()

execute()

func_assig_thread(*origin*, *all_threads*)

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.paths.assignment_paths

Classes

AssignmentPaths(table_name[, project])	Class for accessing path files optionally generated during assignment.
AssignmentResultsTable(table_name[, project])	
TrafficClassIdentifier(name, id)	

class aequilibrae.paths.assignment_paths.**AssignmentPaths**(*table_name: str*, *project=None*)

Class for accessing path files optionally generated during assignment.

static get_path_for_destination_from_files(*path_o: DataFrame*, *path_o_index: DataFrame*, *destination: int*)

for a given path file and path index file, and a given destination, return the path links in o-d order

get_path_for_destination(*origin: int*, *destination: int*, *iteration: int*, *traffic_class_id: str*)

Return all link ids, i.e. the full path, for a given destination

read_path_file(*origin: int*, *iteration: int*, *traffic_class_id: str) -> (<class 'pandas.DataFrame'>, <class 'pandas.DataFrame'>*)

class aequilibrae.paths.assignment_paths.**AssignmentResultsTable**(*table_name: str*, *project=None*)

get_traffic_class_names_and_id() → List[*TrafficClassIdentifier*]

class aequilibrae.paths.assignment_paths.**TrafficClassIdentifier**(*name: str*, *id: str*)

aequilibrae.paths.connectivity_analysis

Classes

ConnectivityAnalysis(graph[, origins, project])	

class aequilibrae.paths.connectivity_analysis.**ConnectivityAnalysis**(*graph*, *origins=None*, *project=None*)

```
>>> from aequilibrae.paths.connectivity_analysis import ConnectivityAnalysis

>>> project = create_example(project_path)

>>> network = project.network
>>> network.build_graphs()

>>> graph = network.graphs['c']
>>> graph.set_graph(cost_field="distance")
>>> graph.set_blocked_centroid_flows(False)

>>> conn_test = ConnectivityAnalysis(graph)
>>> conn_test.execute()

# The connectivity tester report as a Pandas DataFrame
>>> disconnected = conn_test.disconnected_pairs

>>> 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

connectivity = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.paths.graph

Classes

Graph(*args, **kwargs)	
GraphBase([logger])	Graph class.
NetworkGraphIndices(network_ab_idx, ...)	
TransitGraph([config, od_node_mapping])	

class aequilibrae.paths.graph.**Graph**(**args*, ***kwargs*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (`str`): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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`): heuristic to use if `a_star` is enabled. Default is `None`.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()
```

(continues on next page)

(continued from previous page)

```
>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk(*filename: str*) → None

Loads graph from disk

Arguments

filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments

filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments

block_centroid_flows (`bool`): Blocking or not paths to go 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

class aequilibrae.paths.graph.**GraphBase**(*logger=None*)

Graph class.

AequilibraE graphs implement two forms of compression.

- link contraction, and
- dead end removal.

Link contraction creates a topological equivalent graph by contracting sequences of links between nodes with degrees of two. This compresses long streams of links, such as along highways or curved roads, into single links.

Dead end removal attempts to remove dead ends and fish spines from the network. It does this based on the observation that in a graph with non-negative weights a dead end will only ever appear in the results of a short(est) path if the origin or destination is present within that dead end.

Dead end removal is applied before link contraction and does not create a strictly topological equivalent graph, however, all centroids are preserved.

The compressed graph is used internally.

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (`str`): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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`): heuristic to use if `a_star` is enabled. Default is `None`.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()

>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk(*filename: str*) → None

Loads graph from disk

Arguments

filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments

filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments

block_centroid_flows (`bool`): Blocking or not paths to go 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

class aequilibrae.paths.graph.**NetworkGraphIndices**(*network_ab_idx: <built-in function array>*, *network_ba_idx: <built-in function array>*, *graph_ab_idx: <built-in function array>*, *graph_ba_idx: <built-in function array>*)

graph_ab_idx: array

graph_ba_idx: array

network_ab_idx: array

network_ba_idx: array

class aequilibrae.paths.graph.**TransitGraph**(*config: dict = None*, *od_node_mapping: DataFrame = None*, **args*, ***kwargs*)

available_skims() → List[str]

Returns graph fields that are available to be set as skims.

Returns

list (`str`): Skimmeable field names

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*)

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`): heuristic to use if `a_star` is enabled. Default is `None`.

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()

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.

```
>>> project = create_example(project_path)

>>> project.network.build_graphs()

>>> graph = project.network.graphs['c']
>>> graph.prepare_graph(np.arange(1,25))

>>> idx, data, node_mapping = graph.create_compressed_link_network_mapping()

>>> 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'

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

load_from_disk(*filename: str*) → None

Loads graph from disk

Arguments
filename (`str`): Path to file

prepare_graph(*centroids: ndarray | None = None*, *remove_dead_ends: bool = True*) → 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`.

reverse()

save_compressed_correspondence(*path*, *mode_name*, *mode_id*)

Save graph and nodes_to_indices to disk

save_to_disk(*filename: str*) → None

Saves graph to disk

Arguments
filename (`str`): Path to file. Usual file extension is `aeg`.

set_blocked_centroid_flows(*block_centroid_flows*) → None

Chooses whether we want to block paths to go through centroids or not. Default value is `True`.

Arguments
block_centroid_flows (`bool`): Blocking or not paths to go 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

property config

aequilibrae.paths.graph_building

aequilibrae.paths.linear_approximation

Classes

LinearApproximation(assig_spec, algorithm[, ...])

class aequilibrae.paths.linear_approximation.**LinearApproximation**(*assig_spec*, *algorithm*, *project=None*)

calculate_biconjugate_direction()

calculate_conjugate_stepsize()

calculate_stepsize()

Calculate optimal stepsize in descent direction

check_convergence()

Calculate relative gap and return `True` if it is smaller than desired precision.

Two relative gaps are computed and stored on the instance:

- `self.rgap` - the AequilibraE convention, `|Σ flow·cost − Σ AON·cost| / Σ flow·cost`. **This is the only quantity used for the stopping criterion** (compared against `self.rgap_target`). `(Σ flow·cost − Σ direction·cost) / Σ flow·cost`, where `direction` is the BFW combined step direction

doWork()

execute()

assignment = <aequilibrae.utils.python_signal.PythonSignal object>

equilibration = <aequilibrae.utils.python_signal.PythonSignal object>

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.paths.multi_threaded_aon

Classes

MultiThreadedAoN()

class aequilibrae.paths.multi_threaded_aon.**MultiThreadedAoN**

prepare(*graph*, *results*)

aequilibrae.paths.multi_threaded_paths

Classes

MultiThreadedPaths()

class aequilibrae.paths.multi_threaded_paths.**MultiThreadedPaths**

prepare_(*graph*, *cores*, *nodes*)

aequilibrae.paths.multi_threaded_skimming

Classes

MultiThreadedNetworkSkimming()

class aequilibrae.paths.multi_threaded_skimming.**MultiThreadedNetworkSkimming**

prepare(*graph*, *cores*, *nodes*, *num_skims*)

prepare_(*graph*, *cores*, *nodes*)

aequilibrae.paths.network_skimming

Classes

NetworkSkimming(graph[, origins, project])

class aequilibrae.paths.network_skimming.**NetworkSkimming**(*graph*, *origins=None*, *project=None*)

```
>>> from aequilibrae.paths.network_skimming import NetworkSkimming

>>> project = create_example(project_path)
>>> project.network.build_graphs(modes=["c"])

>>> graph = project.network.graphs['c']
>>> graph.set_graph("distance")
>>> graph.set_skimming("distance")

>>> skm = NetworkSkimming(graph)
>>> skm.execute()

# The skim report (if any error generated) is available here
>>> skm.report
[]

# To access the skim matrix directly from its temporary file
>>> matrix = skm.results.skims

# Or you can save the results to disk
>>> skm.save_to_project('skimming_result_omx', 'omx')

>>> project.close()
```

doWork()

execute()

Runs the skimming process as specified in the graph

save_to_project(*name: str*, *format='omx'*, *project=None*) → None

Saves skim results to the project folder and creates record in the database

Arguments

name (`str`): Name of the matrix. Same value for matrix record name and file (plus extension)

format (`str`, *Optional*): File format ('aem' or 'omx'). Default is 'omx'

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active project

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>

aequilibrae.paths.optimal_strategies

Classes

OptimalStrategies(assig_spec)

class aequilibrae.paths.optimal_strategies.**OptimalStrategies**(*assig_spec*)

execute()

aequilibrae.paths.public_transport

aequilibrae.paths.results

path computation related code

STILL NEED TO ADD SOME EXPLANATIONS HERE

class aequilibrae.paths.results.**AssignmentResults**

Assignment result holder for a single `TrafficClass` with multiple user classes

get_graph_to_network_mapping()

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (`pd.DataFrame`): Pandas DataFrame data with the traffic class assignment results

get_sl_results() → DataFrame

prepare(*graph:* Graph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (`Graph`): Needs to have been set with number of centroids and list of skims (if any)

matrix (`AequilibraeMatrix`): Matrix properly set for computation with `matrix.computational_view(:obj:`list`)`

reset() → None

Resets object to prepared and pre-computation state

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

total_flows() → None

Totals all link flows for this class into a single link load

Results are placed into *total_link_loads* class member

class aequilibrae.paths.results.**PathResults**

Path computation result holder

```
>>> from aequilibrae.paths.results import PathResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize distance
>>> car_graph.set_graph('distance')

# If you want to compute skims
# It does increase path computation time substantially
>>> car_graph.set_skimming(['distance', 'free_flow_time'])

>>> res = PathResults()
>>> res.prepare(car_graph)
>>> res.compute_path(1, 17)

# Update all the outputs mentioned above for destination 9. Same origin: 1
>>> res.update_trace(9)

# clears all computation results
>>> res.reset()

>>> project.close()
```

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*) → None

Computes the path between two nodes in the network.

*A** heuristics are currently only valid distance cost fields.

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`): Heuristic to use if `a_star` is enabled. Default is `None`.

get_heuristics() → List[str]

Return the available heuristics.

prepare(*graph:* Graph) → None

Prepares the object with dimensions corresponding to the graph object

Arguments

graph (`Graph`): Needs to have been set with number of centroids and list of skims (if any)

reset() → None

Resets object to prepared and pre-computation state

set_heuristic(*heuristic: str*) → None

Set the heuristics to be used in A*. Must be one of *get_heuristics()*.

Arguments

heuristic (`str`): Heuristic to use in A*.

update_trace(*destination: int*) → None

Updates the path's nodes, links, skims and mileposts

If the previously computed path had *early_exit* enabled, *update_trace* will check if the *destination* has already been found, if not the shortest path tree will be recomputed with the *early_exit* argument passed on.

If the previously computed path had *a_star* enabled, *update_trace* always recompute the path.

Arguments

destination (`int`): ID of the node we are computing the path too

class aequilibrae.paths.results.**SkimResults**

Network skimming result holder.

```
>>> from aequilibrae.paths.results import SkimResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize travel time
```

(continues on next page)

(continued from previous page)

```
>>> car_graph.set_graph('free_flow_time')

# Skims travel time and distance
>>> car_graph.set_skimming(['free_flow_time', 'distance'])

>>> res = SkimResults()
>>> res.prepare(car_graph)

>>> res.skims.export(project_path / "skim_matrices.omx")

>>> project.close()
```

prepare(*graph:* Graph)

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)

class aequilibrae.paths.results.**TransitAssignmentResults**

Assignment result holder for a single Transit

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (pd.DataFrame): DataFrame data with the transit class assignment results

prepare(*graph:* TransitGraph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (TransitGraph): Needs to have been set with number of centroids

matrix (AequilibraeMatrix): Matrix properly set for computation with matrix.computational_view(:obj:`list`)

reset() → None

Resets object to prepared and pre-computation state

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

Modules

assignment_results
path_results

continues on next page

Table 27 – continued from previous page

skim_results

aequilibrae.paths.results.assignment_results

Classes

AssignmentResults()	Assignment result holder for a single `TrafficClass` with multiple user classes
AssignmentResultsBase()	Assignment results base class for traffic and transit assignments.
TransitAssignmentResults()	Assignment result holder for a single `Transit`

class aequilibrae.paths.results.assignment_results.**AssignmentResults**

Assignment result holder for a single `TrafficClass` with multiple user classes

get_graph_to_network_mapping()

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (`pd.DataFrame`): Pandas DataFrame data with the traffic class assignment results

get_sl_results() → DataFrame

prepare(*graph:* Graph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (`Graph`): Needs to have been set with number of centroids and list of skims (if any)

matrix (`AequilibraeMatrix`): Matrix properly set for computation with `matrix.computational_view(:obj:`list`)`

reset() → None

Resets object to prepared and pre-computation state

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

total_flows() → None

Totals all link flows for this class into a single link load

Results are placed into *total_link_loads* class member

class aequilibrae.paths.results.assignment_results.**AssignmentResultsBase**

Assignment results base class for traffic and transit assignments.

abstractmethod prepare(*graph:* GraphBase, *matrix:* AequilibraeMatrix) → None

abstractmethod reset() → None

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

class aequilibrae.paths.results.assignment_results.**TransitAssignmentResults**

Assignment result holder for a single `Transit`

get_load_results() → DataFrame

Translates the assignment results from the graph format into the network format

Returns

dataset (`pd.DataFrame`): DataFrame data with the transit class assignment results

prepare(*graph:* TransitGraph, *matrix:* AequilibraeMatrix) → None

Prepares the object with dimensions corresponding to the assignment matrix and graph objects

Arguments

graph (`TransitGraph`): Needs to have been set with number of centroids

matrix (`AequilibraeMatrix`): Matrix properly set for computation with `matrix.computational_view(:obj:`list`)`

reset() → None

Resets object to prepared and pre-computation state

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

aequilibrae.paths.results.path_results

Classes

PathResults()	Path computation result holder

class aequilibrae.paths.results.path_results.**PathResults**

Path computation result holder

```
>>> from aequilibrae.paths.results import PathResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize distance
>>> car_graph.set_graph('distance')

# If you want to compute skims
# It does increase path computation time substantially
>>> car_graph.set_skimming(['distance', 'free_flow_time'])

>>> res = PathResults()
>>> res.prepare(car_graph)
>>> res.compute_path(1, 17)

# Update all the outputs mentioned above for destination 9. Same origin: 1
>>> res.update_trace(9)

# clears all computation results
>>> res.reset()

>>> project.close()
```

compute_path(*origin: int*, *destination: int*, *early_exit: bool = False*, *a_star: bool = False*, *heuristic: str | None = None*) → None

Computes the path between two nodes in the network.

*A** heuristics are currently only valid distance cost fields.

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`): Heuristic to use if `a_star` is enabled. Default is `None`.

get_heuristics() → List[str]

Return the available heuristics.

prepare(*graph:* Graph) → None

Prepares the object with dimensions corresponding to the graph object

Arguments

graph (`Graph`): Needs to have been set with number of centroids and list of skims (if any)

reset() → None

Resets object to prepared and pre-computation state

set_heuristic(*heuristic: str*) → None

Set the heuristics to be used in A*. Must be one of *get_heuristics()*.

Arguments

heuristic (`str`): Heuristic to use in A*.

update_trace(*destination: int*) → None

Updates the path's nodes, links, skims and mileposts

If the previously computed path had *early_exit* enabled, *update_trace* will check if the *destination* has already been found, if not the shortest path tree will be recomputed with the *early_exit* argument passed on.

If the previously computed path had *a_star* enabled, *update_trace* always recompute the path.

Arguments

destination (`int`): ID of the node we are computing the path too

aequilibrae.paths.results.skim_results

Classes

SkimResults()	Network skimming result holder.

class aequilibrae.paths.results.skim_results.**SkimResults**

Network skimming result holder.

```
>>> from aequilibrae.paths.results import SkimResults

>>> project = create_example(project_path)
>>> project.network.build_graphs()

# Mode c is car in this project
>>> car_graph = project.network.graphs['c']

# minimize travel time
>>> car_graph.set_graph('free_flow_time')

# Skims travel time and distance
>>> car_graph.set_skimming(['free_flow_time', 'distance'])

>>> res = SkimResults()
>>> res.prepare(car_graph)

>>> res.skims.export(project_path / "skim_matrices.omx")

>>> project.close()
```

prepare(*graph:* Graph)

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)

aequilibrae.paths.route_choice

Classes

RouteChoice(graph[, project])	

class aequilibrae.paths.route_choice.**RouteChoice**(*graph:* Graph, *project=None*)

add_demand(*demand*, *fill: float = 0.0*)

Add demand DataFrame or matrix for the assignment.

Arguments

demand (`Union[pd.DataFrame, AequilibraeMatrix]`): Demand to add to assignment. If the supplied demand is a DataFrame, it should have a 2-level MultiIndex of Origin and Destination node IDs. If an AequilibraE Matrix is supplied node IDs will be inferred from the index. Demand values should be either `float32` or `float64`.

fill (`float`): Value to fill any `NaN` with.

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 a list of links IDs stored as integers with the direction encoded as the sign. 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 graphs current cost field and the `beta` and `cutoff_prob` parameters.

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.

If `recompute_psl` is `False` the table returned from `self.get_results()` will have all zeros for the cost and path overlap fields, and all True for the mask field. If `recompute_psl` is `True` these fields will be recalculated as required.

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.

execute_single(*origin: int*, *destination: int*, *demand: float = 0.0*) → List[Tuple[int]]

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.

Returns

route set (`List[Tuple[int]]`): A list of routes as tuples of link IDs.

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`. When the link id in the route set is positive it represents the ab direction, while negative represents 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, Dict[str, coo_matrix]]

Get the select link OD matrix results as a sparse matrix.

Returns

select link OD matrix results (`Dict[str, Dict[str, scipy.sparse.coo_matrix]]`): Returns a dict of select link set names to a dict of demand column names to a sparse OD matrix

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.

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 active 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_flows(*table_name: str*, *project=None*) → None

Saves the select link link flows for all classes into the results database. Additionally, it exports the OD matrices into OMX format.

Arguments

table_name (`str`): Name of the table being inserted to and the name of the OpenMatrix file used for OD matrices.

project (`Project`, *Optional*): Project we want to save the results to. Defaults to the active project

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.
- When performing an assignment, `cutoff_prob` can be provided to exclude routes from the path-sized logit model. The `cutoff_prob` is used to compute an inverse binary logit and obtain a max difference in utilities. If a paths total cost is greater than the minimum cost path in the route set plus the max difference, the route is excluded from the PSL calculations. The route is still returned, but with a probability of 0.0.
- The `cutoff_prob` should be in the range $[0, 1]$. It is then rescaled internally to $[0.5, 1]$ as probabilities below 0.5 produce negative differences in utilities because the choice is between two routes only, one of which is the shortest path. A higher `cutoff_prob` includes less routes. A value of 1.0 will only include the minimum cost route. A value of 0.0 includes all routes.

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_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, dirirection)`.

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']

cores: int

default_parameters = {'bfsle': {'penalty': 1.0}, 'generic': {'beta': 1.0,
'cutoff_prob': 0.0, 'max_depth': 0, 'max_misses': 100, 'max_routes': 0,
'penalty': 1.01, 'seed': 0, 'store_results': True}, 'link-penalisation': {}}

demand_index_names = ['origin id', 'destination id']

sl_compact_link_loads: Dict[str, array] | None

sl_link_loads: Dict[str, array] | None

where: Path | None
```

aequilibrae.paths.sub_area

Classes

SubAreaAnalysis(graph, subarea, demand[, ...])	

class aequilibrae.paths.sub_area.**SubAreaAnalysis**(*graph: Graph*, *subarea: GeoDataFrame*, *demand: DataFrame | AequilibraeMatrix*, *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.

aequilibrae.paths.traffic_assignment

Classes

AssignmentBase([project])	
TrafficAssignment([project])	Traffic assignment class.
TransitAssignment(*args[, project])	

class aequilibrae.paths.traffic_assignment.**AssignmentBase**(*project=None*)

add_class(*transport_class:* TransportClassBase) → None

Adds a Transport class to the assignment

Arguments

transport_class (`TransportClassBase`): Transport class

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns

skim list (`list`): Lists of all skims with the results for each class

abstractmethod info() → dict

abstractmethod log_specification()

report() → DataFrame

Returns the assignment convergence report

Returns

DataFrame (`pd.DataFrame`): Convergence report

abstractmethod results() → DataFrame

abstractmethod save_results(*table_name: str*, *keep_zero_flows=True*, *project=None*) → None

abstractmethod set_algorithm(*algorithm: str*)

set_classes(*classes: List[*TransportClassBase*]*) → None

Sets Transport classes to be assigned

Arguments

classes (`List[TransportClassBase]`): List of TransportClass's for assignment

abstractmethod set_cores(*cores: int*) → None

set_time_field(*time_field: str*) → None

algorithm: str

assignment: ***LinearApproximation*** **|** ***OptimalStrategies***

classes: List[*TrafficClass*]

cores: int

description: str

free_flow_tt: ndarray

time_field: str

total_flow: ndarray

class aequilibrae.paths.traffic_assignment.**TrafficAssignment**(*project=None*)

Traffic assignment class.

For a comprehensive example on use, see the Use examples page.

```
>>> from aequilibrae.paths import TrafficAssignment, TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(True)

>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")

# We will only assign one user class stored as 'matrix' inside the OMX file
>>> demand.computational_view(['matrix'])

# Creates the assignment class
>>> assigclass = TrafficClass("car", graph, demand)

>>> assig = TrafficAssignment()

# The first thing to do is to add at list of traffic classes to be assigned
>>> assig.set_classes([assigclass])

# Then we set the volume delay function
>>> assig.set_vdf("BPR")  # This is not case-sensitive

# And its parameters
>>> assig.set_vdf_parameters({"alpha": "b", "beta": "power"})

# The capacity and free flow travel times as they exist in the graph
>>> assig.set_capacity_field("capacity")
>>> assig.set_time_field("free_flow_time")

# And the algorithm we want to use to assign
>>> assig.set_algorithm('bfw')

>>> assig.max_iter = 10
```

(continues on next page)

(continued from previous page)

```
>>> assig.rgap_target = 0.00001

>>> assig.execute() # we then execute the assignment

# 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 = assigclass.results.skims # blended ones
>>> last_skims = assigclass._aon_results.skims # those for the last iteration

>>> project.close()
```

add_class(*traffic_class:* TrafficClass) → None

Adds a traffic class to the assignment

Arguments

traffic_class (`TrafficClass`): Traffic class

add_preload(*preload: DataFrame*, *name: str = None*) → None

Given a dataframe of 'link_id', 'direction' and 'preload', merge into current preloads dataframe.

Arguments

preload (`pd.DataFrame`): dataframe mapping 'link_id' & 'direction' to 'preload' **name** (`str`): Name for particular preload (optional - default name will be chosen if not specified)

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns

skim list (`list`): Lists of all skims with the results for each class

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

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 active 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 active 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 active 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*) → None

Saves the skims (if any) to the skim folder and registers in the matrix list

Arguments
name (`str`): Name of the matrix record to hold this matrix (same name used for file name)

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 ('aem' or 'omx'). Default is 'omx'

project (`Project`, *Optional*): **Project we want to save the results to.**
Defaults to the active project

select_link_flows() → Dict[str, DataFrame]

Returns a dataframe of the select link flows for each class

set_algorithm(*algorithm: str*)

Chooses the assignment algorithm. e.g. 'frank-wolfe', 'bfw', 'msa'

'fw' is also accepted as an alternative to 'frank-wolfe'

Arguments
algorithm (`str`): Algorithm to be used

set_capacity_field(*capacity_field: str*) → None

Sets the graph field that contains link capacity for the assignment period -> e.g. 'capacity1h'

Arguments
capacity_field (`str`): Field name

set_classes(*classes: List[*TrafficClass*]*) → None

Sets Traffic classes to be assigned

Arguments
classes (`List[TrafficClass]`): List of Traffic classes for assignment

set_cores(*cores: int*) → None

Allows one to set the number of cores to be used AFTER traffic classes have been added

Inherited from `AssignmentResultsBase`

Arguments
cores (`int`): Number of CPU cores to use

set_path_file_format(*file_format: str*) → None

Specify path saving format. Either parquet or feather.

Arguments
file_format (`str`): Name of file format to use for path files

set_save_path_files(*save_it: bool*) → None

Turn path saving on or off.

Arguments
save_it (`bool`): Boolean to indicate whether paths should be saved

set_time_field(*time_field: str*) → None

Sets the graph field that contains free flow travel time -> e.g. 'fftime'

Arguments
time_field (`str`): Field name

set_vdf(*vdf_function: str*) → None

Sets the Volume-delay function to be used

Arguments
vdf_function (`str`): Name of the VDF to be used

set_vdf_parameters(*par: dict*) → None

Sets the parameters for the Volume-delay function.

Parameter values can be scalars (same values for the entire network) or network field names (link-specific values) - Examples: {'alpha': 0.15, 'beta': 4.0} or {'alpha': 'alpha', 'beta': 'beta'}

The Akcelik VDF parameter 'tau' value has typical `8` factor absorbed into it. Users should supply `8 * tau` to match other common usages. Additionally the standard `0.25` factor can be overridden by supplying the 'alpha' parameter.

Arguments

par (`dict`): Dictionary with all parameters for the chosen VDF

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.

algorithm: str

all_algorithms = ['all-or-nothing', 'msa', 'frank-wolfe', 'fw', 'cfw', 'bfw']

assignment: *LinearApproximation* | *OptimalStrategies*

bpr_parameters = ['alpha', 'beta']

capacity: ndarray

capacity_field: str

classes: List[*TrafficClass*]

congested_time: ndarray

cores: int

description: str

free_flow_tt: ndarray

preloads: DataFrame

save_path_files: bool

time_field: str

total_flow: ndarray

vdf_parameters: list

class aequilibrae.paths.traffic_assignment.**TransitAssignment**(**args*, *project=None*, ***kwargs*)

add_class(*transport_class:* TransportClassBase) → None

Adds a Transport class to the assignment

Arguments

transport_class (`TransportClassBase`): Transport class

algorithms_available() → list

Returns all algorithms available for use

Returns

`list`: List of string values to be used with **set_algorithm**

execute(*log_specification=True*) → None

Processes assignment

get_skim_results() → list

Prepares the assignment skim results for all classes

Returns

skim list (`list`): Lists of all skims with the results for each class

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 active 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[TransportClassBase]*) → None

Sets Transport classes to be assigned

Arguments
classes (`List[TransportClassBase]`): List of TransportClass's for assignment

set_cores(*cores: int*) → 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

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']

assignment: *LinearApproximation* | *OptimalStrategies*

classes: List[*TrafficClass*]

cores: int

description: str

free_flow_tt: ndarray

time_field: str

total_flow: ndarray

aequilibrae.paths.traffic_class

Classes

TrafficClass(name, graph, matrix)	Traffic class for equilibrium traffic assignment
TransitClass(name, graph, matrix)	
TransportClassBase(name, graph, matrix)	

class aequilibrae.paths.traffic_class.**TrafficClass**(*name: str*, *graph:* Graph, *matrix:* AequilibraeMatrix)

Traffic class for equilibrium traffic assignment

```
>>> from aequilibrae.paths import TrafficClass

>>> project = create_example(project_path)
>>> project.network.build_graphs()

>>> graph = project.network.graphs['c'] # we grab the graph for cars
>>> graph.set_graph('free_flow_time') # let's say we want to minimize time
>>> graph.set_skimming(['free_flow_time', 'distance']) # And will skim time and␣
↪distance
>>> graph.set_blocked_centroid_flows(False)

>>> proj_matrices = project.matrices

>>> demand = proj_matrices.get_matrix("demand_omx")
>>> demand.computational_view()

>>> tc = TrafficClass("car", graph, demand)
>>> tc.set_pce(1.3)

>>> project.close()
```

set_fixed_cost(*field_name: str*, *multiplier=1*)

Sets value of time

Arguments

field_name (`str`): Name of the graph field with fixed costs for this class

multiplier (`Union[float, int]`): Multiplier for the fixed cost. Defaults to 1 if not set

set_pce(*pce: float | int*) → None

Sets Passenger Car equivalent

Arguments

pce (`Union[float, int]`): PCE. Defaults to 1 if not set

set_select_links(*links: Dict[str, List[Tuple[int, int]]]*)

Set the selected links. Checks if the links and directions are valid. Translates link_id and direction into unique link id used in compact graph. Supply links=None to disable select link analysis.

Arguments

links (`Union[None, Dict[str, List[Tuple[int, int]]]]`): name of link set and Link IDs and directions to be used in select link analysis

set_vot(*value_of_time: float*) → None

Sets value of time

Arguments

value_of_time (`Union[float, int]`): Value of time. Defaults to 1 if not set

skim_congested(*skim_fields=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.

Arguments

skim_fields (`Union[None, str]`): Name of the skims to use. If None, uses default only

property info: dict

class aequilibrae.paths.traffic_class.**TransitClass**(*name: str*, *graph:* TransitGraph, *matrix:* AequilibraeMatrix)

set_demand_matrix_core(*core: str*)

Set the matrix core to use for demand.

Arguments

core (`str`):

property info: dict

class aequilibrae.paths.traffic_class.**TransportClassBase**(*name: str*, *graph:* GraphBase, *matrix:* AequilibraeMatrix)

property info: dict

aequilibrae.paths.vdf

Classes

VDF()	Volume-Delay function

class aequilibrae.paths.vdf.**VDF**

Volume-Delay function

```
>>> from aequilibrae.paths import VDF

>>> vdf = VDF()
>>> vdf.functions_available()
['bpr', 'bpr2', 'conical', 'inrets', 'akcelik']
```

functions_available() → list

returns a list of all functions available

11.1.7 aequilibrae.project

class aequilibrae.project.**About**(*project*)

Provides an interface for querying and editing the **about** table of an AequilibraE project

```
>>> project = create_example(project_path)

# Adding a new field and saving it
>>> project.about.add_info_field('my_super_relevant_field')
>>> project.about.my_super_relevant_field = 'super relevant information'
>>> project.about.write_back()

# changing the value for an existing value/field
>>> project.about.scenario_name = 'Just a better scenario name'
>>> project.about.write_back()

>>> project.close()
```

add_info_field(*info_field: str*) → None

Adds new information field to the model

Arguments

info_field (str): Name of the desired information field to be added. Has to be a valid Python VARIABLE name (i.e. letter as first character, no spaces and no special characters)

```
>>> project = create_example(project_path)

>>> project.about.add_info_field('a_cool_field')
>>> project.about.a_cool_field = 'super relevant information'
>>> project.about.write_back()

>>> project.close()
```

create()

Creates the 'about' table for project files that did not previously contain it

list_fields() → list

Returns a list of all characteristics the about table holds

write_back()

Saves the information parameters back to the project database

```
>>> project = create_example(project_path)

>>> project.about.description = 'This is the example project. Do not use for␣
↪forecast'
>>> project.about.write_back()

>>> project.close()
```

class aequilibrae.project.**FieldEditor**(*project*, *table_name: str*)

Allows user to edit the project data tables

The field editor is used for two different purposes:

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

This is a general class used to manage all project's data tables accessible to the user and but it should be accessed directly from within the module corresponding to the data table one wants to edit. Example:

```
>>> project = create_example(project_path)

# To edit the fields of the link_types table
>>> lt_fields = project.network.link_types.fields

# To edit the fields of the modes table
>>> m_fields = project.network.modes.fields

>>> project.close()
```

Field descriptions are kept in the table *attributes_documentation*

add(*field_name: str*, *description: str*, *data_type='NUMERIC'*) → None

Adds new field to the data table

Arguments

field_name (str): Field to be added to the table. Must be a valid SQLite field name

description (str): Description of the field to be inserted in the metadata

data_type (str, *Optional*): Valid SQLite Data type. Default: "NUMERIC"

all_fields() → List[str]

Returns the list of fields available in the database

remove(*field_name: str*) → None

save() → None

Saves any field descriptions which my have been changed to the database and update layer statistics.

This is required for new fields to appear in applications like QGIS.

class aequilibrae.project.**Log**(*project_base_path: Path*)

API entry point to the log file contents

```
>>> project = Project()
>>> project.new(project_path)

>>> log = project.log()

# We get all entries for the log file
>>> entries = log.contents()

# Or clear everything (NO UN-DOs)
>>> log.clear()

>>> project.close()
```

clear()

Clears the log file. Use it wisely

contents() → list

Returns contents of log file

Returns

log_contents (list): List with all entries in the log file

class aequilibrae.project.**Matrices**(*project*)

Gateway into the matrices available/recorded in the model

check_exists(*name: str*) → bool

Checks whether a matrix with a given name exists

Returns
exists (bool): Does the matrix exist?

clear_database() → None

Removes records from the matrices database that do not exist in disk

delete_record(*matrix_name: str*) → None

Deletes a Matrix Record from the model and attempts to remove from disk

get_matrix(*matrix_name: str*) → *AequilibraeMatrix*

Returns an AequilibraE matrix available in the project

Raises an error if matrix does not exist

Arguments
matrix_name (str): Name of the matrix to be loaded

Returns
matrix (AequilibraeMatrix): Matrix object

get_record(*matrix_name: str*) → *MatrixRecord*

Returns a model Matrix Record for manipulation in memory

list() → DataFrame

List of all matrices available

Returns
df (pd.DataFrame): Pandas DataFrame listing all matrices available in the model

new_record(*name: str*, *file_name: str*, *matrix=None*) → *MatrixRecord*

Creates a new record for a matrix in disk, but does not save it

If the matrix file is not already on disk, it will fail

Arguments
name (str): Name of the matrix

file_name (str): Name of the file on disk

Returns
matrix_record (MatrixRecord): A matrix record that can be manipulated in memory before saving

reload()

Discards all memory matrices in memory and loads recreate them

update_database() → None

Adds records to the matrices database for matrix files found on disk

class aequilibrae.project.**Network**(*project*)

Network class. Member of an AequilibraE Project

build_graphs(*fields: list = None*, *modes: list = None*, *limit_to_area: Polygon = None*) → None

Builds graphs for all modes currently available in the model

When called, it overwrites all graphs previously created and stored in the networks' dictionary of graphs

Arguments

fields (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to specify which fields to use

modes (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to generate only those we need

limit_to_area (`Polygon`, *Optional*): When working with a very large model area, you may want to filter your database to a small area for your computation, which you can do by providing a polygon. The search is limited to a spatial index search, so it is very fast but NOT PRECISE.

To use the 'fields' parameter, a minimalistic option is the following

```
>>> project = create_example(project_path)

>>> fields = ['distance']
>>> project.network.build_graphs(fields, modes = ['c', 'w'])

>>> project.close()
```

convex_hull() → Polygon

Queries the model for the convex hull of the entire network

Returns

model coverage (`Polygon`): Shapely (Multi)polygon of the model network.

count_centroids() → int

Returns the number of centroids in the model

Returns

`int`: Number of centroids

count_links() → int

Returns the number of links in the model

Returns

`int`: Number of links

count_nodes() → int

Returns the number of nodes in the model

Returns

`int`: Number of nodes

create_from_gmns(*link_file_path: str*, *node_file_path: str*, *use_group_path: str = None*, *geometry_path: str = None*, *srid: int = 4326*) → None

Creates AequilibraE model from links and nodes in GMNS format.

Arguments

link_file_path (`str`): Path to a links csv file in GMNS format

node_file_path (`str`): Path to a nodes csv file in GMNS format

use_group_path (`str`, *Optional*): Path to a csv table containing groupings of uses. This helps AequilibraE know when a GMNS use is actually a group of other GMNS uses

geometry_path (`str`, *Optional*): Path to a csv file containing geometry information for a line object, if not specified in the link table

srid (`int`, *Optional*): Spatial Reference ID in which the GMNS geometries were created

create_from_osm(*model_area: Polygon | None = None*, *place_name: str | None = None*, *modes=('car', 'transit', 'bicycle', 'walk')*, *clean=True*) → None

Downloads the network from OpenStreetMap (OSM)

Arguments

area (`Polygon`, *Optional*): Polygon for which the network will be downloaded. If not provided, a place name would be required

place_name (`str`, *Optional*): If not downloading with East-West-North-South boundingbox, this is required

modes (`tuple`, *Optional*): List of all modes to be downloaded. Defaults to the modes in the parameter file

clean (`bool`, *Optional*): Keeps only the links that intersects the model area polygon. Defaults to `True`. Does not apply to networks downloaded with a place name

```
>>> project = Project()
>>> project.new(project_path)

# Now we can import the network for any place we want
>>> project.network.create_from_osm(place_name="my_beautiful_hometown")

>>> project.close()
```

export_to_gmns(*path: str*)

Exports AequilibraE network to csv files in GMNS format.

Arguments

path (`str`): Output folder path.

extent()

Queries the extent of the network included in the model

Returns

model extent (`Polygon`): Shapely polygon with the bounding box of the model network.

list_modes()

Returns a list of all the modes in this model

Returns

`list`: List of all modes

set_time_field(*time_field: str*) → None

Set the time field for all graphs built in the model

Arguments

time_field (`str`): Network field with travel time information

skimmable_fields()

Returns a list of all fields that can be skimmed

Returns

`list`: List of all fields that can be skimmed

link_types: *LinkTypes* = None

protected_fields = ['ogc_fid', 'geometry']

req_link_flds = ['link_id', 'a_node', 'b_node', 'direction', 'distance', 'modes', 'link_type']

req_node_flds = ['node_id', 'is_centroid']

signal = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.project.**NetworkSimplifier**(*project=None*)

collapse_links_into_nodes(*links: List[int]*)

Collapses links into nodes, adjusting the network in the neighborhood.

Arguments

links (`List[int]`): List containing link IDs to be collapsed.

rebuild_network()

Rebuilds the network elements that would have to be rebuilt after massive network simplification

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

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.**Period**(*dataset*, *project*)

A Period object represents a single record in the *periods* table

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period1 = all_periods.get(1)

# We can find out which fields exist for the period
>>> which_fields_do_we_have = period1.data_fields()

>>> project.close()
```

data_fields() → list

Lists all data fields for the period, as available in the database

Returns

data fields (`list`): list of all fields available for editing

renumber(*new_id: int*)

Renumbers the period in the network

Logs a warning if another period already exists with this period_id

Arguments

new_id (`int`): New period_id

save()

Saves period to database

class aequilibrae.project.**Periods**(*net*)

Access to the API resources to manipulate the periods table in the network

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period = all_periods.get(1)

# We can save changes for all periods we have edited so far
>>> all_periods.save()

>>> project.close()
```

extent()

Queries the extent of the layer included in the model

Returns

model extent (Polygon): Shapely polygon with the bounding box of the layer.

get(*period_id: int*) → *Period*

Get a period from the network by its **period_id**

It raises an error if period_id does not exist

Arguments

period_id (int): Id of a period to retrieve

Returns

period (*Period*): Period object for requested period_id

new_period(*period_id: int*, *start: int*, *end: int*, *description: str = None*) → *Period*

Creates a new period with a given ID

Arguments

period_id (int): Id of the centroid to be created

start (int): Start time of the period to be created

end (int): End time of the period to be created

description (str): Optional human readable description of the time period e.g. '1pm - 5pm'

refresh()

Refreshes all the periods in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: DataFrame

Returns all periods data as a Pandas DataFrame

Returns

table (DataFrame): Pandas DataFrame with all the periods

property default_period: *Period*

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

sql = ''

Query sql for retrieving periods

class aequilibrae.project.**Project**

AequilibraE project class

Listing 3: Create Project

```
>>> new_project = Project()
>>> new_project.new(project_path)

# Safely closes the project
>>> new_project.close()
```

Listing 4: Open Project

```
>>> existing_project = Project()
>>> existing_project.open(project_path)

>>> existing_project.close()
```

classmethod from_path(*project_folder*)

activate()

check_file_indices() → None

Makes results_database.sqlite and the matrices folder compatible with project database

clone_scenario(*scenario_name: str*, *description: str = ''*)

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 = ''*)

Creates an empty scenario, without any links, nodes, and zones.

Arguments

scenario_name (str): scenario name

description (str): useful scenario description

deactivate()

list_scenarios()

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

new(*project_path: str*) → None

Creates a new project

Arguments

project_path (`str`): Full path to the project data folder. If folder exists, it will fail

open(*project_path: str*) → None

Loads project from disk

Arguments

project_path (`str`): Full path to the project data folder. If the project inside does not exist, it will fail.

upgrade(*ignore_project: bool = False*, *ignore_transit: bool = False*, *ignore_results: bool = False*)

Find and apply all applicable migrations.

All database upgrades are 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*)

Switch the active scenario.

Arguments

scenario_name (`str`): name of the scenario to be activated

property about: *About*

property db_connection

property db_connection_spatial

Deprecated alias for `db_connection`, which is now a spatial connection.

property logger: Logger

property matrices: *Matrices*

property network: *Network*

property parameters: dict

property path_to_file

property project_base_path

property project_parameters: *Parameters*

property results: *Results*

property results_connection

property run

Load and return the AequilibraE run module with the default arguments from `parameters.yml` partially applied.

Refer to `run/__init__.py` file within the project folder for documentation.

property transit: *Transit*

property transit_connection

property zoning

class aequilibrae.project.**Scenario**(*name: str*, *base_path: Path*, *path_to_file: Path*)

Represents a modelling scenario within an AequilibraE project.

Each scenario operates independently with its own database and file structure while sharing the overall project configuration.

Scenarios are typically managed through the Project class rather than instantiated directly by users.

The root scenario is special-cased and represents the original project configuration. All other scenarios are stored in subdirectories and reference their own database files.

about: *About*

base_path: Path

logger: Logger

matrices: *Matrices*

name: str

network: *Network*

path_to_file: Path

results: *Results*

transit: *Transit*

zoning: *Zoning*

class aequilibrae.project.**Zone**(*dataset: dict*, *zoning*)

Single zone object that can be queried and manipulated in memory

add_centroid(*point: Point*, *robust=True*) → None

Adds a centroid to the network file

Arguments

point (`Point`): Shapely Point corresponding to the desired centroid position. If None, uses the geometric center of the zone

robust (`Bool`, *Optional*): Moves the centroid location around to avoid node conflict. Defaults to `True`.

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *conn: Connection | None = None*, *limit_to_zone=True*) → None

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting the zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the zone.

The selection of the nodes that will be connected is done simply by searching for the node closest to the zone centroid, or the N closest nodes to the centroid.

If fewer candidates than required connectors are found, all candidates are connected.

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered. eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

conn (`sqlite3.Connection`, *Optional*): Connection to the database.

limit_to_zone (`bool`): Limits the search for nodes inside the zone. Defaults to `True`.

delete()

Removes the zone from the database

disconnect_mode(*mode_id: str*) → None

Removes centroid connectors for the desired mode from the network file

Arguments

mode_id (`str`): Mode ID we are trying to disconnect from this zone

save()

Saves/Updates the zone data to the database

class aequilibrae.project.**Zoning**(*network*)

Access to the API resources to manipulate the 'zones' table in the project

```
>>> project = create_example(project_path)

>>> zoning = project.zoning

>>> zone_downtown = zoning.get(1)
>>> zone_downtown.population = 637
>>> zone_downtown.employment = 10039
>>> zone_downtown.save()

# We can also add one more field to the table
>>> fields = zoning.fields
```

(continues on next page)

(continued from previous page)

```
>>> fields.add('parking_spots', 'Total licensed parking spots', 'INTEGER')

>>> project.close()
```

add_centroids(*robust=True*)

Adds automatic centroids to the network file. It adds centroids to all zones that do not have one Centroid is added to the geographic centroid of the zone.

Arguments

robust (`bool`, *Optional*): Moves the centroid location around to avoid node conflict. Defaults to `True`.

all_zones() → dict

Returns a dictionary with all Zone objects available in the model, using `zone_id` as key

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *limit_to_zone=True*, *bulk: bool = False*)

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting each zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the zone.

The selection of the nodes that will be connected is done simply by searching for the node closest to each zone centroid, or the N closest nodes to the centroid.

If fewer candidates than required connectors are found, all candidates are connected.

CENTROIDS THAT ARE CURRENTLY CONNECTED ARE SKIPPED ALTOGETHER

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered.
eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

limit_to_zone (`bool`): Limits the search for nodes inside the zone. Defaults to `True`.

bulk (`bool`, *Optional*): Whether to use the bulk connector method or not. This is method is
considerably faster for connecting a large amount of centroids but has a high runtime overhead.

coverage() → Polygon

Returns a single polygon for the entire zoning coverage

Returns

model coverage (`Polygon`): Shapely (Multi)polygon of the zoning system.

create_zoning_layer()

Creates the 'zones' table for project files that did not previously contain it

extent() → Polygon

Queries the extent of the layer included in the model

Returns

model extent (`Polygon`): Shapely polygon with the bounding box of the layer.

get(*zone_id: str*) → *Zone*

Get a zone from the model by its `zone_id`

get_closest_zone(*geometry: Point | LineString | MultiLineString*) → int

Returns the zone in which the given geometry is located.

If the geometry is not fully enclosed by any zone, the zone closest to the geometry is returned

Arguments

geometry (`Point` or `LineString`): A Shapely geometry object

Returns

zone_id (`int`): ID of the zone applicable to the point provided

new(*zone_id: int*) → *Zone*

Creates a new zone

Returns

zone (*Zone*): A new zone object populated only with `zone_id` (but not saved in the model yet)

refresh_geo_index()

save()

property data: GeoDataFrame

Returns all zones data as a Pandas DataFrame

Returns

table (`GeoDataFrame`): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

Modules

about
basic_table
data
data_loader
database_connection
field_editor
network
project
project_cleaning
project_creation
scenario
table_loader
tools
zone
zoning

aequilibrae.project.about

Classes

About(project)	Provides an interface for querying and editing the **about** table of an AequilibraE project

class aequilibrae.project.about.**About**(*project*)

Provides an interface for querying and editing the **about** table of an AequilibraE project

```
>>> project = create_example(project_path)

# Adding a new field and saving it
>>> project.about.add_info_field('my_super_relevant_field')
>>> project.about.my_super_relevant_field = 'super relevant information'
>>> project.about.write_back()

# changing the value for an existing value/field
>>> project.about.scenario_name = 'Just a better scenario name'
>>> project.about.write_back()

>>> project.close()
```

add_info_field(*info_field: str*) → None

Adds new information field to the model

Arguments

info_field (`str`): Name of the desired information field to be added. Has to be a valid Python VARIABLE name (i.e. letter as first character, no spaces and no special characters)

```
>>> project = create_example(project_path)

>>> project.about.add_info_field('a_cool_field')
>>> project.about.a_cool_field = 'super relevant information'
>>> project.about.write_back()

>>> project.close()
```

create()

Creates the 'about' table for project files that did not previously contain it

list_fields() → list

Returns a list of all characteristics the about table holds

write_back()

Saves the information parameters back to the project database

```
>>> project = create_example(project_path)

>>> project.about.description = 'This is the example project. Do not use for␣
↪forecast'
>>> project.about.write_back()

>>> project.close()
```

aequilibrae.project.basic_table

Classes

BasicTable(project)	Basic resources used by all subclasses

class aequilibrae.project.basic_table.**BasicTable**(*project*)

Basic resources used by all subclasses

extent() → Polygon

Queries the extent of the layer included in the model

Returns

model extent (Polygon): Shapely polygon with the bounding box of the layer.

property fields: ***FieldEditor***

Returns a FieldEditor class instance to edit the zones table fields and their metadata

aequilibrae.project.data

class aequilibrae.project.data.**Matrices**(*project*)

Gateway into the matrices available/recorded in the model

check_exists(*name: str*) → bool

Checks whether a matrix with a given name exists

Returns

exists (bool): Does the matrix exist?

clear_database() → None

Removes records from the matrices database that do not exist in disk

delete_record(*matrix_name: str*) → None

Deletes a Matrix Record from the model and attempts to remove from disk

get_matrix(*matrix_name: str*) → *AequilibraeMatrix*

Returns an AequilibraE matrix available in the project

Raises an error if matrix does not exist

Arguments

matrix_name (str): Name of the matrix to be loaded

Returns

matrix (AequilibraeMatrix): Matrix object

get_record(*matrix_name: str*) → *MatrixRecord*

Returns a model Matrix Record for manipulation in memory

list() → DataFrame

List of all matrices available

Returns

df (pd.DataFrame): Pandas DataFrame listing all matrices available in the model

new_record(*name: str*, *file_name: str*, *matrix=None*) → *MatrixRecord*

Creates a new record for a matrix in disk, but does not save it

If the matrix file is not already on disk, it will fail

Arguments
name (`str`): Name of the matrix

file_name (`str`): Name of the file on disk

Returns
matrix_record (`MatrixRecord`): A matrix record that can be manipulated in memory before saving

reload()

Discards all memory matrices in memory and loads recreate them

update_database() → None

Adds records to the matrices database for matrix files found on disk

class aequilibrae.project.data.**Results**(*project*, *project_conn: Connection | None = None*, *results_conn: Connection | None = None*)

Gateway into the results available/recorded in the model

check_exists(*table_name: str*) → bool

Checks whether a result with a given name exists.

Parameters
****table_name**** (`str`) – Name of the result to check

Returns
Does the result exist?

Return type
exists (`bool`)

clear_database() → None

Removes records from the results table that do not exist in the results database.

delete_record(*table_name: str*) → None

Deletes a ResultRecord from the model and attempts to remove it from the results database.

Parameters
****table_name**** (`str`) – Name of the result to delete

Raises
****ValueError**** – If the result doesn't exist

get_record(*table_name: str*) → *ResultRecord*

Returns a model ResultsRecord for manipulation in memory.

Parameters
****table_name**** (`str`) – Name of the result record to retrieve

Returns
The requested result record

Return type
record (`ResultRecord`)

Raises
****ValueError**** – If the result doesn't exist or was deleted

get_results(*table_name: str*) → DataFrame

Returns a DataFrame containing the results.

Raises an error if results do not exist.

Parameters
****table_name**** (`str`) – Name of the results to be loaded

Returns
Results as a DataFrame

Return type
results (`pd.DataFrame`)

Raises
****ValueError**** – If the result doesn't exist

list() → DataFrame

List of all results available.

Parameters
****conn**** (`Optional[sqlite3.Connection]`) – Optional connection to use

Returns
Pandas DataFrame listing all results available in the model

Return type
df (`pd.DataFrame`)

new_record(*table_name: str*, *procedure: str = None*, *procedure_id: str = None*, *procedure_report: dict = None*, *timestamp: str = None*, *description: str = None*, *scenario: str = None*, *year: str = None*, *reference_table: str = 'links'*) → *ResultRecord*

Creates a new record for a result.

Parameters

- ****table_name**** (`str`) – Name of the table
- ****procedure**** (`str`, optional) – Name of the procedure
- ****procedure_id**** (`str`, optional) – ID of the procedure
- ****procedure_report**** (`dict`, optional) – Report associated with the procedure
- ****timestamp**** (`str`, optional) – Timestamp for the record
- ****description**** (`str`, optional) – Description of the record

Returns
A result record that can be manipulated in memory before saving

Return type
result_record (`ResultRecord`)

Raises
****ValueError**** – If a result with the same name already exists

reload() → None

Reloads the results from the database.

update_database() → None

Adds records to the results table for results found in the results database.

Modules

matrices
matrix_record
result_record
results

aequilibrae.project.data.matrices

Classes

Matrices(project)	Gateway into the matrices available/recorded in the model

class aequilibrae.project.data.matrices.**Matrices**(*project*)

Gateway into the matrices available/recorded in the model

check_exists(*name: str*) → bool

Checks whether a matrix with a given name exists

Returns

exists (bool): Does the matrix exist?

clear_database() → None

Removes records from the matrices database that do not exist in disk

delete_record(*matrix_name: str*) → None

Deletes a Matrix Record from the model and attempts to remove from disk

get_matrix(*matrix_name: str*) → *AequilibraeMatrix*

Returns an AequilibraE matrix available in the project

Raises an error if matrix does not exist

Arguments

matrix_name (str): Name of the matrix to be loaded

Returns

matrix (AequilibraeMatrix): Matrix object

get_record(*matrix_name: str*) → *MatrixRecord*

Returns a model Matrix Record for manipulation in memory

list() → DataFrame

List of all matrices available

Returns

df (pd.DataFrame): Pandas DataFrame listing all matrices available in the model

new_record(*name: str*, *file_name: str*, *matrix=None*) → *MatrixRecord*

Creates a new record for a matrix in disk, but does not save it

If the matrix file is not already on disk, it will fail

Arguments

name (str): Name of the matrix

file_name (str): Name of the file on disk

Returns
matrix_record (`MatrixRecord`): A matrix record that can be manipulated in memory before saving

reload()

Discards all memory matrices in memory and loads recreate them

update_database() → None

Adds records to the matrices database for matrix files found on disk

aequilibrae.project.data.matrix_record

Classes

MatrixRecord(data_set, project)	

class aequilibrae.project.data.matrix_record.**MatrixRecord**(*data_set: dict*, *project*)

delete()

Deletes this matrix record and the underlying data from disk

get_data() → *AequilibraeMatrix*

Returns the actual matrix for further computation

Returns
Matrix object

Return type
matrix (`AequilibraeMatrix`)

save()

Saves matrix record to the project database

update_cores()

Updates this matrix record with the matrix core count in disk

aequilibrae.project.data.result_record

Classes

ResultRecord(data_set, project[, ...])	Class for handling records of results in the AequilibraE project database.

class aequilibrae.project.data.result_record.**ResultRecord**(*data_set: dict*, *project*, *project_conn: Connection | None = None*, *results_conn: Connection | None = None*)

Class for handling records of results in the AequilibraE project database.

This class provides methods to save, delete, and retrieve result records and their data.

Parameters

- ****data_set**** (`dict`) – Dictionary containing the result record data.
- ****project**** – (`Project`): Project object this result record belongs to.

- ****project_conn**** (Optional[sqlite3.Connection]) – Connection to the project database. If None, the project's connection will be used.
- ****results_conn**** (Optional[sqlite3.Connection]) – Connection to the results database. If None, the project's results connection will be used.

delete() → None

Deletes this results record and the underlying data from disk.

Removes both the record from the project database and the data table from the results database.

get_data() → DataFrame

Returns the results data for further computation.

Returns
DataFrame containing the results data.

Return type
df (pd.DataFrame)

save() → None

Saves results record to the project database.

Creates a new record if it doesn't exist or updates an existing one.

set_data(*df: DataFrame*, ***kwargs*) → None

Set the results data corresponding to this record. Additionally saves this record.

Additional keyword arguments forwarded to the pd.DataFrame.to_sql method.

Parameters
****df**** (pd.DataFrame) – DataFrame object to save. Uses pd.DataFrame.to_sql.

aequilibrae.project.data.results

Classes

Results(project[, project_conn, results_conn])	Gateway into the results available/recorded in the model

class aequilibrae.project.data.results.**Results**(*project*, *project_conn: Connection | None = None*, *results_conn: Connection | None = None*)

Gateway into the results available/recorded in the model

check_exists(*table_name: str*) → bool

Checks whether a result with a given name exists.

Parameters
****table_name**** (str) – Name of the result to check

Returns
Does the result exist?

Return type
exists (bool)

clear_database() → None

Removes records from the results table that do not exist in the results database.

delete_record(*table_name: str*) → None

Deletes a ResultRecord from the model and attempts to remove it from the results database.

Parameters
****table_name**** (str) – Name of the result to delete

Raises
****ValueError**** – If the result doesn't exist

get_record(*table_name: str*) → *ResultRecord*

Returns a model ResultsRecord for manipulation in memory.

Parameters
****table_name**** (str) – Name of the result record to retrieve

Returns
The requested result record

Return type
record (ResultRecord)

Raises
****ValueError**** – If the result doesn't exist or was deleted

get_results(*table_name: str*) → DataFrame

Returns a DataFrame containing the results.

Raises an error if results do not exist.

Parameters
****table_name**** (str) – Name of the results to be loaded

Returns
Results as a DataFrame

Return type
results (pd.DataFrame)

Raises
****ValueError**** – If the result doesn't exist

list() → DataFrame

List of all results available.

Parameters
****conn**** (Optional[sqlite3.Connection]) – Optional connection to use

Returns
Pandas DataFrame listing all results available in the model

Return type
df (pd.DataFrame)

new_record(*table_name: str*, *procedure: str = None*, *procedure_id: str = None*, *procedure_report: dict = None*, *timestamp: str = None*, *description: str = None*, *scenario: str = None*, *year: str = None*, *reference_table: str = 'links'*) → *ResultRecord*

Creates a new record for a result.

Parameters

- ****table_name**** (str) – Name of the table
- ****procedure**** (str, optional) – Name of the procedure

- ****procedure_id**** (str, optional) – ID of the procedure
- ****procedure_report**** (dict, optional) – Report associated with the procedure
- ****timestamp**** (str, optional) – Timestamp for the record
- ****description**** (str, optional) – Description of the record

Returns
A result record that can be manipulated in memory before saving

Return type
result_record (ResultRecord)

Raises
****ValueError**** – If a result with the same name already exists

reload() → None

Reloads the results from the database.

update_database() → None

Adds records to the results table for results found in the results database.

aequilibrae.project.data_loader

Classes

DataLoader(path_to_file, table_name)	

class aequilibrae.project.data_loader.**DataLoader**(*path_to_file: PathLike*, *table_name: str*)

load_table() → GeoDataFrame | DataFrame

aequilibrae.project.database_connection

Functions

database_connection(db_type[, project_path])	
database_path(db_type[, project_path])	

aequilibrae.project.database_connection.**database_connection**(*db_type: str*, *project_path=None*) → Connection

aequilibrae.project.database_connection.**database_path**(*db_type: str*, *project_path=None*) → Path

aequilibrae.project.field_editor

Classes

FieldEditor(project, table_name)	Allows user to edit the project data tables

class aequilibrae.project.field_editor.**FieldEditor**(*project*, *table_name: str*)

Allows user to edit the project data tables

The field editor is used for two different purposes:

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

This is a general class used to manage all project's data tables accessible to the user and but it should be accessed directly from within the module corresponding to the data table one wants to edit. Example:

```
>>> project = create_example(project_path)

# To edit the fields of the link_types table
>>> lt_fields = project.network.link_types.fields

# To edit the fields of the modes table
>>> m_fields = project.network.modes.fields

>>> project.close()
```

Field descriptions are kept in the table *attributes_documentation*

add(*field_name: str*, *description: str*, *data_type='NUMERIC'*) → None

Adds new field to the data table

Arguments

field_name (str): Field to be added to the table. Must be a valid SQLite field name

description (str): Description of the field to be inserted in the metadata

data_type (str, *Optional*): Valid SQLite Data type. Default: "NUMERIC"

all_fields() → List[str]

Returns the list of fields available in the database

remove(*field_name: str*) → None

save() → None

Saves any field descriptions which my have been changed to the database and update layer statistics.

This is required for new fields to appear in applications like QGIS.

aequilibrae.project.network

class aequilibrae.project.network.**Link**(*dataset*, *project*)

A Link object represents a single record in the *links* table

```
>>> project = create_example(project_path)

>>> all_links = project.network.links

# Let's get a mode to work with
>>> modes = project.network.modes
>>> car_mode = modes.get('c')

# We can just get one link in specific
>>> link1 = all_links.get(3)
>>> link2 = all_links.get(17)

# We can find out which fields exist for the links
>>> which_fields_do_we_have = link1.data_fields()
```

(continues on next page)

(continued from previous page)

```
# And edit each one like this
>>> link1.lanes_ab = 3
>>> link1.lanes_ba = 2

# we can drop a mode from the link
>>> link1.drop_mode(car_mode)  # or link1.drop_mode('c')

# we can add a mode to the link
>>> link2.add_mode(car_mode)  # or link2.add_mode('c')

# Or set all modes at once
>>> link2.set_modes('cbtw')

# We can just save the link
>>> link1.save()
>>> link2.save()

>>> project.close()
```

add_mode(*mode: str | Mode*)

Adds a new mode to this link

Raises a warning if mode is already allowed on the link, and fails if mode does not exist

Arguments

mode_id (`str` or *Mode*): Mode_id of the mode or mode object to be added to the link

data_fields() → list

lists all data fields for the link, as available in the database

Returns

data fields (`list`): list of all fields available for editing

delete()

Deletes link from database

drop_mode(*mode: str | Mode*)

Removes a mode from this link

Raises a warning if mode is already NOT allowed on the link, and fails if mode does not exist

Arguments

mode_id (`str` or *Mode*): Mode_id of the mode or mode object to be removed from the link

save(*conn=None*)

Saves link to database

set_modes(*modes: str*)

Sets the modes acceptable for this link

Arguments

modes (`str`): string with all mode_ids to be assigned to this link

class aequilibrae.project.network.**LinkType**(*data_set: dict*, *project*)

A link_type object represents a single record in the *link_types* table

delete()

save()

class aequilibrae.project.network.**LinkTypes**(*net*)

Access to the API resources to manipulate the link_types table in the network.

```
>>> project = create_example(project_path)

>>> link_types = project.network.link_types

# We can get a dictionary of link types in the model
>>> all_link_types = link_types.all_types()

# And do a bulk change and save it
>>> for link_type_id, link_type_obj in all_link_types.items():
...     link_type_obj.beta = 1

# We can save changes for all link types in one go
>>> link_types.save()

# or just get one link_type in specific
>>> default_link_type = link_types.get('y')

# or just get it by name
>>> default_link_type = link_types.get_by_name('default')

# We can change the description of the link types
>>> default_link_type.description = 'My own new description'

# Let's say we are using alpha to store lane capacity during the night as 90% of␣
↪the standard
>>> default_link_type.alpha = 0.9 * default_link_type.lane_capacity

# To save this link types we can simply
>>> default_link_type.save()

# We can also create a completely new link_type and add to the model
>>> new_type = link_types.new('a')
>>> new_type.link_type = 'Arterial'  # Only ASCII letters and *_* allowed # other␣
↪fields are not mandatory

# We then save it to the database
>>> new_type.save()

# we can even keep editing and save it directly once we have added it to the␣
↪project
>>> new_type.lanes = 3
>>> new_type.lane_capacity = 1100
>>> new_type.save()

>>> project.close()
```

all_types() → dict

Returns a dictionary with all LinkType objects available in the model. link_type_id as key

delete(*link_type_id: str*) → None

Removes the link_type with *link_type_id* from the project

get(*link_type_id: str*) → *LinkType*

Get a link_type from the network by its *link_type_id*

get_by_name(*link_type: str*) → *LinkType*

Get a link_type from the network by its *link_type* (i.e. name)

new(*link_type_id: str*) → *LinkType*

save()

property fields: ***FieldEditor***

Returns a FieldEditor class instance to edit the Link_Types table fields and their metadata

class aequilibrae.project.network.**Links**(*net*)

Access to the API resources to manipulate the links table in the network

```
>>> project = create_example(project_path)

>>> all_links = project.network.links

# We can just get one link in specific
>>> link = all_links.get(1)

# We can save changes for all links we have edited so far
>>> all_links.save()

>>> project.close()
```

copy_link(*link_id: int*) → *Link*

Creates a copy of a link with a new id

It raises an error if link_id does not exist

Arguments

link_id (`int`): Id of the link to copy

Returns

link (*Link*): Link object for requested link_id

delete(*link_id: int*) → None

Removes the link with link_id from the project

Arguments

link_id (`int`): Id of a link to delete

extent() → Polygon

Queries the extent of the layer included in the model

Returns

model extent (`Polygon`): Shapely polygon with the bounding box of the layer.

get(*link_id: int*) → *Link*

Get a link from the network by its *link_id*

It raises an error if link_id does not exist

Arguments
link_id (`int`): Id of a link to retrieve

Returns
link (*Link*): Link object for requested link_id

new() → *Link*

Creates a new link

Returns
link (*Link*): A new link object populated only with link_id (not saved in the model yet)

refresh()

Refreshes all the links in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: GeoDataFrame

Returns all links data as a Pandas DataFrame

Returns
table (`GeoDataFrame`): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

sql = ''

Query sql for retrieving links

class aequilibrae.project.network.**Mode**(*mode_id: str*, *project*)

A mode object represents a single record in the *modes* table

save()

class aequilibrae.project.network.**Modes**(*net*)

Access to the API resources to manipulate the modes table in the network

```
>>> project = create_example(project_path)

>>> modes = project.network.modes

# We can get a dictionary of all modes in the model
>>> all_modes = modes.all_modes()

# And do a bulk change and save it
>>> for mode_id, mode_obj in all_modes.items():
...     mode_obj.beta = 1
...     mode_obj.save()

# or just get one mode in specific
>>> car_mode = modes.get('c')

# or just get this same mode by name
>>> car_mode = modes.get_by_name('car')
```

(continues on next page)

(continued from previous page)

```
# We can change the description of the mode
>>> car_mode.description = 'personal autos only'

# Let's say we are using alpha to store the PCE for a future year with much␣
↪smaller cars
>>> car_mode.alpha = 0.95

# To save this mode we can simply
>>> car_mode.save()

# We can also create a completely new mode and add to the model
>>> new_mode = modes.new('k')
>>> new_mode.mode_name = 'flying_car'  # Only ASCII letters and *_* allowed #␣
↪other fields are not mandatory

# We then explicitly add it to the network
>>> modes.add(new_mode)

# we can even keep editing and save it directly once we have added it to the␣
↪project
>>> new_mode.description = 'this is my new description'
>>> new_mode.save()

>>> project.close()
```

add(*mode:* Mode) → None

We add a mode to the project

all_modes() → dict

Returns a dictionary with all mode objects available in the model. mode_id as key

delete(*mode_id: str*) → None

Removes the mode with *mode_id* from the project

get(*mode_id: str*) → *Mode*

Get a mode from the network by its *mode_id*

get_by_name(*mode: str*) → *Mode*

Get a mode from the network by its *mode_name*

new(*mode_id: str*) → *Mode*

Returns a new mode with *mode_id* that can be added to the model later

property fields: ***FieldEditor***

Returns a FieldEditor class instance to edit the Modes table fields and their metadata

class aequilibrae.project.network.**Network**(*project*)

Network class. Member of an AequilibraE Project

build_graphs(*fields: list = None*, *modes: list = None*, *limit_to_area: Polygon = None*) → None

Builds graphs for all modes currently available in the model

When called, it overwrites all graphs previously created and stored in the networks' dictionary of graphs

Arguments

fields (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to specify which fields to use

modes (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to generate only those we need

limit_to_area (`Polygon`, *Optional*): When working with a very large model area, you may want to filter your database to a small area for your computation, which you can do by providing a polygon. The search is limited to a spatial index search, so it is very fast but NOT PRECISE.

To use the 'fields' parameter, a minimalistic option is the following

```
>>> project = create_example(project_path)

>>> fields = ['distance']
>>> project.network.build_graphs(fields, modes = ['c', 'w'])

>>> project.close()
```

convex_hull() → Polygon

Queries the model for the convex hull of the entire network

Returns

model coverage (`Polygon`): Shapely (Multi)polygon of the model network.

count_centroids() → int

Returns the number of centroids in the model

Returns

`int`: Number of centroids

count_links() → int

Returns the number of links in the model

Returns

`int`: Number of links

count_nodes() → int

Returns the number of nodes in the model

Returns

`int`: Number of nodes

create_from_gmns(*link_file_path: str*, *node_file_path: str*, *use_group_path: str = None*, *geometry_path: str = None*, *srid: int = 4326*) → None

Creates AequilibraE model from links and nodes in GMNS format.

Arguments

link_file_path (`str`): Path to a links csv file in GMNS format

node_file_path (`str`): Path to a nodes csv file in GMNS format

use_group_path (`str`, *Optional*): Path to a csv table containing groupings of uses. This helps AequilibraE know when a GMNS use is actually a group of other GMNS uses

geometry_path (`str`, *Optional*): Path to a csv file containing geometry information for a line object, if not specified in the link table

srid (`int`, *Optional*): Spatial Reference ID in which the GMNS geometries were created

create_from_osm(*model_area: Polygon | None = None*, *place_name: str | None = None*, *modes=('car', 'transit', 'bicycle', 'walk')*, *clean=True*) → None

Downloads the network from OpenStreetMap (OSM)

Arguments

area (`Polygon`, *Optional*): Polygon for which the network will be downloaded. If not provided, a place name would be required

place_name (`str`, *Optional*): If not downloading with East-West-North-South boundingbox, this is required

modes (`tuple`, *Optional*): List of all modes to be downloaded. Defaults to the modes in the parameter file

clean (`bool`, *Optional*): Keeps only the links that intersects the model area polygon. Defaults to `True`. Does not apply to networks downloaded with a place name

```
>>> project = Project()
>>> project.new(project_path)

# Now we can import the network for any place we want
>>> project.network.create_from_osm(place_name="my_beautiful_hometown")

>>> project.close()
```

export_to_gmns(*path: str*)

Exports AequilibraE network to csv files in GMNS format.

Arguments

path (`str`): Output folder path.

extent()

Queries the extent of the network included in the model

Returns

model extent (`Polygon`): Shapely polygon with the bounding box of the model network.

list_modes()

Returns a list of all the modes in this model

Returns

`list`: List of all modes

set_time_field(*time_field: str*) → None

Set the time field for all graphs built in the model

Arguments

time_field (`str`): Network field with travel time information

skimmable_fields()

Returns a list of all fields that can be skimmed

Returns

`list`: List of all fields that can be skimmed

graphs: Dict[*Graph*]

link_types: *LinkTypes* = None

protected_fields = ['ogc_fid', 'geometry']

req_link_flds = ['link_id', 'a_node', 'b_node', 'direction', 'distance', 'modes', 'link_type']

req_node_flds = ['node_id', 'is_centroid']

signal = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.project.network.**Node**(*dataset*, *project*)

A Node object represents a single record in the *nodes* table

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

>>> project = create_example(project_path)

>>> all_nodes = project.network.nodes

# We can just get one link in specific
>>> node1 = all_nodes.get(7)

# We can find out which fields exist for the links
>>> which_fields_do_we_have = node1.data_fields()

# It success if the node_id already does not exist
>>> node1.renumber(998877)

>>> node1.geometry = Point(1,2)

# We can just save the node
>>> node1.save()

>>> project.close()
```

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *conn: Connection | None = None*, *area: Polygon | None = None*)

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting the zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the provided area.

The selection of the nodes that will be connected is done simply by computing running the KMeans2 clustering algorithm from SciPy and selecting the nodes closest to each cluster centroid.

When there are no node candidates inside the provided area, is it progressively expanded until at least one candidate is found.

If fewer candidates than required connectors are found, all candidates are connected.

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered. eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

area (`Polygon`, *Optional*): Area limiting the search for connectors

data_fields() → list

lists all data fields for the node, as available in the database

Returns
data fields (list): list of all fields available for editing

renumber(*new_id: int*)

Renumbers the node in the network

Logs a warning if another node already exists with this node_id

Arguments
new_id (int): New node_id

save()

Saves node to database

class aequilibrae.project.network.**Nodes**(*net*)

Access to the API resources to manipulate the nodes table in the network

```
>>> project = create_example(project_path)

>>> all_nodes = project.network.nodes

# We can just get one link in specific
>>> node = all_nodes.get(21)

# We can save changes for all nodes we have edited so far
>>> all_nodes.save()

>>> project.close()
```

extent() → Polygon

Queries the extent of the layer included in the model

Returns
model extent (Polygon): Shapely polygon with the bounding box of the layer.

get(*node_id: int*) → *Node*

Get a node from the network by its node_id

It raises an error if node_id does not exist

Arguments
node_id (int): ID of a node to retrieve

Returns
node (*Node*): Node object for requested node_id

new_centroid(*node_id: int*) → *Node*

Creates a new centroid with a given ID

Arguments
node_id (int): ID of the centroid to be created

refresh()

Refreshes all the nodes in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: GeoDataFrame

Returns all nodes data as a Pandas DataFrame

Returns

table (GeoDataFrame): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

property lonlat: DataFrame

Returns all nodes lon/lat coords as a Pandas DataFrame

Returns

table (DataFrame): Pandas DataFrame with all the nodes, with geometry as lon/lat

sql = ''

Query sql for retrieving nodes

class aequilibrae.project.network.**Period**(*dataset*, *project*)

A Period object represents a single record in the *periods* table

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period1 = all_periods.get(1)

# We can find out which fields exist for the period
>>> which_fields_do_we_have = period1.data_fields()

>>> project.close()
```

data_fields() → list

Lists all data fields for the period, as available in the database

Returns

data fields (list): list of all fields available for editing

renumber(*new_id: int*)

Renumbers the period in the network

Logs a warning if another period already exists with this period_id

Arguments

new_id (int): New period_id

save()

Saves period to database

class aequilibrae.project.network.**Periods**(*net*)

Access to the API resources to manipulate the periods table in the network

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period = all_periods.get(1)

# We can save changes for all periods we have edited so far
>>> all_periods.save()

>>> project.close()
```

extent()

Queries the extent of the layer included in the model

Returns

model extent (Polygon): Shapely polygon with the bounding box of the layer.

get(*period_id: int*) → *Period*

Get a period from the network by its **period_id**

It raises an error if period_id does not exist

Arguments

period_id (int): Id of a period to retrieve

Returns

period (*Period*): Period object for requested period_id

new_period(*period_id: int*, *start: int*, *end: int*, *description: str = None*) → *Period*

Creates a new period with a given ID

Arguments

period_id (int): Id of the centroid to be created

start (int): Start time of the period to be created

end (int): End time of the period to be created

description (str): Optional human readable description of the time period e.g. '1pm - 5pm'

refresh()

Refreshes all the periods in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: DataFrame

Returns all periods data as a Pandas DataFrame

Returns

table (DataFrame): Pandas DataFrame with all the periods

property default_period: *Period*

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

sql = ''
Query sql for retrieving periods

Modules

connector_creation
gmns_builder
gmns_exporter
haversine
link
link_type
link_types
links
mode
modes
network
node
nodes
osm
period
periods
safe_class

aequilibrae.project.network.connector_creation

Functions

bulk_connector_creation(conn, project_nodes, ...)	Creates or updates centroid connectors between zone centroids and network nodes.
connector_creation(zone_id, mode_id, ...[, ...])	
k_nearest(k, centroids, nodes, ...)	Finds the k nearest nodes to each centroid using a KDTree spatial index.
k_nearest_in_zone(k, zones, centroids, ...)	Finds the k nearest nodes within each zone to the corresponding centroid.
normalise_mode_strings(x)	Normalises a collection of mode strings by sorting unique characters.

aequilibrae.project.network.connector_creation.**bulk_connector_creation**(*conn: Connection*, *project_nodes: GeoDataFrame*, *project_links: GeoDataFrame*, *project_zones: GeoDataFrame*, *modes: list[str]*, *k_connectors: int = 1*, *limit_to_zone: bool = True*, *distance_upper_bound: float = inf*, *projected_crs: str | int | None = None*)

Creates or updates centroid connectors between zone centroids and network nodes.

This function generates k-nearest neighbour connections from each zone centroid to nearby network nodes that support the specified transport modes. It can either limit connections to nodes within the same zone or find the globally nearest nodes.

Arguments

conn (`Connection`): Database connection for executing SQL operations.

project_nodes (`gpd.GeoDataFrame`): GeoDataFrame containing network nodes with columns including node_id, is_centroid, modes, and geometry.

project_links (`gpd.GeoDataFrame`): GeoDataFrame containing network links with columns including link_id, a_node, b_node, modes, direction, and link_type.

project_zones (`gpd.GeoDataFrame`): GeoDataFrame containing zone polygons with columns including zone_id and geometry.

modes (`list[str]`): List of transport mode strings to create connectors for.

k_connectors (`int`, *Optional*): Number of nearest neighbour connections to create per centroid. Defaults to 1.

limit_to_zone (`bool`, *Optional*): If True, only connects to nodes within the same zone. If False, finds globally nearest nodes. Defaults to True.

distance_upper_bound (`float`, *Optional*): Maximum distance for connections. Defaults to infinity.

projected_crs (`str | int`, *Optional*): Coordinate reference system for distance calculations. If None, uses the CRS from the input data.

aequilibrae.project.network.connector_creation.**connector_creation**(*zone_id: int*, *mode_id: str*, *network*, *proj_nodes*, *proj_links*, *link_types=''*, *connectors=1*, *conn: Connection | None = None*, *delimiting_area: Polygon = None*)

aequilibrae.project.network.connector_creation.**k_nearest**(*k: int*, *centroids: GeoDataFrame*, *nodes: GeoDataFrame*, *distance_upper_bound: float*, *crs: int | str*)

Finds the k nearest nodes to each centroid using a KDTree spatial index.

Arguments

k (`int`): Number of nearest neighbours to find for each centroid.

centroids (`gpd.GeoDataFrame`): GeoDataFrame containing centroid points with node_id and geometry columns.

nodes (`gpd.GeoDataFrame`): GeoDataFrame containing network nodes with node_id and geometry columns to search within.

distance_upper_bound (`float`): Maximum distance for neighbour search.

crs (`int | str`): Coordinate reference system for distance calculations.

Returns

pd.DataFrame: DataFrame with columns a_node (centroid), b_node (nearest node), and distance, sorted by a_node and distance.

aequilibrae.project.network.connector_creation.**k_nearest_in_zone**(*k: int*, *zones: GeoDataFrame*, *centroids: GeoDataFrame*, *nodes: GeoDataFrame*, *distance_upper_bound: float*, *crs: int | str*)

Finds the k nearest nodes within each zone to the corresponding centroid.

Uses spatial indexing to first determine which nodes fall within each zone, then calculates distances only between centroids and nodes in the same zone.

Arguments

k (`int`): Number of nearest neighbours to find for each centroid.

zones (`gpd.GeoDataFrame`): GeoDataFrame containing zone polygons with zone_id and geometry columns.

centroids (`gpd.GeoDataFrame`): GeoDataFrame containing centroid points with node_id and geometry columns, corresponding to zones.

nodes (`gpd.GeoDataFrame`): GeoDataFrame containing network nodes with node_id and geometry columns to search within.

distance_upper_bound (`float`): Maximum distance for neighbour search.

crs (`int | str`): Coordinate reference system for distance calculations.

Returns

pd.DataFrame: DataFrame with columns a_node (centroid), b_node (nearest node), and distance, limited to k nearest nodes per centroid within the same zone.

aequilibrae.project.network.connector_creation.**normalise_mode_strings**(*x*)

Normalises a collection of mode strings by sorting unique characters.

Takes a sequence of mode strings and returns a single string containing unique characters sorted alphabetically.

Arguments

x (`Iterable[str]`): Collection of mode strings to normalise.

Returns

str: Normalised string with unique characters sorted alphabetically.

aequilibrae.project.network.gmns_builder

Functions

resolve_recusive_dict(base_dict)	Resolve each entry in the graph.

Classes

GMNSBuilder(net, link_path, node_path[, ...])	

class aequilibrae.project.network.gmns_builder.**GMNSBuilder**(*net*, *link_path: str*, *node_path: str*, *uses_path: str = None*, *geom_path: str = None*, *srid: int = 4326*)

correct_geometries()

doWork()

get_ab_lists(*direction*)

get_aeq_direction()

maybe_transform_srid(*srid*)

save_modes_to_aeq()

save_to_database(*links_fields*, *nodes_fields*)

save_types_to_aeq()

aequilibrae.project.network.gmns_builder.**resolve_recusive_dict**(*base_dict*)

Resolve each entry in the graph.

aequilibrae.project.network.gmns_exporter

Classes

GMNSExporter(net, path)	

class aequilibrae.project.network.gmns_exporter.**GMNSExporter**(*net*, *path*)

doWork()

reorder_fields()

update_direction_field()

update_field_names()

Updates field names according to equivalency between AequilibraE and GMNS fields.

update_modes_fields()

Updates AequilibraE modes table so it can be exported as a GMNS use_definition table.

aequilibrae.project.network.haversine

Functions

haversine(lon1, lat1, lon2, lat2)	Calculate the great circle distance between two points on the earth (specified in decimal degrees)

aequilibrae.project.network.haversine.**haversine**(*lon1*, *lat1*, *lon2*, *lat2*)

Calculate the great circle distance between two points on the earth (specified in decimal degrees)

aequilibrae.project.network.link

Classes

Link(dataset, project)	A Link object represents a single record in the *links* table

class aequilibrae.project.network.link.**Link**(*dataset*, *project*)

A Link object represents a single record in the *links* table

```
>>> project = create_example(project_path)

>>> all_links = project.network.links

# Let's get a mode to work with
>>> modes = project.network.modes
>>> car_mode = modes.get('c')

# We can just get one link in specific
>>> link1 = all_links.get(3)
>>> link2 = all_links.get(17)

# We can find out which fields exist for the links
>>> which_fields_do_we_have = link1.data_fields()

# And edit each one like this
>>> link1.lanes_ab = 3
>>> link1.lanes_ba = 2

# we can drop a mode from the link
>>> link1.drop_mode(car_mode)  # or link1.drop_mode('c')

# we can add a mode to the link
>>> link2.add_mode(car_mode)  # or link2.add_mode('c')

# Or set all modes at once
>>> link2.set_modes('cbtw')

# We can just save the link
>>> link1.save()
>>> link2.save()

>>> project.close()
```

add_mode(*mode: str* | Mode)

Adds a new mode to this link

Raises a warning if mode is already allowed on the link, and fails if mode does not exist

Arguments

mode_id (`str` or *Mode*): Mode_id of the mode or mode object to be added to the link

data_fields() → list

lists all data fields for the link, as available in the database

Returns

data fields (`list`): list of all fields available for editing

delete()

Deletes link from database

drop_mode(*mode: str* | Mode)

Removes a mode from this link

Raises a warning if mode is already NOT allowed on the link, and fails if mode does not exist

Arguments
mode_id (`str` or *Mode*): Mode_id of the mode or mode object to be removed from the link

save(*conn=None*)

Saves link to database

set_modes(*modes: str*)

Sets the modes acceptable for this link

Arguments
modes (`str`): string with all mode_ids to be assigned to this link

aequilibrae.project.network.link_type

Classes

LinkType(data_set, project)	A link_type object represents a single record in the *link_types* table

class aequilibrae.project.network.link_type.**LinkType**(*data_set: dict*, *project*)

A link_type object represents a single record in the *link_types* table

delete()

save()

aequilibrae.project.network.link_types

Classes

LinkTypes(net)	Access to the API resources to manipulate the link_types table in the network.

class aequilibrae.project.network.link_types.**LinkTypes**(*net*)

Access to the API resources to manipulate the link_types table in the network.

```
>>> project = create_example(project_path)

>>> link_types = project.network.link_types

# We can get a dictionary of link types in the model
>>> all_link_types = link_types.all_types()

# And do a bulk change and save it
>>> for link_type_id, link_type_obj in all_link_types.items():
...     link_type_obj.beta = 1

# We can save changes for all link types in one go
>>> link_types.save()

# or just get one link_type in specific
```

(continues on next page)

(continued from previous page)

```
>>> default_link_type = link_types.get('y')

# or just get it by name
>>> default_link_type = link_types.get_by_name('default')

# We can change the description of the link types
>>> default_link_type.description = 'My own new description'

# Let's say we are using alpha to store lane capacity during the night as 90% of␣
↪the standard
>>> default_link_type.alpha = 0.9 * default_link_type.lane_capacity

# To save this link types we can simply
>>> default_link_type.save()

# We can also create a completely new link_type and add to the model
>>> new_type = link_types.new('a')
>>> new_type.link_type = 'Arterial'  # Only ASCII letters and *_* allowed # other␣
↪fields are not mandatory

# We then save it to the database
>>> new_type.save()

# we can even keep editing and save it directly once we have added it to the␣
↪project
>>> new_type.lanes = 3
>>> new_type.lane_capacity = 1100
>>> new_type.save()

>>> project.close()
```

all_types() → dict

Returns a dictionary with all LinkType objects available in the model. link_type_id as key

delete(*link_type_id: str*) → None

Removes the link_type with *link_type_id* from the project

get(*link_type_id: str*) → *LinkType*

Get a link_type from the network by its *link_type_id*

get_by_name(*link_type: str*) → *LinkType*

Get a link_type from the network by its *link_type* (i.e. name)

new(*link_type_id: str*) → *LinkType*

save()

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the Link_Types table fields and their metadata

aequilibrae.project.network.links

Classes

Links(net)	Access to the API resources to manipulate the links table in the network

class aequilibrae.project.network.links.**Links**(*net*)

Access to the API resources to manipulate the links table in the network

```
>>> project = create_example(project_path)

>>> all_links = project.network.links

# We can just get one link in specific
>>> link = all_links.get(1)

# We can save changes for all links we have edited so far
>>> all_links.save()

>>> project.close()
```

copy_link(*link_id: int*) → *Link*

Creates a copy of a link with a new id

It raises an error if link_id does not exist

Arguments
link_id (`int`): Id of the link to copy

Returns
link (`Link`): Link object for requested link_id

delete(*link_id: int*) → None

Removes the link with link_id from the project

Arguments
link_id (`int`): Id of a link to delete

extent() → Polygon

Queries the extent of the layer included in the model

Returns
model extent (`Polygon`): Shapely polygon with the bounding box of the layer.

get(*link_id: int*) → *Link*

Get a link from the network by its *link_id*

It raises an error if link_id does not exist

Arguments
link_id (`int`): Id of a link to retrieve

Returns
link (`Link`): Link object for requested link_id

new() → *Link*

Creates a new link

Returns

link (Link): A new link object populated only with link_id (not saved in the model yet)

refresh()

Refreshes all the links in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: GeoDataFrame

Returns all links data as a Pandas DataFrame

Returns

table (GeoDataFrame): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

sql = ''

Query sql for retrieving links

aequilibrae.project.network.mode

Classes

Mode(mode_id, project)	A mode object represents a single record in the *modes* table

class aequilibrae.project.network.mode.**Mode**(*mode_id: str*, *project*)

A mode object represents a single record in the *modes* table

save()

aequilibrae.project.network.modes

Classes

Modes(net)	Access to the API resources to manipulate the modes table in the network

class aequilibrae.project.network.modes.**Modes**(*net*)

Access to the API resources to manipulate the modes table in the network

```
>>> project = create_example(project_path)

>>> modes = project.network.modes

# We can get a dictionary of all modes in the model
```

(continues on next page)

(continued from previous page)

```
>>> all_modes = modes.all_modes()

# And do a bulk change and save it
>>> for mode_id, mode_obj in all_modes.items():
...     mode_obj.beta = 1
...     mode_obj.save()

# or just get one mode in specific
>>> car_mode = modes.get('c')

# or just get this same mode by name
>>> car_mode = modes.get_by_name('car')

# We can change the description of the mode
>>> car_mode.description = 'personal autos only'

# Let's say we are using alpha to store the PCE for a future year with much␣
↪smaller cars
>>> car_mode.alpha = 0.95

# To save this mode we can simply
>>> car_mode.save()

# We can also create a completely new mode and add to the model
>>> new_mode = modes.new('k')
>>> new_mode.mode_name = 'flying_car'  # Only ASCII letters and *_* allowed #␣
↪other fields are not mandatory

# We then explicitly add it to the network
>>> modes.add(new_mode)

# we can even keep editing and save it directly once we have added it to the␣
↪project
>>> new_mode.description = 'this is my new description'
>>> new_mode.save()

>>> project.close()
```

add(*mode:* Mode) → None

We add a mode to the project

all_modes() → dict

Returns a dictionary with all mode objects available in the model. mode_id as key

delete(*mode_id: str*) → None

Removes the mode with *mode_id* from the project

get(*mode_id: str*) → *Mode*

Get a mode from the network by its *mode_id*

get_by_name(*mode: str*) → *Mode*

Get a mode from the network by its *mode_name*

new(*mode_id: str*) → *Mode*

Returns a new mode with *mode_id* that can be added to the model later

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the Modes table fields and their metadata

aequilibrae.project.network.network

Classes

Network(project)	Network class.

class aequilibrae.project.network.network.**Network**(*project*)

Network class. Member of an AequilibraE Project

build_graphs(*fields: list = None*, *modes: list = None*, *limit_to_area: Polygon = None*) → None

Builds graphs for all modes currently available in the model

When called, it overwrites all graphs previously created and stored in the networks' dictionary of graphs

Arguments

fields (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to specify which fields to use

modes (`list`, *Optional*): When working with very large graphs with large number of fields in the database, it may be useful to generate only those we need

limit_to_area (`Polygon`, *Optional*): When working with a very large model area, you may want to filter your database to a small area for your computation, which you can do by providing a polygon. The search is limited to a spatial index search, so it is very fast but NOT PRECISE.

To use the 'fields' parameter, a minimalistic option is the following

```
>>> project = create_example(project_path)

>>> fields = ['distance']
>>> project.network.build_graphs(fields, modes = ['c', 'w'])

>>> project.close()
```

convex_hull() → Polygon

Queries the model for the convex hull of the entire network

Returns

model coverage (`Polygon`): Shapely (Multi)polygon of the model network.

count_centroids() → int

Returns the number of centroids in the model

Returns

`int`: Number of centroids

count_links() → int

Returns the number of links in the model

Returns

`int`: Number of links

count_nodes() → int

Returns the number of nodes in the model

Returns
`int`: Number of nodes

create_from_gmns(*link_file_path: str*, *node_file_path: str*, *use_group_path: str = None*, *geometry_path: str = None*, *srid: int = 4326*) → None

Creates AequilibraE model from links and nodes in GMNS format.

Arguments
link_file_path (`str`): Path to a links csv file in GMNS format

node_file_path (`str`): Path to a nodes csv file in GMNS format

use_group_path (`str`, *Optional*): Path to a csv table containing groupings of uses. This helps AequilibraE know when a GMNS use is actually a group of other GMNS uses

geometry_path (`str`, *Optional*): Path to a csv file containing geometry information for a line object, if not specified in the link table

srid (`int`, *Optional*): Spatial Reference ID in which the GMNS geometries were created

create_from_osm(*model_area: Polygon | None = None*, *place_name: str | None = None*, *modes=('car', 'transit', 'bicycle', 'walk')*, *clean=True*) → None

Downloads the network from OpenStreetMap (OSM)

Arguments
area (`Polygon`, *Optional*): Polygon for which the network will be downloaded. If not provided, a place name would be required

place_name (`str`, *Optional*): If not downloading with East-West-North-South boundingbox, this is required

modes (`tuple`, *Optional*): List of all modes to be downloaded. Defaults to the modes in the parameter file

clean (`bool`, *Optional*): Keeps only the links that intersects the model area polygon. Defaults to `True`. Does not apply to networks downloaded with a place name

```
>>> project = Project()
>>> project.new(project_path)

# Now we can import the network for any place we want
>>> project.network.create_from_osm(place_name="my_beautiful_hometown")

>>> project.close()
```

export_to_gmns(*path: str*)

Exports AequilibraE network to csv files in GMNS format.

Arguments
path (`str`): Output folder path.

extent()

Queries the extent of the network included in the model

Returns
model extent (`Polygon`): Shapely polygon with the bounding box of the model network.

list_modes()

Returns a list of all the modes in this model

Returns

`list`: List of all modes

set_time_field(*time_field: str*) → None

Set the time field for all graphs built in the model

Arguments

time_field (`str`): Network field with travel time information

skimmable_fields()

Returns a list of all fields that can be skimmed

Returns

`list`: List of all fields that can be skimmed

graphs: Dict[*Graph*]

link_types: *LinkTypes* = None

protected_fields = ['ogc_fid', 'geometry']

req_link_flds = ['link_id', 'a_node', 'b_node', 'direction', 'distance', 'modes', 'link_type']

req_node_flds = ['node_id', 'is_centroid']

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.project.network.node

Classes

Node(dataset, project)	A Node object represents a single record in the *nodes* table

class aequilibrae.project.network.node.**Node**(*dataset*, *project*)

A Node object represents a single record in the *nodes* table

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

>>> project = create_example(project_path)

>>> all_nodes = project.network.nodes

# We can just get one link in specific
>>> node1 = all_nodes.get(7)

# We can find out which fields exist for the links
>>> which_fields_do_we_have = node1.data_fields()

# It success if the node_id already does not exist
>>> node1.renumber(998877)
```

(continues on next page)

(continued from previous page)

```
>>> node1.geometry = Point(1,2)

# We can just save the node
>>> node1.save()

>>> project.close()
```

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *conn: Connection | None = None*, *area: Polygon | None = None*)

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting the zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the provided area.

The selection of the nodes that will be connected is done simply by computing running the KMeans2 clustering algorithm from SciPy and selecting the nodes closest to each cluster centroid.

When there are no node candidates inside the provided area, is it progressively expanded until at least one candidate is found.

If fewer candidates than required connectors are found, all candidates are connected.

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered. eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

area (`Polygon`, *Optional*): Area limiting the search for connectors

data_fields() → list

lists all data fields for the node, as available in the database

Returns

data fields (`list`): list of all fields available for editing

renumber(*new_id: int*)

Renumbers the node in the network

Logs a warning if another node already exists with this node_id

Arguments

new_id (`int`): New node_id

save()

Saves node to database

aequilibrae.project.network.nodes

Classes

Nodes(net)	Access to the API resources to manipulate the nodes table in the network

class aequilibrae.project.network.nodes.**Nodes**(*net*)

Access to the API resources to manipulate the nodes table in the network

```
>>> project = create_example(project_path)

>>> all_nodes = project.network.nodes

# We can just get one link in specific
>>> node = all_nodes.get(21)

# We can save changes for all nodes we have edited so far
>>> all_nodes.save()

>>> project.close()
```

extent() → Polygon

Queries the extent of the layer included in the model

Returns
model extent (Polygon): Shapely polygon with the bounding box of the layer.

get(*node_id: int*) → *Node*

Get a node from the network by its node_id

It raises an error if node_id does not exist

Arguments
node_id (int): ID of a node to retrieve

Returns
node (Node): Node object for requested node_id

new_centroid(*node_id: int*) → *Node*

Creates a new centroid with a given ID

Arguments
node_id (int): ID of the centroid to be created

refresh()

Refreshes all the nodes in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: GeoDataFrame

Returns all nodes data as a Pandas DataFrame

Returns
table (GeoDataFrame): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

property lonlat: DataFrame

Returns all nodes lon/lat coords as a Pandas DataFrame

Returns
table (DataFrame): Pandas DataFrame with all the nodes, with geometry as lon/lat

sql = ''

Query sql for retrieving nodes

aequilibrae.project.network.osm

aequilibrae.project.network.osm.**placegetter**(*place: str*) → Tuple[None | List[float], list]

Send a request to the Nominatim API via HTTP GET and return a geometry polygon for the region we are querying

Parameters

- **place** (*str*) – Name of the place we want to download a network for
- **http** (*Adapted from*)

Modules

model_area_gridding
osm_builder
osm_downloader
osm_params
place_getter

aequilibrae.project.network.osm.model_area_gridding

Functions

geometry_grid(model_area, srid)

aequilibrae.project.network.osm.model_area_gridding.**geometry_grid**(*model_area*, *srid*) → GeoDataFrame

aequilibrae.project.network.osm.osm_builder

Classes

OSMBuilder(data, project, model_area, clean)

class aequilibrae.project.network.osm.osm_builder.**OSMBuilder**(*data*, *project*, *model_area: Polygon*, *clean: bool*)

static get_link_field_type(*field_name*)

static get_link_fields()

doWork()

importing_network(*conn*)

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.project.network.osm.osm_downloader

" Large portions of this code were adopted from OSMNx, by Geoff Boeing.

Although attempts to use OSMNx were made (including refactoring its entire code base as a contribution to that package), it became clear that its integration with libraries not available with QGIS' Python distribution was too tight, and was therefore not practical to detach them in order to use OSMNx as a dependency or submodule

For the original work, please see https://github.com/gboeing/osmnx

Classes

OSMDownloader(polygons, modes[, logger])	

class aequilibrae.project.network.osm.osm_downloader.**OSMDownloader**(*polygons: List[Polygon]*, *modes*, *logger: Logger = None*)

doWork()

get_osm_filter(*modes: list*) → str

loosely adapted from http://www.github.com/gboeing/osmnx

overpass_request(*data*, *pause_duration=None*, *timeout=180*, *error_pause_duration=None*)

Send a request to the Overpass API via HTTP POST and return the JSON response.

Arguments

data:obj:`dict` or `OrderedDict`): key-value pairs of parameters to post to the API**pause_duration** (int): how long to pause in seconds before requests, if None, will query API status endpoint to find when next slot is available **timeout** (int): the timeout interval for the requests library **error_pause_duration**(int): how long to pause in seconds before re-trying requests if error

Returns

dict

signal = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.project.network.osm.osm_params

Functions

default_headers()	

aequilibrae.project.network.osm.osm_params.**default_headers**()

aequilibrae.project.network.osm.place_getter

Functions

placegetter(place)	Send a request to the Nominatim API via HTTP GET and return a geometry polygon for the region we are querying

aequilibrae.project.network.osm.place_getter.**placegetter**(*place: str*) → Tuple[None | List[float], list]

Send a request to the Nominatim API via HTTP GET and return a geometry polygon for the region we are querying

Parameters

- **place** (*str*) – Name of the place we want to download a network for
- **http** (*Adapted from*)

aequilibrae.project.network.period

Classes

Period(dataset, project)	A Period object represents a single record in the *periods* table

class aequilibrae.project.network.period.**Period**(*dataset*, *project*)

A Period object represents a single record in the *periods* table

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period1 = all_periods.get(1)

# We can find out which fields exist for the period
>>> which_fields_do_we_have = period1.data_fields()

>>> project.close()
```

data_fields() → list

Lists all data fields for the period, as available in the database

Returns

data fields (list): list of all fields available for editing

renumber(*new_id: int*)

Renumbers the period in the network

Logs a warning if another period already exists with this period_id

Arguments

new_id (int): New period_id

save()

Saves period to database

aequilibrae.project.network.periods

Classes

Periods(net)	Access to the API resources to manipulate the periods table in the network

class aequilibrae.project.network.periods.**Periods**(*net*)

Access to the API resources to manipulate the periods table in the network

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

>>> all_periods = project.network.periods

# We can just get one link in specific
>>> period = all_periods.get(1)

# We can save changes for all periods we have edited so far
>>> all_periods.save()

>>> project.close()
```

extent()

Queries the extent of the layer included in the model

Returns

model extent (Polygon): Shapely polygon with the bounding box of the layer.

get(*period_id: int*) → *Period*

Get a period from the network by its **period_id**

It raises an error if period_id does not exist

Arguments

period_id (int): Id of a period to retrieve

Returns

period (Period): Period object for requested period_id

new_period(*period_id: int*, *start: int*, *end: int*, *description: str = None*) → *Period*

Creates a new period with a given ID

Arguments

period_id (int): Id of the centroid to be created

start (int): Start time of the period to be created

end (int): End time of the period to be created

description (str): Optional human readable description of the time period e.g. '1pm - 5pm'

refresh()

Refreshes all the periods in memory

refresh_fields() → None

After adding a field one needs to refresh all the fields recognized by the software

save()

property data: DataFrame

Returns all periods data as a Pandas DataFrame

Returns
table (DataFrame): Pandas DataFrame with all the periods

property default_period: *Period*

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

sql = ''

Query sql for retrieving periods

aequilibrae.project.network.safe_class

Classes

SafeClass(data_set, project)	

class aequilibrae.project.network.safe_class.**SafeClass**(*data_set: dict*, *project*)

aequilibrae.project.project

Classes

Project()	AequilibraE project class

class aequilibrae.project.project.**Project**

AequilibraE project class

Listing 5: Create Project

```
>>> new_project = Project()
>>> new_project.new(project_path)

# Safely closes the project
>>> new_project.close()
```

Listing 6: Open Project

```
>>> existing_project = Project()
>>> existing_project.open(project_path)

>>> existing_project.close()
```

classmethod from_path(*project_folder*)

activate()

check_file_indices() → None

Makes results_database.sqlite and the matrices folder compatible with project database

clone_scenario(*scenario_name: str*, *description: str = ''*)

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 = ''*)

Creates an empty scenario, without any links, nodes, and zones.

Arguments

scenario_name (`str`): scenario name

description (`str`): useful scenario description

deactivate()

list_scenarios()

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

new(*project_path: str*) → None

Creates a new project

Arguments

project_path (`str`): Full path to the project data folder. If folder exists, it will fail

open(*project_path: str*) → None

Loads project from disk

Arguments

project_path (`str`): Full path to the project data folder. If the project inside does not exist, it will fail.

upgrade(*ignore_project: bool = False*, *ignore_transit: bool = False*, *ignore_results: bool = False*)

Find and apply all applicable migrations.

All database upgrades are 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*)

Switch the active scenario.

Arguments

scenario_name (str): name of the scenario to be activated

property about: *About*

property db_connection

property db_connection_spatial

Deprecated alias for db_connection, which is now a spatial connection.

property logger: Logger

property matrices: *Matrices*

property network: *Network*

property parameters: dict

property path_to_file

property project_base_path

property project_parameters: *Parameters*

property results: *Results*

property results_connection

property run

Load and return the AequilibraE run module with the default arguments from parameters.yml partially applied.

Refer to run/__init__.py file within the project folder for documentation.

property transit: *Transit*

property transit_connection

property zoning

aequilibrae.project.project_cleaning

Functions

clean(project)

aequilibrae.project.project_cleaning.**clean**(*project*)

aequilibrae.project.project_creation

Functions

add_triggers(conn, logger, db_type)	Adds consistency triggers to the project
create_base_tables(conn, logger, db_type)	
initialize_tables(logger, db_type, conn)	
recreate_columns(conn, logger, table, old_table)	Recreate columns for a table if any were added.
remove_triggers(conn, logger, db_type[, ...])	
run_queries_from_sql_file(conn, logger, qry_file)	

aequilibrae.project.project_creation.**add_triggers**(*conn: Connection*, *logger: Logger*, *db_type: str*) → None

Adds consistency triggers to the project

aequilibrae.project.project_creation.**create_base_tables**(*conn: Connection*, *logger: Logger*, *db_type: str*) → None

aequilibrae.project.project_creation.**initialize_tables**(*logger*, *db_type: str*, *conn: Connection*) → None

aequilibrae.project.project_creation.**recreate_columns**(*conn: Connection*, *logger: Logger*, *table: str*, *old_table: str*) → dict[str, str]

Recreate columns for a table if any were added. Returns a dict of the old column names to type

aequilibrae.project.project_creation.**remove_triggers**(*conn: Connection*, *logger: Logger*, *db_type: str*, *use_aequilibrae_prefix: bool = True*) → None

aequilibrae.project.project_creation.**run_queries_from_sql_file**(*conn: Connection*, *logger: Logger*, *qry_file: Path*) → None

aequilibrae.project.scenario

Classes

Scenario(name, base_path, path_to_file)	Represents a modelling scenario within an AequilibraE project.

class aequilibrae.project.scenario.**Scenario**(*name: str*, *base_path: Path*, *path_to_file: Path*)

Represents a modelling scenario within an AequilibraE project.

Each scenario operates independently with its own database and file structure while sharing the overall project configuration.

Scenarios are typically managed through the Project class rather than instantiated directly by users.

The root scenario is special-cased and represents the original project configuration. All other scenarios are stored in subdirectories and reference their own database files.

about: *About*

base_path: **Path**

logger: **Logger**

matrices: ***Matrices***

name: **str**

network: ***Network***

path_to_file: **Path**

results: ***Results***

transit: ***Transit***

zoning: ***Zoning***

aequilibrae.project.table_loader

Classes

TableLoader()

class aequilibrae.project.table_loader.**TableLoader**

load_structure(*conn: Connection*, *table_name: str*) → None

load_table(*conn: Connection*, *table_name: str*) → List[dict]

aequilibrae.project.tools

class aequilibrae.project.tools.**Migration**(*id: int*, *name: str*, *file: Path*, *type: str = None*)

Small utility class to wrap files used for database upgrades/migrations.

Individual migrations can report their status, be marked as 'seen' or as another status, and applied. SQL migrations are executed using `sqlite3.executescript`. Python migrations are loaded as a module, they should expose a `migrate` function which accepts an `sqlite3.Connection` as a single positional argument.

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.

Applying a migration will update the status to 'APPLIED' with the current timestamp.

A migration's status cannot be downgraded without force.

Migrations are identified based on their `id` attribute and the `id` field of the `migrations` table.

apply(*conn: Connection*, *connections: dict[str, Connection]*)

Apply this migration.

Successful application will mark the migration as `APPLIED`.

Python migrations should never use `executescript` as it will commit the pending transaction and place SQLite in autocommit mode. If the migration then fails the database will be bad state.

Arguments

conn (`sqlite3.Connection`): Main SQLite database connection. This is connection is used for the migrations table and '.sql' migrations.

connections (`dict[str, sqlite3.Connection]`): All open SQLite connections. Passed as keyword arguments for Python migrations.

mark_as(*conn: Connection*, *status:* MigrationStatus, *force: bool = False*)

Update or insert this migration with the given status.

If the migration is not present in the table it will be inserted. If it is present and the new status is a 'upgrade' or `force=True`, then it will be updated. Otherwise no change will be made.

Arguments

conn (`sqlite3.Connection`): SQLite database connection.

status (*`MigrationStatus`*): Migration status enum.

mark_as_seen(*conn: Connection*)

Mark this migration 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

conn (`sqlite3.Connection`): SQLite database connection.

status(*conn: Connection*) → *MigrationStatus*

Query the database for this migrations status.

If the `migrations` table is not present all migrations are considered `MISSING`.

Arguments

conn (`sqlite3.Connection`): SQLite database connection.

Returns

status (*`MigrationStatus`*): Migration status enum.

file: Path

id: int

name: str

type: str = None

class aequilibrae.project.tools.**MigrationManager**(*migration_file: Path*)

Small utility class to manage, validate, and apply a set of `Migration`s.

Arguments

migration_file (`pathlib.Path`): A path to a Python with which defines a global `migrations` variable as a list of `pathlib.Path` to migrations.

find_applicable(*conn: Connection*)

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

conn (`sqlite3.Connection`): SQLite database connection.

mark_all_as_seen(*conn: Connection*)

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

conn (`sqlite3.Connection`): SQLite database connection.

status(*conn: Connection*) → dict[int, *MigrationStatus*]

Query the database for all migrations' status.

If the `migrations` table is not present all migrations are considered `MISSING`.

Arguments

conn (`sqlite3.Connection`): SQLite database connection.

Returns

status (`dict[int, MigrationStatus]`): Migration status enums by their ID.

upgrade(*main_conn: str*, *connections: dict[str,* AequilibraEConnection | *None]*, *skip: set[int] = None*)

Find and apply all applicable migrations.

Optionally skip some migrations. Take care when skipping migrations.

Arguments

main_conn (`str`): Main SQLite database connection. This is connection is used for the migrations table. Must be a key in `connections`.

skip (`set[int]`): Set of migration IDs to skip.

connections (`dict[str, Optional[AequilibraEConnection]]`): Dictionary mapping connection names to *AequilibraEConnection* objects. These connections are used during the migration process.

migrations: dict[int, *Migration*]

network_migration_file = PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/project/database_specification/network/migrations/migrations.py')

transit_migration_file = PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/project/database_specification/transit/migrations/migrations.py')

class aequilibrae.project.tools.**MigrationStatus**(*value*)

classmethod from_bytes(*bytes*, *byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed

Indicates whether two's complement is used to represent the integer.

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

length
: Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
: The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
: Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

APPLIED: int = 3

MISSING: int = 1

SKIPPED: int = 2

denominator

the denominator of a rational number in lowest terms

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

class aequilibrae.project.tools.**NetworkSimplifier**(*project=None*)

collapse_links_into_nodes(*links: List[int]*)

Collapses links into nodes, adjusting the network in the neighborhood.

Arguments

links (`List[int]`): List containing link IDs to be collapsed.

rebuild_network()

Rebuilds the network elements that would have to be rebuilt after massive network simplification

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

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>

Modules

cli
migration_manager
network_simplifier

aequilibrae.project.tools.cli

Functions

add_subcommand_from_function(subparsers, ...)	Create a sub-command from a function's signature.
cli([argv])	Entry point for the aeq command.
list_functions(parser, args, unparsed_args)	List functions present in the run module.
run(args, unparsed_args)	Execute a function from the run module with argument parsing inferred from the function signature.

aequilibrae.project.tools.cli.**add_subcommand_from_function**(*subparsers*, *func*, *defaults: dict*)

Create a sub-command from a function's signature.

aequilibrae.project.tools.cli.**cli**(*argv=None*)

Entry point for the aeq command. argv defaults to sys.argv, it is a parameter for testing.

aequilibrae.project.tools.cli.**list_functions**(*parser*, *args*, *unparsed_args*)

List functions present in the run module.

aequilibrae.project.tools.cli.**run**(*args*, *unparsed_args*)

Execute a function from the run module with argument parsing inferred from the function signature.

aequilibrae.project.tools.migration_manager

Classes

Migration(id, name, file[, type])	Small utility class to wrap files used for database upgrades/migrations.
MigrationManager(migration_file)	Small utility class to manage, validate, and apply a set of `Migrations`.
MigrationStatus(value)	

class aequilibrae.project.tools.migration_manager.**Migration**(*id: int*, *name: str*, *file: Path*, *type: str = None*)

Small utility class to wrap files used for database upgrades/migrations.

Individual migrations can report their status, be marked as 'seen' or as another status, and applied. SQL migrations are executed using `sqlite3.executescript`. Python migrations are loaded as a module, they should expose a `migrate` function which accepts an `sqlite3.Connection` as a single positional argument.

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.

Applying a migration will update the status to 'APPLIED' with the current timestamp.

A migration's status cannot be downgraded without force.

Migrations are identified based on their `id` attribute and the `id` field of the `migrations` table.

apply(*conn: Connection*, *connections: dict[str, Connection]*)

Apply this migration.

Successful application will mark the migration as `APPLIED`.

Python migrations should never use `executescript` as it will commit the pending transaction and place SQLite in autocommit mode. If the migration then fails the database will be bad state.

Arguments

conn (`sqlite3.Connection`): Main SQLite database connection. This is connection is used for the migrations table and '.sql' migrations.

connections (`dict[str, sqlite3.Connection]`): All open SQLite connections. Passed as keyword arguments for Python migrations.

mark_as(*conn: Connection*, *status:* MigrationStatus, *force: bool = False*)

Update or insert this migration with the given status.

If the migration is not present in the table it will be inserted. If it is present and the new status is a 'upgrade' or `force=True`, then it will be updated. Otherwise no change will be made.

Arguments

conn (`sqlite3.Connection`): SQLite database connection.

status (*MigrationStatus*): Migration status enum.

mark_as_seen(*conn: Connection*)

Mark this migration 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

conn (`sqlite3.Connection`): SQLite database connection.

status(*conn: Connection*) → *MigrationStatus*

Query the database for this migrations status.

If the `migrations` table is not present all migrations are considered `MISSING`.

Arguments
conn (`sqlite3.Connection`): SQLite database connection.

Returns
status (*`MigrationStatus`*): Migration status enum.

file: Path

id: int

name: str

type: str = None

class aequilibrae.project.tools.migration_manager.**MigrationManager**(*migration_file: Path*)

Small utility class to manage, validate, and apply a set of `Migration`s.

Arguments
migration_file (`pathlib.Path`): A path to a Python with which defines a global `migrations` variable as a list of `pathlib.Path` to migrations.

find_applicable(*conn: Connection*)

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
conn (`sqlite3.Connection`): SQLite database connection.

mark_all_as_seen(*conn: Connection*)

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
conn (`sqlite3.Connection`): SQLite database connection.

status(*conn: Connection*) → dict[int, *MigrationStatus*]

Query the database for all migrations' status.

If the `migrations` table is not present all migrations are considered `MISSING`.

Arguments
conn (`sqlite3.Connection`): SQLite database connection.

Returns
status (`dict[int, MigrationStatus]`): Migration status enums by their ID.

upgrade(*main_conn: str*, *connections: dict[str,* AequilibraEConnection | *None]*, *skip: set[int] = None*)

Find and apply all applicable migrations.

Optionally skip some migrations. Take care when skipping migrations.

Arguments

main_conn (`str`): Main SQLite database connection. This is connection is used for the migrations table. Must be a key in `connections`.

skip (`set[int]`): Set of migration IDs to skip.

connections (`dict[str, Optional[AequilibraEConnection]]`): Dictionary mapping connection names to *AequilibraEConnection* objects. These connections are used during the migration process.

migrations: dict[int, *Migration*]

network_migration_file = PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/project/database_specification/network/migrations/migrations.py')

transit_migration_file = PosixPath('/home/runner/work/aequilibrae/aequilibrae/aequilibrae/project/database_specification/transit/migrations/migrations.py')

class aequilibrae.project.tools.migration_manager.**MigrationStatus**(*value*)

classmethod from_bytes(*bytes*, *byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

bytes

Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder

The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed

Indicates whether two's complement is used to represent the integer.

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

APPLIED: int = 3

MISSING: int = 1

SKIPPED: int = 2

denominator
the denominator of a rational number in lowest terms

imag
the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

aequilibrae.project.tools.network_simplifier

Classes

NetworkSimplifier([project])

class aequilibrae.project.tools.network_simplifier.**NetworkSimplifier**(*project=None*)

collapse_links_into_nodes(*links: List[int]*)

Collapses links into nodes, adjusting the network in the neighborhood.

Arguments

links (`List[int]`): List containing link IDs to be collapsed.

rebuild_network()

Rebuilds the network elements that would have to be rebuilt after massive network simplification

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

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>

aequilibrae.project.zone

Classes

Zone(dataset, zoning)	Single zone object that can be queried and manipulated in memory

class aequilibrae.project.zone.**Zone**(*dataset: dict*, *zoning*)

Single zone object that can be queried and manipulated in memory

add_centroid(*point: Point*, *robust=True*) → None

Adds a centroid to the network file

Arguments

point (`Point`): Shapely Point corresponding to the desired centroid position. If None, uses the geometric center of the zone

robust (`Bool`, *Optional*): Moves the centroid location around to avoid node conflict. Defaults to `True`.

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *conn: Connection | None = None*, *limit_to_zone=True*) → None

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting the zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the zone.

The selection of the nodes that will be connected is done simply by searching for the node closest to the zone centroid, or the N closest nodes to the centroid.

If fewer candidates than required connectors are found, all candidates are connected.

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered. eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

conn (`sqlite3.Connection`, *Optional*): Connection to the database.

limit_to_zone (`bool`): Limits the search for nodes inside the zone. Defaults to `True`.

delete()

Removes the zone from the database

disconnect_mode(*mode_id: str*) → None

Removes centroid connectors for the desired mode from the network file

Arguments

mode_id (`str`): Mode ID we are trying to disconnect from this zone

save()

Saves/Updates the zone data to the database

aequilibrae.project.zoning

Classes

Zoning(network)	Access to the API resources to manipulate the 'zones' table in the project

class aequilibrae.project.zoning.**Zoning**(*network*)

Access to the API resources to manipulate the 'zones' table in the project

```
>>> project = create_example(project_path)

>>> zoning = project.zoning

>>> zone_downtown = zoning.get(1)
>>> zone_downtown.population = 637
>>> zone_downtown.employment = 10039
>>> zone_downtown.save()

# We can also add one more field to the table
>>> fields = zoning.fields
>>> fields.add('parking_spots', 'Total licensed parking spots', 'INTEGER')

>>> project.close()
```

add_centroids(*robust=True*)

Adds automatic centroids to the network file. It adds centroids to all zones that do not have one Centroid is added to the geographic centroid of the zone.

Arguments

robust (`bool`, *Optional*): Moves the centroid location around to avoid node conflict. Defaults to `True`.

all_zones() → dict

Returns a dictionary with all Zone objects available in the model, using `zone_id` as key

connect_mode(*mode_id: str*, *link_types=''*, *connectors=1*, *limit_to_zone=True*, *bulk: bool = False*)

Adds centroid connectors for the desired mode to the network file

Centroid connectors are created by connecting each zone centroid to one or more nodes selected from all those that satisfy the mode and link_types criteria and are inside the zone.

The selection of the nodes that will be connected is done simply by searching for the node closest to each zone centroid, or the N closest nodes to the centroid.

If fewer candidates than required connectors are found, all candidates are connected.

CENTROIDS THAT ARE CURRENTLY CONNECTED ARE SKIPPED ALTOGETHER

Arguments

mode_id (`str`): Mode ID we are trying to connect

link_types (`str`, *Optional*): String with all the link type IDs that can be considered.
eg: yCdR. Defaults to ALL link types

connectors (`int`, *Optional*): Number of connectors to add. Defaults to 1

limit_to_zone (`bool`): Limits the search for nodes inside the zone. Defaults to `True`.

bulk (`bool`, *Optional*): Whether to use the bulk connector method or not. This is method is
considerably faster for connecting a large amount of centroids but has a high runtime overhead.

`coverage``()` → Polygon

Returns a single polygon for the entire zoning coverage

Returns

model coverage (`Polygon`): Shapely (Multi)polygon of the zoning system.

`create_zoning_layer``()`

Creates the 'zones' table for project files that did not previously contain it

`extent``()` → Polygon

Queries the extent of the layer included in the model

Returns

model extent (`Polygon`): Shapely polygon with the bounding box of the layer.

`get`(*zone_id: str*) → *Zone*

Get a zone from the model by its `zone_id`

`get_closest_zone`(*geometry: Point | LineString | MultiLineString*) → int

Returns the zone in which the given geometry is located.

If the geometry is not fully enclosed by any zone, the zone closest to the geometry is returned

Arguments

geometry (`Point` or `LineString`): A Shapely geometry object

Returns

zone_id (`int`): ID of the zone applicable to the point provided

`new`(*zone_id: int*) → *Zone*

Creates a new zone

Returns

zone (`Zone`): A new zone object populated only with `zone_id` (but not saved in the model yet)

`refresh_geo_index``()`

save()

property data: GeoDataFrame

Returns all zones data as a Pandas DataFrame

Returns

table (GeoDataFrame): GeoPandas GeoDataFrame with all the nodes

property fields: *FieldEditor*

Returns a FieldEditor class instance to edit the zones table fields and their metadata

11.1.8 aequilibrae.reference_files

11.1.9 aequilibrae.transit

class aequilibrae.transit.**Transit**(*project*)

build_pt_preload(*start: int*, *end: int*, *inclusion_cond: str = 'start'*) → DataFrame

Builds a preload vector for the transit network over the specified time period

Arguments

start (int): The start of the period for which to check pt schedules (seconds from midnight)

end (int): The end of the period for which to check pt schedules, (seconds from midnight)

inclusion_cond (str): Specifies condition with which to include/exclude pt trips from the preload.

Returns

preloads (pd.DataFrame): A DataFrame of preload from transit vehicles that can be directly used in an assignment

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

>>> project.network.build_graphs()

>>> start = int(6.5 * 60 * 60) # 6:30 am
>>> end = int(8.5 * 60 * 60)   # 8:30 am

>>> transit = Transit(project)
>>> preload = transit.build_pt_preload(start, end)

>>> project.close()
```

create_graph(***kwargs*) → *TransitGraphBuilder*

Create a transit graph from an existing GTFS import.

All arguments are forwarded to 'TransitGraphBuilder'.

A 'period_id' may be specified to select a time period. By default, a whole day is used. See 'project.network.Periods' for more details.

create_transit_database()

Creates the public transport database

get_table(*table_name*) → DataFrame

load(*period_ids: List[int] = None*)

Load the previously saved transit graphs from the 'public_transport.sqlite' database. Loading may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to load. Defaults to all available graph configurations.

new_gtfs_builder(*agency*, *file_path*, *day=''*, *description=''*) → *GTFSRouteSystemBuilder*

Returns a `GTFSRouteSystemBuilder` object compatible with the project

Arguments

agency (`str`): Name for the agency this feed refers to (e.g. 'CTA')

file_path (`str`): Full path to the GTFS feed (e.g. 'D:/project/my_gtfs_feed.zip')

day (`str`, *Optional*): Service data contained in this field to be imported (e.g. '2019-10-04')

description (`str`, *Optional*): Description for this feed (e.g. 'CTA2019 fixed by John Doe')

Returns

gtfs_feed (`StaticGTFS`): A GTFS feed that can be added to this network

remove_graphs(*period_ids: List[int]*, *unload: bool = False*)

Remove the previously saved transit graphs from the 'public_transport.sqlite' database. Removing may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to save. **unload** (`bool`): Also unload the graph.

save_graphs(*period_ids: List[int] = None*, *force: bool = False*)

Save the previously build transit graphs to the 'public_transport.sqlite' database. Saving may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to save. Defaults to 'project.network.periods.default_period.period_id'.

force (`bool`): Remove the existing graphs before saving the 'period_ids' graphs. Default 'False'.

default_capacities = {'other': [30, 60], 0: [150, 300], 1: [280, 560], 11: [30, 60], 12: [50, 100], 2: [700, 700], 3: [30, 60], 4: [400, 800], 5: [20, 40]}

default_pces = {'other': 2.0, 0: 5.0, 1: 5.0, 11: 3.0, 3: 4.0, 5: 4.0}

graphs: Dict[str, *TransitGraph*] = {}

pt_con: Connection

transit = <aequilibrae.utils.python_signal.PythonSignal object>

class aequilibrae.transit.**TransitGraphBuilder**(*project*, *period_id: int = 1*, *time_margin: int = 0*, *projected_crs: str = 'EPSG:3857'*, *num_threads: int = -1*, *seed: int = None*, *geometry_noise: bool = None*, *noise_coef: float = None*, *with_inner_stop_transfers: bool = False*, *with_outer_stop_transfers: bool = False*, *with_walking_edges: bool = True*, *distance_upper_bound: float = inf*, *blocking_centroid_flows: bool = True*, *connector_method: str = 'nearest_neighbour'*, *max_connectors_per_zone: int = -1*)

Graph builder for the transit assignment Spiess & Florian algorithm.

Arguments

public_transport_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): Period id for the period to be used. Preferred over start and end.

time_margin (`int`): Time margin, extends the start and end times by *time_margin* ([start, end] becomes [start - time_margin, end + time_margin]), in order to include more trips when computing mean values. Defaults to `0`.

projected_crs (`str`): Projected CRS of the network, intended for more accurate distance calculations. Defaults to `EPSG:3857`, Spherical Mercator.

num_threads (`int`): Number of threads to be used where possible. Defaults to `-1` (using all available threads).

seed (`int`): Deprecated. No longer in use.

geometry_noise (`bool`): Deprecated. No longer in use.

noise_coef (`float`): Deprecated. No longer in use.

with_inner_stop_transfers (`bool`): Whether to create transfer edges within parent stations. Defaults to `False`.

with_outer_stop_transfers (`bool`): Whether to create transfer edges between parent stations. Defaults to `False`.

with_walking_edges (`bool`): Whether to create walking edges between distinct stops of each station. Defaults to `True`.

distance_upper_bound (`float`): Upper bound on connector distance. Defaults to `np.inf`.

blocking_centroid_flows (`bool`): Whether to block flow through centroids. Defaults to `True`.

max_connectors_per_zone (`int`): Maximum connectors per zone. Defaults to `-1` for unlimited.

classmethod from_db(*project*, *period_id: int*, ***kwargs*)

Create a SF graph instance from an existing database save.

Assumes the database was constructed with the provided save methods. No checks are performed to see if the provided arguments are compatible with the saved graph.

All arguments are forwarded to the constructor.

Arguments

project (`Project`): AequilbraE project to use.

period_id (`int`): Period ID to use.

classmethod remove(*pt_conn: Connection*, *project_conn: Connection*, *period_id: int*)

static remove_config(*conn: Connection*, *period_id: int*)

Remove a transit graph configuration from the project database specified by it's *period_id*.

Arguments

conn (`sqlite3.Connection`): Connection to the `project.sqlite` database.

period_id (`int`): *period_id* key for the *transit_graph_configs* table.

static remove_edges(*pt_conn: Connection*, *period_id: int*)

Remove a transit graph's edges from the public transport database specified by it's *period_id*.

Arguments

pt_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): *period_id* to remove.

static remove_vertices(*pt_conn: Connection*, *period_id: int*)

Remove a transit graph's vertices from the public transport database specified by it's *period_id*.

Arguments

pt_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): `period_id` to remove.

add_zones(*zones*, *from_crs: str = None*)

Add zones as ODs.

Arguments

zones (`pd.DataFrame`): DataFrame containing the zoning information. Columns must include *zone_id* and *geometry*.

from_crs (`str`): The CRS of the *geometry* column of *zones*. If not provided it's assumed that the geometry is already in `self.projected_crs`. If provided, the geometry will be projected to `self.projected_crs`. Defaults to `None`.

convert_demand_matrix_from_zone_to_node_ids(*demand_matrix*, *o_zone_col='origin_zone_id'*, *d_zone_col='destination_zone'*, *demand_col='demand'*)

Convert a sparse demand matrix from `zone_id`'s to the corresponding `node_id`'s.

create_additional_db_fields(*conn: Connection = None*)

Create the additional required entries in the tables.

Arguments

pt_conn (`sqlite.Connection`): Optional PT connection to use

create_graph()

Create the SF transit graph (vertices and edges).

create_line_geometry(*method='direct'*, *graph='w'*)

Create the LineString for each edge.

The direct method creates a straight line between all points.

The connect project match method uses the existing line geometry within the project to create more accurate line strings. It creates a line string that matches the path between the shortest path between the project nodes closest to either end of the access and egress connectors.

Project graphs must be built for the "connector project match" method.

Arguments

method (`str`): Must be either *direct* or *connector project match*. If method is *direct*, *graph* argument is ignored.

graph (`str`): Must be a key within `project.network.graphs`.

create_od_node_mapping()

Build a dataframe mapping the centroid node ids with to transport assignment zone ids.

save(*robust=True*, *pt_conn: Connection = None*, *project_conn: Connection = None*)

Save the current graph to the public transport database.

Arguments

robust (`bool`): Deprecated. No longer in use.

pt_conn (`sqlite.Connection`): Optional PT connection to use

project_conn (`sqlite.Connection`): Optional project connection to use

save_config(*conn: Connection = None*)

save_edges(*recreate_line_geometry=False*, *conn: Connection = None*)

Save the contents of self.edges to the public transport database.

If no geometry for the edges is present or *recreate_line_geometry* is `True`, direct lines will be created.

Arguments

recreate_line_geometry (`bool`): Whether to recreate the line strings for the edges as direct lines. Defaults to `False`.

pt_conn (`sqlite.Connection`): Optional PT connection to use

save_vertices(*robust=None*, *conn: Connection = None*)

Write the vertices DataFrame to the public transport database.

Within the database nodes may not exist at the exact same point in space, provide `robust=True` to move the nodes slightly.

Arguments

robust (`bool`): Deprecated. No longer in use.

pt_conn (`sqlite.Connection`): Optional PT connection to use

to_transit_graph() → *TransitGraph*

Create an AequilibraE `TransitGraph` object from an SF graph builder.

property config

Modules

column_order
constants
date_tools
functions
gtfs_loader
gtfs_writer
lib_gtfs
parse_csv
route_map_matcher
route_system
route_system_reader
transit
transit_elements

continues on next page

Table 82 – continued from previous page

transit_graph_builder	Create the graph used by public transport assignment algorithms.

aequilibrae.transit.column_order

aequilibrae.transit.constants

Classes

Constants()

class aequilibrae.transit.constants.**Constants**

agencies: Dict[str, Any] = {}

fares: Dict[int, int] = {}

links: Dict[int, int] = {}

pattern_lookup: Dict[int, int] = {}

patterns: Dict[int, int] = {}

routes: Dict[int, int] = {}

srid: Dict[int, int] = {}

stops: Dict[int, int] = {}

transit_links: Dict[int, int] = {}

trips: Dict[int, int] = {}

aequilibrae.transit.date_tools

Functions

create_days_between(range_start_date, ...)	
day_of_week(date)	
format_date(date)	
one_day_before(date_object)	
to_seconds(value)	
to_time_string(value)	

aequilibrae.transit.date_tools.**create_days_between**(*range_start_date*, *range_end_date*)

aequilibrae.transit.date_tools.**day_of_week**(*date: str*)

aequilibrae.transit.date_tools.**format_date**(*date: str*) → str

aequilibrae.transit.date_tools.**one_day_before**(*date_object*)

aequilibrae.transit.date_tools.**to_seconds**(*value: str | None*) → int | Any

aequilibrae.transit.date_tools.**to_time_string**(*value: int | None*) → str | None

aequilibrae.transit.functions

Modules

breaking_links_for_stop_access
compute_line_bearing
get_srid

aequilibrae.transit.functions.breaking_links_for_stop_access

Functions

split_links_at_stops(stops, links[, tolerance])	Breaks links at the closest points for the nodes (transit stops) nearby, within a certain tolerance.

aequilibrae.transit.functions.breaking_links_for_stop_access.**split_links_at_stops**(*stops: GeoDataFrame*, *links: GeoDataFrame*, *tolerance: float = 50*) → List[GeoDataFrame]

Breaks links at the closest points for the nodes (transit stops) nearby, within a certain tolerance. New nodes are created at split points with IDs continuing from the maximum of the existing *a_node* and *b_node* values.

Parameters

- **stops** – GeoDataFrame of points
- **links** – GeoDataFrame of linestrings. Must contain "a_node" and "b_node" columns.
- **tolerance** – Search radius.

Returns

A list containing two GeoDataFrames:

- broken_links: The provided links where some have been split at stop locations. Contains updated "a_node", "b_node", and "geometry". Original link attributes are preserved (duplicated for splits).
- new_nodes: Point geometries representing the new nodes created at split locations, with IDs starting from *start_node_id*.

Return type
List[GeoDataFrame]

aequilibrae.transit.functions.compute_line_bearing

Functions

compute_line_bearing(point_a, point_b)	Computes line bearing for projected (cartesian) coordinates.

aequilibrae.transit.functions.compute_line_bearing.**compute_line_bearing**(*point_a: tuple*, *point_b: tuple*) → float

Computes line bearing for projected (cartesian) coordinates. For non-projected coordinates, see: https://gist.github.com/jeromer/2005586

Arguments

point_a (`tuple`): first point coordinates (lat, lon) **point_b** (`tuple`): second point coordinates (lat, lon)

aequilibrae.transit.functions.get_srid

Functions

get_srid()	Get the project SRID.

aequilibrae.transit.functions.get_srid.**get_srid**()

Get the project SRID. Currently only supports 4326.

aequilibrae.transit.gtfs_loader

Classes

GTFSReader(conn)	

class aequilibrae.transit.gtfs_loader.**GTFSReader**(*conn*)

load_data(*service_date: str*)

Loads the data for a respective service date.

Arguments

service_date (`str`): service date. e.g. "2020-04-01".

set_feed_path(*file_path*)

Sets GTFS feed source to be used

Arguments

file_path (`str`): Full path to the GTFS feed (e.g. 'D:/project/my_gtfs_feed.zip')

signal = <aequilibrae.utils.python_signal.PythonSignal object>

Loader for GTFS data. Not meant to be used directly by the user

aequilibrae.transit.gtfs_writer

aequilibrae.transit.gtfs_writer.**write_agencies**(*agencies: List[Agency]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.**write_fares**(*folder_path: str*, *conn*)

aequilibrae.transit.gtfs_writer.**write_routes**(*routes: List[Route]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.**write_shapes**(*patterns: List[Pattern]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.**write_stop_times**(*stop_times: DataFrame*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.**write_stops**(*stops: List[Stop]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.**write_trips**(*trips: List[Trip]*, *folder_path: str*, *conn: Connection*)

Modules

agency_writer
fare_writer
routes_writer
shape_writer
stop_times_writer
stops_writer
trips_writer

aequilibrae.transit.gtfs_writer.agency_writer

Functions

write_agencies(agencies, folder_path)

aequilibrae.transit.gtfs_writer.agency_writer.**write_agencies**(*agencies: List[Agency]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.fare_writer

Functions

write_fares(folder_path, conn)

aequilibrae.transit.gtfs_writer.fare_writer.**write_fares**(*folder_path: str*, *conn*)

aequilibrae.transit.gtfs_writer.routes_writer

Functions

write_routes(routes, folder_path)

aequilibrae.transit.gtfs_writer.routes_writer.**write_routes**(*routes: List[Route]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.shape_writer

Functions

write_shapes(patterns, folder_path)	

aequilibrae.transit.gtfs_writer.shape_writer.**write_shapes**(*patterns: List[Pattern]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.stop_times_writer

Functions

write_stop_times(stop_times, folder_path)	

aequilibrae.transit.gtfs_writer.stop_times_writer.**write_stop_times**(*stop_times: DataFrame*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.stops_writer

Functions

write_stops(stops, folder_path)	

aequilibrae.transit.gtfs_writer.stops_writer.**write_stops**(*stops: List[Stop]*, *folder_path: str*)

aequilibrae.transit.gtfs_writer.trips_writer

Functions

write_trips(trips, folder_path, conn)	

aequilibrae.transit.gtfs_writer.trips_writer.**write_trips**(*trips: List[Trip]*, *folder_path: str*, *conn: Connection*)

aequilibrae.transit.lib_gtfs

Classes

GTFSRouteSystemBuilder(network, ...[, day, ...])	

class aequilibrae.transit.lib_gtfs.**GTFSRouteSystemBuilder**(*network*, *agency_identifier*, *file_path*, *day=''*, *description=''*, *capacities=None*, *pces=None*)

builds_map_matchers()

Build the graph for links for a certain mode while splitting the closest links at stops' projection

Arguments
mode_id (`int`): Mode ID for which we will build the graph for

dates_available() → list

Returns a list of all dates available for this feed.

Returns
feed dates (`list`): list of all dates available for this feed

doWork()

Alias for execute_import

execute_import()

load_date(*service_date: str*) → None

Loads the transit services available for *service_date*

Arguments
service_date (`str`): Service data contained in this field to be imported (e.g. '2019-10-04')

map_match(*route_types=(3,)*) → None

Performs map-matching for all routes of one or more types.

Defaults to map-matching Bus routes (type 3) only.

For a reference of route types, see the inputs for route_type here.

Arguments
route_types (`List[int]` or `Tuple[int]`): Default is `(3,)`, for bus only

save_to_disk()

Saves all transit elements built in memory to disk

set_agency_identifier(*agency_id: str*) → None

Adds agency ID to this GTFS for use on import.

Arguments
agency_id (`str`): ID for the agency this feed refers to (e.g. 'CTA')

set_allow_map_match(*allow=True*)

Changes behavior for finding transit-link shapes. Defaults to `True`.

Arguments
allow (`bool` *Optional*): If `True`, allows uses map-matching in search of precise transit_link shapes. If `False`, sets transit_link shapes equal to straight lines between stops. In the presence of GTFS raw shapes it has no effect.

set_capacities(*capacities: dict*)

Sets default capacities for modes/vehicles.

Arguments
capacities (`dict`): Dictionary with GTFS types as keys, each with a list of 3 items for values for capacities: seated and total i.e. -> "{0: [150, 300],…}"

set_date(*service_date: str*) → None

Sets the date for import without doing any of data processing, which is left for the importer

set_description(*description: str*) → None

Adds description to be added to the imported layers metadata

Arguments

description (str): Description for this feed
(e.g. 'CTA2019 fixed by John Doe after strong coffee')

set_feed(*feed_path: str*) → None

Sets GTFS feed source to be used.

Arguments

file_path (str): Full path to the GTFS feed (e.g. 'D:/project/my_gtfs_feed.zip')

set_maximum_speeds(*max_speeds: DataFrame*)

Sets the maximum speeds to be enforced at segments.

Arguments

max_speeds (pd.DataFrame): Requires 4 fields: mode, min_distance, max_distance, speed. Modes not covered in the data will not be touched and distance brackets not covered will receive the maximum speed, with a warning

set_pces(*pces: dict*)

Sets default passenger car equivalent (PCE) factor for each GTFS mode.

Arguments

pces (dict): Dictionary with GTFS types as keys and the corresponding PCE value i.e. -> "{0: 2.0,…}"

signal = <aequilibrae.utils.python_signal.PythonSignal object>

Container for GTFS feeds providing data retrieval for the importer

aequilibrae.transit.parse_csv

Functions

parse_csv(file_name[, column_order])

Classes

empty()

class aequilibrae.transit.parse_csv.**empty**

shape = [0]

aequilibrae.transit.parse_csv.**parse_csv**(*file_name: str*, *column_order=[]*)

aequilibrae.transit.route_map_matcher

Classes

RouteMapMatcher(link_gdf, nodes_gdf, stops_gdf)

class aequilibrae.transit.route_map_matcher.**RouteMapMatcher**(*link_gdf: GeoDataFrame*, *nodes_gdf: GeoDataFrame*, *stops_gdf: GeoDataFrame*, *distance_to_project=50*)

assemble_shape(*df: DataFrame*, *enforce_single_parts=True*) → LineString | MultiLineString

Assembles a LineString shape from the matched path links.

Parameters
df – DataFrame with 'link_id' and 'dir' columns from map_match_route()

Returns
The assembled route shape

Return type
LineString

initialize_graph()

Build the graph for links for a certain mode while splitting the closest links at stops' projection

Arguments
mode_id (`int`): Mode ID for which we will build the graph for

distance_to_project (`float`, *Optional*): Radius search for links to break at the stops. Defaults to 50m

map_match_route(*route_stops: GeoDataFrame*, *route_shape: LineString | None = None*, *pattern_id: str | None = None*)

aequilibrae.transit.route_system

Classes

RouteSystem(project)

class aequilibrae.transit.route_system.**RouteSystem**(*project*)

load_route_system()

write_GTFS(*path_to_folder: str*)

aequilibrae.transit.route_system_reader

aequilibrae.transit.route_system_reader.**read_agencies**(*conn: Connection*)

aequilibrae.transit.route_system_reader.**read_patterns**(*conn: Connection*, *transformer*)

aequilibrae.transit.route_system_reader.**read_routes**(*conn: Connection*)

aequilibrae.transit.route_system_reader.**read_stop_times**(*conn: Connection*)

aequilibrae.transit.route_system_reader.**read_stops**(*conn: Connection*, *transformer*)

aequilibrae.transit.route_system_reader.**read_trips**(*conn: Connection*)

Modules

agency_reader
pattern_reader
routes_reader
stop_reader
stop_times_reader
trips_reader

aequilibrae.transit.route_system_reader.agency_reader

Functions

read_agencies(conn)

aequilibrae.transit.route_system_reader.agency_reader.**read_agencies**(*conn: Connection*)

aequilibrae.transit.route_system_reader.pattern_reader

Functions

read_patterns(conn, transformer)

aequilibrae.transit.route_system_reader.pattern_reader.**read_patterns**(*conn: Connection*, *transformer*)

aequilibrae.transit.route_system_reader.routes_reader

Functions

read_routes(conn)

aequilibrae.transit.route_system_reader.routes_reader.**read_routes**(*conn: Connection*)

aequilibrae.transit.route_system_reader.stop_reader

Functions

read_stops(conn, transformer)

aequilibrae.transit.route_system_reader.stop_reader.**read_stops**(*conn: Connection*, *transformer*)

aequilibrae.transit.route_system_reader.stop_times_reader

Functions

read_stop_times(conn)

aequilibrae.transit.route_system_reader.stop_times_reader.**read_stop_times**(*conn: Connection*)

aequilibrae.transit.route_system_reader.trips_reader

Functions

read_trips(conn)

aequilibrae.transit.route_system_reader.trips_reader.**read_trips**(*conn: Connection*)

aequilibrae.transit.transit

Classes

Transit(project)

class aequilibrae.transit.transit.**Transit**(*project*)

build_pt_preload(*start: int*, *end: int*, *inclusion_cond: str = 'start'*) → DataFrame

Builds a preload vector for the transit network over the specified time period

Arguments

start (`int`): The start of the period for which to check pt schedules (seconds from midnight)

end (`int`): The end of the period for which to check pt schedules, (seconds from midnight)

inclusion_cond (`str`): Specifies condition with which to include/exclude pt trips from the preload.

Returns

preloads (`pd.DataFrame`): A DataFrame of preload from transit vehicles that can be directly used in an assignment

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

>>> project.network.build_graphs()

>>> start = int(6.5 * 60 * 60) # 6:30 am
>>> end = int(8.5 * 60 * 60)   # 8:30 am

>>> transit = Transit(project)
>>> preload = transit.build_pt_preload(start, end)

>>> project.close()
```

create_graph(***kwargs*) → *TransitGraphBuilder*

Create a transit graph from an existing GTFS import.

All arguments are forwarded to 'TransitGraphBuilder'.

A 'period_id' may be specified to select a time period. By default, a whole day is used. See 'project.network.Periods' for more details.

create_transit_database()

Creates the public transport database

get_table(*table_name*) → DataFrame

load(*period_ids: List[int] = None*)

Load the previously saved transit graphs from the 'public_transport.sqlite' database. Loading may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to load. Defaults to all available graph configurations.

new_gtfs_builder(*agency*, *file_path*, *day=''*, *description=''*) → *GTFSRouteSystemBuilder*

Returns a `GTFSRouteSystemBuilder` object compatible with the project

Arguments

agency (`str`): Name for the agency this feed refers to (e.g. 'CTA')

file_path (`str`): Full path to the GTFS feed (e.g. 'D:/project/my_gtfs_feed.zip')

day (`str`, *Optional*): Service data contained in this field to be imported (e.g. '2019-10-04')

description (`str`, *Optional*): Description for this feed (e.g. 'CTA2019 fixed by John Doe')

Returns

gtfs_feed (`StaticGTFS`): A GTFS feed that can be added to this network

remove_graphs(*period_ids: List[int]*, *unload: bool = False*)

Remove the previously saved transit graphs from the 'public_transport.sqlite' database. Removing may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to save. **unload** (`bool`): Also unload the graph.

save_graphs(*period_ids: List[int] = None*, *force: bool = False*)

Save the previously build transit graphs to the 'public_transport.sqlite' database. Saving may be filtered by 'period_id'.

Arguments

period_ids (`int`): List of periods of to save. Defaults to
'project.network.periods.default_period.period_id'.

force (`bool`): Remove the existing graphs before saving the 'period_ids' graphs. Default 'False'.

default_capacities = {'other': [30, 60], 0: [150, 300], 1: [280, 560], 11: [30, 60], 12: [50, 100], 2: [700, 700], 3: [30, 60], 4: [400, 800], 5: [20, 40]}

default_pces = {'other': 2.0, 0: 5.0, 1: 5.0, 11: 3.0, 3: 4.0, 5: 4.0}

graphs: Dict[str, *TransitGraph*] = {}

pt_con: Connection

transit = <aequilibrae.utils.python_signal.PythonSignal object>

aequilibrae.transit.transit_elements

class aequilibrae.transit.transit_elements.**Agency**(*conn: Connection*)

Transit Agency to load into the database

- agency_id (int): ID for the transit agency
- agency (str): Name of the transit agency
- feed_date (str): Date for the transit feed using in the import
- service_date (str): Date for the route services being imported
- description (str): Description of the feed

from_row(*data: Series*)

save_to_database(*conn: Connection*) → None

Saves route to the database

class aequilibrae.transit.transit_elements.**Fare**(*agency_id: int*)

Transit Fare

- fare_id (int): ID of the fare as in the network model
- fare (str): ID of the fare as in GTFS
- agency (str): Corresponding agency as inputted during import
- agency_id (int): ID of the corresponding agency as in the network model
- price (int): As in GTFS
- currency (str): As in GTFS
- payment_method (int): As in GTFS
- transfer (int): As in GTFS
- transfer_duration (int): As in GTFS

populate(*record: tuple*, *headers: list*) → None

Adds fare information.

save_to_database(*conn: Connection*) → None

Saves Fare attributes to the database

class aequilibrae.transit.transit_elements.**FareRule**

Transit Fare

- fare_id (int): Fare Id to which this rule applies
- fare (str): Name of the fare rule
- route (str): Route ID as in GTFS to which this fare rule applies
- route_id (int): Route ID as in network model to which this fare rule applies
- origin (text): Transit fare zone ID for origin
- destination (text): Transit fare zone ID for destination
- contains (str): As in GTFS
- agency_id (int): Agency ID as in the network model

populate(*record: tuple*, *headers: list*) → None

Adds fare information.

save_to_database(*conn: Connection*) → None

Saves Fare rules to the database

class aequilibrae.transit.transit_elements.**Link**(*srid*)

Transit link element.

- transit_link (int): ID of the transit link (updated when inserted in the database)
- from_stop (str): Origin of the transit connection
- to_stop (str): Destination of the transit connection
- pair (str): Identifier of the stop pair as FROM_ID##TO_ID. For identification only
- geo (LineString): Geometry of the transit link as direct connection between stops
- length (float): Link length measured directly from the geometry object
- type (int): Route type (mode) for this transit link
- srid (int): srid of our working database

build_geo(*from_point: Point*, *to_point: Point*, *gtfs_shape: LineString*, *previous_end: Point*)

Builds link geometry.

get_link_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves Transit link to the database

class aequilibrae.transit.transit_elements.**Pattern**(*route_id*, *gtfs_feed*)

Represents a stop pattern for a particular route, as defined in GTFS.

best_shape() → LineString

Gets the best version of shape available for this pattern

from_row(*data: Series*)

map_match()

Map matches the route into the network, considering its appropriate shape.

Part of the map-matching process is to find the network links corresponding the pattern's raw shape, so that method will be called in case it has not been called before.

The basic algorithm behind the map-matching algorithm is described in https://doi.org/10.3141%2F2646-08

In a nutshell, we compute the shortest path between the nodes corresponding to the links to which stops were geographically matched, for each pair of identified links.

We do not consider links that are in perfect sequence, as we found that it introduces severe issues when stops are close to intersections without clear upsides.

When issues are found, we remove the stops in the immediate vicinity of the issue and attempt new path finding. The First and last stops/corresponding links are always kept.

If an error was found, (record for it will show in the log), it is stored within the object.

save_to_database(*conn: Connection*, *commit=True*) → None

Saves the pattern to the routes table

class aequilibrae.transit.transit_elements.**Route**(*agency_id*)

Transit route element to feed into Transit_routes

- route_id (str): ID of this route, starting with the agency prefix ID
- route_short_name (str): Short name as found in the GTFS feed
- route_long_name (str): Long name as found in the GTFS feed
- route_desc (str): Route description as found in the GTFS feed
- route_type (int): Route type (mode) for this transit link
- route_url (str): Route URL as found in the GTFS feed
- route_color (str): Route color for mapping as found in the GTFS feed
- route_text_color (str): Route color (text) for mapping as found in the GTFS feed
- route_sort_order (int): Route rendering order as found in the GTFS feed
- agency_id (str): Agency ID
- pce (float): Vehicle PCE for this route
- seated_capacity (float): Vehicle seated capacity for this route
- total_capacity (float): Total vehicle capacity for this route

from_row(*data: Series*)

populate(*record: tuple*, *headers: list*) → None

save_to_database(*conn: Connection*, *commit=True*) → None

Saves route to the database

property data

class aequilibrae.transit.transit_elements.**Service**

Transit service built with data from calendar.txt and calendar_dates.txt from GTFS

- service_id (str):
- monday (int): Flag if the route runs on mondays (1 for **True**, 0 for **False**)
- tuesday (int): Flag if the route runs on tuesdays (1 for **True**, 0 for **False**)
- wednesday (int): Flag if the route runs on wednesdays (1 for **True**, 0 for **False**)
- thursday (int): Flag if the route runs on thursdays (1 for **True**, 0 for **False**)
- friday (int): Flag if the route runs on fridays (1 for **True**, 0 for **False**)
- saturday (int): Flag if the route runs on saturdays (1 for **True**, 0 for **False**)
- sunday (int): Flag if the route runs on sundays (1 for **True**, 0 for **False**)
- start_date (str): Start date for this service
- end_date (str): End date for this service
- dates (List[str]): List of all dates for which this service is active between its start and end dates

class aequilibrae.transit.transit_elements.**Stop**(*agency_id: int*, *record: tuple*, *headers: list*)

Transit stop as read from the GTFS feed

from_row(*data: Series*)

get_node_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves Transit Stop to the database

property data: list

class aequilibrae.transit.transit_elements.**Trip**

Transit trips read from trips.txt

- trip (str): Trip ID as read from the GTFS feed
- route (str): Route ID as read from the GTFS feed
- service_id (str): Service ID as read from the GTFS feed
- trip_headsign (str): Trip headsign as read from the GTFS feed
- trip_short_name (str): Trip short name as read from the GTFS feed
- direction_id (int): Direction ID as read from the GTFS feed
- block_id (int): Block ID as read from the GTFS feed
- bikes_allowed (int): Bikes allowed flag as read from the GTFS feed
- wheelchair_accessible (int): Wheelchair accessibility flag as read from the GTFS feed
- shape_id (str): Shape ID as read from the GTFS feed
- trip_id (int): Unique trip_id as it will go into the database
- route_id (int): Unique Route ID as will be available in the routes table
- pattern_id (int): Unique Pattern ID for this route/stop-pattern as it will go into the database
- pattern_hash (str): Pattern ID derived from stops for this route/stop-pattern
- arrivals (List[int]): Sequence of arrival at stops for this trip
- departures (List[int]): Sequence of departures from stops for this trip
- stops (List[Stop]): Sequence of stops for this trip
- shape (LineString): Shape for this trip. Directly from shapes.txt or rebuilt from sequence of stops

from_row(*data: Series*)

get_trip_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves trips to the database

Modules

agency
basic_element
fare
fare_rule
link
mode_correspondence
pattern

continues on next page

Table 111 – continued from previous page

route
service
stop
trip

aequilibrae.transit.transit_elements.agency

Classes

Agency(conn)	Transit Agency to load into the database

class aequilibrae.transit.transit_elements.agency.**Agency**(*conn: Connection*)

Transit Agency to load into the database

- agency_id (int): ID for the transit agency
- agency (str): Name of the transit agency
- feed_date (str): Date for the transit feed using in the import
- service_date (str): Date for the route services being imported
- description (str): Description of the feed

from_row(*data: Series*)

save_to_database(*conn: Connection*) → None

Saves route to the database

aequilibrae.transit.transit_elements.basic_element

Classes

BasicPTElement()	

class aequilibrae.transit.transit_elements.basic_element.**BasicPTElement**

from_row(*data: Series*)

aequilibrae.transit.transit_elements.fare

Classes

Fare(agency_id)	Transit Fare

class aequilibrae.transit.transit_elements.fare.**Fare**(*agency_id: int*)

Transit Fare

- fare_id (int): ID of the fare as in the network model
- fare (str): ID of the fare as in GTFS
- agency (str): Corresponding agency as inputted during import

- agency_id (int): ID of the corresponding agency as in the network model
- price (int): As in GTFS
- currency (str): As in GTFS
- payment_method (int): As in GTFS
- transfer (int): As in GTFS
- transfer_duration (int): As in GTFS

populate(*record: tuple*, *headers: list*) → None
Adds fare information.

save_to_database(*conn: Connection*) → None
Saves Fare attributes to the database

aequilibrae.transit.transit_elements.fare_rule

Classes

FareRule()	Transit Fare

class aequilibrae.transit.transit_elements.fare_rule.**FareRule**
Transit Fare

- fare_id (int): Fare Id to which this rule applies
- fare (str): Name of the fare rule
- route (str): Route ID as in GTFS to which this fare rule applies
- route_id (int): Route ID as in network model to which this fare rule applies
- origin (text): Transit fare zone ID for origin
- destination (text): Transit fare zone ID for destination
- contains (str): As in GTFS
- agency_id (int): Agency ID as in the network model

populate(*record: tuple*, *headers: list*) → None
Adds fare information.

save_to_database(*conn: Connection*) → None
Saves Fare rules to the database

aequilibrae.transit.transit_elements.link

Classes

Link(srid)	Transit link element.

class aequilibrae.transit.transit_elements.link.**Link**(*srid*)
Transit link element.

- transit_link (int): ID of the transit link (updated when inserted in the database)

- from_stop (`str`): Origin of the transit connection
- to_stop (`str`): Destination of the transit connection
- pair (`str`): Identifier of the stop pair as FROM_ID##TO_ID. For identification only
- geo (`LineString`): Geometry of the transit link as direct connection between stops
- length (`float`): Link length measured directly from the geometry object
- type (`int`): Route type (mode) for this transit link
- srid (`int`): srid of our working database

build_geo(*from_point: Point*, *to_point: Point*, *gtfs_shape: LineString*, *previous_end: Point*)

Builds link geometry.

get_link_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves Transit link to the database

aequilibrae.transit.transit_elements.mode_correspondence

aequilibrae.transit.transit_elements.pattern

Classes

Pattern(route_id, gtfs_feed)	Represents a stop pattern for a particular route, as defined in GTFS.

class aequilibrae.transit.transit_elements.pattern.**Pattern**(*route_id*, *gtfs_feed*)

Represents a stop pattern for a particular route, as defined in GTFS.

best_shape() → LineString

Gets the best version of shape available for this pattern

from_row(*data: Series*)

map_match()

Map matches the route into the network, considering its appropriate shape.

Part of the map-matching process is to find the network links corresponding the pattern's raw shape, so that method will be called in case it has not been called before.

The basic algorithm behind the map-matching algorithm is described in https://doi.org/10.3141%2F2646-08

In a nutshell, we compute the shortest path between the nodes corresponding to the links to which stops were geographically matched, for each pair of identified links.

We do not consider links that are in perfect sequence, as we found that it introduces severe issues when stops are close to intersections without clear upsides.

When issues are found, we remove the stops in the immediate vicinity of the issue and attempt new path finding. The First and last stops/corresponding links are always kept.

If an error was found, (record for it will show in the log), it is stored within the object.

save_to_database(*conn: Connection*, *commit=True*) → None

Saves the pattern to the routes table

aequilibrae.transit.transit_elements.route

Classes

Route(agency_id)	Transit route element to feed into Transit_routes

class aequilibrae.transit.transit_elements.route.**Route**(*agency_id*)

Transit route element to feed into Transit_routes

- route_id (str): ID of this route, starting with the agency prefix ID
- route_short_name (str): Short name as found in the GTFS feed
- route_long_name (str): Long name as found in the GTFS feed
- route_desc (str): Route description as found in the GTFS feed
- route_type (int): Route type (mode) for this transit link
- route_url (str): Route URL as found in the GTFS feed
- route_color (str): Route color for mapping as found in the GTFS feed
- route_text_color (str): Route color (text) for mapping as found in the GTFS feed
- route_sort_order (int): Route rendering order as found in the GTFS feed
- agency_id (str): Agency ID
- pce (float): Vehicle PCE for this route
- seated_capacity (float): Vehicle seated capacity for this route
- total_capacity (float): Total vehicle capacity for this route

from_row(*data: Series*)

populate(*record: tuple*, *headers: list*) → None

save_to_database(*conn: Connection*, *commit=True*) → None

Saves route to the database

property data

aequilibrae.transit.transit_elements.service

Classes

Service()	Transit service built with data from calendar.txt and calendar_dates.txt from GTFS

class aequilibrae.transit.transit_elements.service.**Service**

Transit service built with data from calendar.txt and calendar_dates.txt from GTFS

- service_id (str):
- monday (int): Flag if the route runs on mondays (1 for **True**, 0 for **False**)
- tuesday (int): Flag if the route runs on tuesdays (1 for **True**, 0 for **False**)
- wednesday (int): Flag if the route runs on wednesdays (1 for **True**, 0 for **False**)

- thursday (int): Flag if the route runs on thursdays (1 for **True**, 0 for **False**)
- friday (int): Flag if the route runs on fridays (1 for **True**, 0 for **False**)
- saturday (int): Flag if the route runs on saturdays (1 for **True**, 0 for **False**)
- sunday (int): Flag if the route runs on sundays (1 for **True**, 0 for **False**)
- start_date (str): Start date for this service
- end_date (str): End date for this service
- dates (List[str]): List of all dates for which this service is active between its start and end dates

aequilibrae.transit.transit_elements.stop

Classes

Stop(agency_id, record, headers)	Transit stop as read from the GTFS feed

class aequilibrae.transit.transit_elements.stop.**Stop**(*agency_id: int*, *record: tuple*, *headers: list*)

Transit stop as read from the GTFS feed

from_row(*data: Series*)

get_node_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves Transit Stop to the database

property data: list

aequilibrae.transit.transit_elements.trip

Classes

Trip()	Transit trips read from trips.txt

class aequilibrae.transit.transit_elements.trip.**Trip**

Transit trips read from trips.txt

- trip (str): Trip ID as read from the GTFS feed
- route (str): Route ID as read from the GTFS feed
- service_id (str): Service ID as read from the GTFS feed
- trip_headsign (str): Trip headsign as read from the GTFS feed
- trip_short_name (str): Trip short name as read from the GTFS feed
- direction_id (int): Direction ID as read from the GTFS feed
- block_id (int): Block ID as read from the GTFS feed
- bikes_allowed (int): Bikes allowed flag as read from the GTFS feed
- wheelchair_accessible (int): Wheelchair accessibility flag as read from the GTFS feed
- shape_id (str): Shape ID as read from the GTFS feed

- trip_id (`int`): Unique trip_id as it will go into the database
- route_id (`int`): Unique Route ID as will be available in the routes table
- pattern_id (`int`): Unique Pattern ID for this route/stop-pattern as it will go into the database
- pattern_hash (`str`): Pattern ID derived from stops for this route/stop-pattern
- arrivals (`List[int]`): Sequence of arrival at stops for this trip
- departures (`List[int]`): Sequence of departures from stops for this trip
- stops (`List[Stop]`): Sequence of stops for this trip
- shape (`LineString`): Shape for this trip. Directly from shapes.txt or rebuilt from sequence of stops

from_row(*data: Series*)

get_trip_id()

save_to_database(*conn: Connection*, *commit=True*) → None

Saves trips to the database

aequilibrae.transit.transit_graph_builder

Create the graph used by public transport assignment algorithms.

Naming Conventions: - a_node/b_node is head/tail vertex

TransitGraphBuilder Assumptions: - opposite directions are not supported. In the GTFS files, this corresponds to direction_id

from trips.txt (indicates the direction of travel for a trip),

- all times are expressed in seconds [s], all frequencies in [1/s], and
- headways are uniform for trips of the same pattern.

Classes

TransitGraphBuilder(project[, period_id, ...])	Graph builder for the transit assignment Spiess & Florian algorithm.

class aequilibrae.transit.transit_graph_builder.**TransitGraphBuilder**(*project*, *period_id: int = 1*, *time_margin: int = 0*, *projected_crs: str = 'EPSG:3857'*, *num_threads: int = -1*, *seed: int = None*, *geometry_noise: bool = None*, *noise_coef: float = None*, *with_inner_stop_transfers: bool = False*, *with_outer_stop_transfers: bool = False*, *with_walking_edges: bool = True*, *distance_upper_bound: float = inf*, *blocking_centroid_flows: bool = True*, *connector_method: str = 'nearest_neighbour'*, *max_connectors_per_zone: int = -1*)

Graph builder for the transit assignment Spiess & Florian algorithm.

Arguments

public_transport_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): Period id for the period to be used. Preferred over start and end.

time_margin (`int`): Time margin, extends the start and end times by *time_margin* ([start, end] becomes [start - time_margin, end + time_margin]), in order to include more trips when computing mean values. Defaults to `0`.

projected_crs (`str`): Projected CRS of the network, intended for more accurate distance calculations. Defaults to `EPSG:3857`, Spherical Mercator.

num_threads (`int`): Number of threads to be used where possible. Defaults to `-1` (using all available threads).

seed (`int`): Deprecated. No longer in use.

geometry_noise (`bool`): Deprecated. No longer in use.

noise_coef (`float`): Deprecated. No longer in use.

with_inner_stop_transfers (`bool`): Whether to create transfer edges within parent stations. Defaults to `False`.

with_outer_stop_transfers (`bool`): Whether to create transfer edges between parent stations. Defaults to `False`.

with_walking_edges (`bool`): Whether to create walking edges between distinct stops of each station. Defaults to `True`.

distance_upper_bound (`float`): Upper bound on connector distance. Defaults to `np.inf`.

blocking_centroid_flows (`bool`): Whether to block flow through centroids. Defaults to `True`.

max_connectors_per_zone (`int`): Maximum connectors per zone. Defaults to `-1` for unlimited.

classmethod from_db(*project*, *period_id: int*, ***kwargs*)

Create a SF graph instance from an existing database save.

Assumes the database was constructed with the provided save methods. No checks are performed to see if the provided arguments are compatible with the saved graph.

All arguments are forwarded to the constructor.

Arguments

project (`Project`): AequilbraE project to use.

period_id (`int`): Period ID to use.

classmethod remove(*pt_conn: Connection*, *project_conn: Connection*, *period_id: int*)

static remove_config(*conn: Connection*, *period_id: int*)

Remove a transit graph configuration from the project database specified by it's *period_id*.

Arguments

conn (`sqlite3.Connection`): Connection to the `project.sqlite` database.

period_id (`int`): *period_id* key for the *transit_graph_configs* table.

static remove_edges(*pt_conn: Connection*, *period_id: int*)

Remove a transit graph's edges from the public transport database specified by it's *period_id*.

Arguments

pt_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): *period_id* to remove.

static remove_vertices(*pt_conn: Connection*, *period_id: int*)

Remove a transit graph's vertices from the public transport database specified by it's *period_id*.

Arguments

pt_conn (`sqlite3.Connection`): Connection to the `public_transport.sqlite` database.

period_id (`int`): `period_id` to remove.

add_zones(*zones*, *from_crs: str = None*)

Add zones as ODs.

Arguments

zones (`pd.DataFrame`): DataFrame containing the zoning information. Columns must include *zone_id* and *geometry*.

from_crs (`str`): The CRS of the *geometry* column of *zones*. If not provided it's assumed that the geometry is already in `self.projected_crs`. If provided, the geometry will be projected to `self.projected_crs`. Defaults to `None`.

convert_demand_matrix_from_zone_to_node_ids(*demand_matrix*, *o_zone_col='origin_zone_id'*, *d_zone_col='destination_zone'*, *demand_col='demand'*)

Convert a sparse demand matrix from `zone_id`'s to the corresponding `node_id`'s.

create_additional_db_fields(*conn: Connection = None*)

Create the additional required entries in the tables.

Arguments

pt_conn (`sqlite.Connection`): Optional PT connection to use

create_graph()

Create the SF transit graph (vertices and edges).

create_line_geometry(*method='direct'*, *graph='w'*)

Create the LineString for each edge.

The direct method creates a straight line between all points.

The connect project match method uses the existing line geometry within the project to create more accurate line strings. It creates a line string that matches the path between the shortest path between the project nodes closest to either end of the access and egress connectors.

Project graphs must be built for the "connector project match" method.

Arguments

method (`str`): Must be either *direct* or *connector project match*. If method is *direct*, *graph* argument is ignored.

graph (`str`): Must be a key within `project.network.graphs`.

create_od_node_mapping()

Build a dataframe mapping the centroid node ids with to transport assignment zone ids.

save(*robust=True*, *pt_conn: Connection = None*, *project_conn: Connection = None*)

Save the current graph to the public transport database.

Arguments

robust (`bool`): Deprecated. No longer in use.

pt_conn (`sqlite.Connection`): Optional PT connection to use

project_conn (`sqlite.Connection`): Optional project connection to use

save_config(*conn: Connection = None*)

save_edges(*recreate_line_geometry=False*, *conn: Connection = None*)

Save the contents of self.edges to the public transport database.

If no geometry for the edges is present or *recreate_line_geometry* is `True`, direct lines will be created.

Arguments

recreate_line_geometry (`bool`): Whether to recreate the line strings for the edges as direct lines. Defaults to `False`.

pt_conn (`sqlite.Connection`): Optional PT connection to use

save_vertices(*robust=None*, *conn: Connection = None*)

Write the vertices DataFrame to the public transport database.

Within the database nodes may not exist at the exact same point in space, provide `robust=True` to move the nodes slightly.

Arguments

robust (`bool`): Deprecated. No longer in use.

pt_conn (`sqlite.Connection`): Optional PT connection to use

to_transit_graph() → *TransitGraph*

Create an AequilibraE `TransitGraph` object from an SF graph builder.

property config

11.1.10 aequilibrae.utils

Modules

aeq_signal	
core_setter	
create_delaunay_network	
create_example	
db_utils	
find_table_fields	
geo_index	
geo_utils	Convenience functions for working with geospatial data.
get_table	
interface	
list_tables_in_db	
model_run_utils	
python_signal	
qgis_utils	
simwrapper	
spatialite_utils	

aequilibrae.utils.aeq_signal

Functions

noop(_)	

Classes

simple_progress(thing, signal[, msg])	A *tqdm* style iterable wrapper using aequilibrae signals

class aequilibrae.utils.aeq_signal.**simple_progress**(*thing*, *signal*, *msg=None*)

A *tqdm* style iterable wrapper using aequilibrae signals

aequilibrae.utils.aeq_signal.**noop**(_)

aequilibrae.utils.core_setter

Functions

set_cores(cores_count)	

aequilibrae.utils.core_setter.**set_cores**(*cores_count: int*)

aequilibrae.utils.create_delaunay_network

Classes

DelaunayAnalysis(project)	

class aequilibrae.utils.create_delaunay_network.**DelaunayAnalysis**(*project*)

assign_matrix(*matrix:* AequilibraeMatrix, *result_name: str*)

create_network(*source='zones'*, *overwrite=False*)

Creates a delaunay network based on the existing model

Arguments

source (`str`, *Optional*): Source of the centroids/zones. Either `zones` or `network`. Default `zones`

overwrite path (`bool`, *Optional*): Whether to should overwrite an existing Delaunay Network. Default `False`

aequilibrae.utils.create_example

Functions

create_example(path[, from_model])	Copies an example model to a new project project and returns the project handle
list_examples()	

aequilibrae.utils.create_example.**create_example**(*path: PathLike*, *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.utils.create_example.**list_examples**() → List[str]

aequilibrae.utils.db_utils

Functions

add_column(conn, table_name, col_name, col_type)	
add_column_unless_exists(conn, table_name, ...)	
get_schema(conn, table_name)	
has_column(conn, table_name, col_name)	
has_table(conn, table_name)	
list_columns(conn, table_name)	
list_tables_in_db(conn)	
read_and_close(filepath[, spatial])	A context manager for sqlite connections (alias for *commit_and_close(db,commit=False)*).
read_sql(sql, filepath, **kwargs)	
safe_connect(filepath[, missing_ok])	

Classes

AequilibraEConnection(*args, **kwargs)	This custom factory class intends to solve the issue of premature commits when trying to use manual transaction control.
ColumnDef(idx, name, type, not_null, ...)	
commit_and_close(db[, commit, missing_ok, ...])	A context manager for sqlite connections which closes and commits.

class aequilibrae.utils.db_utils.**AequilibraEConnection**(**args*, ***kwargs*)

This custom factory class intends to solve the issue of premature commits when trying to use manual transaction control.

After `manual_transaction` is called, context manager enters and exits are tracked via their depth, the `sqlite3.Connection` is placed into manual transaction control and a transaction is started. If another transaction is already in progress an RuntimeError is raised. When exiting with depth == 0, the normal context manager enter and exit is called.

backup(*target*, ***, *pages=-1*, *progress=None*, *name='main'*, *sleep=0.25*)

Makes a backup of the database.

blobopen(*table*, *column*, *row*, */ (Positional-only parameter separator (PEP 570))*, ***, *readonly=False*, *name='main'*)

Open and return a BLOB object.

table
Table name.

column
Column name.

row
Row index.

readonly
Open the BLOB without write permissions.

name
Database name.

close()

Close the database connection.

Any pending transaction is not committed implicitly.

commit()

Commit any pending transaction to the database.

If there is no open transaction, this method is a no-op.

create_aggregate(*name*, *n_arg*, *aggregate_class*)

Creates a new aggregate.

create_collation(*name*, *callback*, */*)

Creates a collation function.

create_function(*name*, *narg*, *func*, ***, *deterministic=False*)

Creates a new function.

create_window_function(*name*, *num_params*, *aggregate_class*, /)

Creates or redefines an aggregate window function. Non-standard.

name
The name of the SQL aggregate window function to be created or redefined.

num_params
The number of arguments the step and inverse methods takes.

aggregate_class
A class with step(), finalize(), value(), and inverse() methods. Set to None to clear the window function.

cursor()

Return a cursor for the connection.

deserialize(*data*, /, *, *name='main'*)

Load a serialized database.

data
The serialized database content.

name
Which database to reopen with the deserialization.

The deserialize interface causes the database connection to disconnect from the target database, and then reopen it as an in-memory database based on the given serialized data.

The deserialize interface will fail with SQLITE_BUSY if the database is currently in a read transaction or is involved in a backup operation.

enable_load_extension(*enable*, /)

Enable dynamic loading of SQLite extension modules.

execute()

Executes an SQL statement.

executemany(*sql*, *parameters*, /)

Repeatedly executes an SQL statement.

executescript(*sql_script*, /)

Executes multiple SQL statements at once.

getlimit(*category*, /)

Get connection run-time limits.

category
The limit category to be queried.

interrupt()

Abort any pending database operation.

iterdump()

Returns iterator to the dump of the database in an SQL text format.

load_extension(*name*, /)

Load SQLite extension module.

manual_transaction()

rollback()

Roll back to the start of any pending transaction.

If there is no open transaction, this method is a no-op.

serialize(*, *name='main'*)

Serialize a database into a byte string.

name
Which database to serialize.

For an ordinary on-disk database file, the serialization is just a copy of the disk file. For an in-memory database or a "temp" database, the serialization is the same sequence of bytes which would be written to disk if that database were backed up to disk.

set_authorizer(*authorizer_callback*)

Sets authorizer callback.

set_progress_handler(*progress_handler*, *n*)

Sets progress handler callback.

set_trace_callback(*trace_callback*)

Sets a trace callback called for each SQL statement (passed as unicode).

setlimit(*category*, *limit*, */*)

Set connection run-time limits.

category
The limit category to be set.

limit
The new limit. If the new limit is a negative number, the limit is unchanged.

Attempts to increase a limit above its hard upper bound are silently truncated to the hard upper bound. Regardless of whether or not the limit was changed, the prior value of the limit is returned.

DataError

DatabaseError

Error

IntegrityError

InterfaceError

InternalError

NotSupportedError

OperationalError

ProgrammingError

Warning

in_transaction

isolation_level

row_factory

text_factory

total_changes

class aequilibrae.utils.db_utils.**ColumnDef**(*idx: int*, *name: str*, *type: str*, *not_null: bool*, *default: str*, *is_pk: bool*)

default: str

idx: int

is_pk: bool

name: str

not_null: bool

type: str

class aequilibrae.utils.db_utils.**commit_and_close**(*db: str | Path | Connection*, *commit: bool = True*, *missing_ok: bool = False*, *spatial=False*)

A context manager for sqlite connections which closes and commits.

aequilibrae.utils.db_utils.**add_column**(*conn*, *table_name*, *col_name*, *col_type*, *constraints=None*)

aequilibrae.utils.db_utils.**add_column_unless_exists**(*conn*, *table_name*, *col_name*, *col_type*, *constraints=None*)

aequilibrae.utils.db_utils.**get_schema**(*conn*, *table_name*)

aequilibrae.utils.db_utils.**has_column**(*conn*, *table_name*, *col_name*)

aequilibrae.utils.db_utils.**has_table**(*conn*, *table_name*)

aequilibrae.utils.db_utils.**list_columns**(*conn*, *table_name*)

aequilibrae.utils.db_utils.**list_tables_in_db**(*conn: Connection*)

aequilibrae.utils.db_utils.**read_and_close**(*filepath*, *spatial=False*)

A context manager for sqlite connections (alias for *commit_and_close(db,commit=False)*).

aequilibrae.utils.db_utils.**read_sql**(*sql*, *filepath*, ***kwargs*)

aequilibrae.utils.db_utils.**safe_connect**(*filepath: PathLike*, *missing_ok=False*)

aequilibrae.utils.find_table_fields

Functions

find_table_fields(table_name[, conn, db_path])	

aequilibrae.utils.find_table_fields.**find_table_fields**(*table_name*, *conn=None*, *db_path=None*)

aequilibrae.utils.geo_index

Classes

GeoIndex()	Implements a generic GeoIndex class that uses the QGIS index when using the GUI and RTree otherwise

class aequilibrae.utils.geo_index.**GeoIndex**

Implements a generic GeoIndex class that uses the QGIS index when using the GUI and RTree otherwise

build_from_layer(*layer*) → dict

delete(*feature_id*, *geometry: Point | Polygon | LineString | MultiPoint | MultiPolygon*)

insert(*feature_id: int*, *geometry: Point | Polygon | LineString | MultiPoint | MultiPolygon | MultiLineString*) → None

Inserts a valid shapely geometry in the index

Arguments

feature_id (`int`): ID of the geometry being inserted **geo** (`Shapely.geometry`): Any valid shapely geometry

nearest(*geo: Point | Polygon | LineString | MultiPoint | MultiPolygon*, *num_results*) → List[int]

Finds nearest neighbor for a given geometry

Arguments

geo (`Shapely geometry`): Any valid shapely geometry

num_results (`int`): A positive integer for the number of neighbors to return

Returns

neighbors (`List[int]`): List of IDs of the closest neighbors in the index

reset()

aequilibrae.utils.geo_utils

Convenience functions for working with geospatial data.

Functions

haversine(lon1, lat1, lon2, lat2)	Calculate the great circle distance between two points on the earth (specified in decimal degrees)
metre_crs_for_gdf(gdf)	

aequilibrae.utils.geo_utils.**haversine**(*lon1*, *lat1*, *lon2*, *lat2*)

Calculate the great circle distance between two points on the earth (specified in decimal degrees)

aequilibrae.utils.geo_utils.**metre_crs_for_gdf**(*gdf*)

aequilibrae.utils.get_table

Functions

get_geo_table(table_name, conn)	
get_table(table_name, conn)	Selects table from database.

aequilibrae.utils.get_table.**get_geo_table**(*table_name*, *conn*)

aequilibrae.utils.get_table.**get_table**(*table_name*, *conn*)

Selects table from database.

Arguments

table_name (`str`): desired table name **conn** (`sqlite3.Connection`): database connection

aequilibrae.utils.interface

Modules

worker_thread

aequilibrae.utils.interface.worker_thread

Classes

WorkerThread(*arg)

class aequilibrae.utils.interface.worker_thread.**WorkerThread**(**arg*)

aequilibrae.utils.list_tables_in_db

Functions

list_tables_in_db(conn)	Return a list with all tables within a database.

aequilibrae.utils.list_tables_in_db.**list_tables_in_db**(*conn*)

Return a list with all tables within a database.

Arguments

conn (:obj: *sqlite3.Connection*): database connection

aequilibrae.utils.model_run_utils

Functions

import_file_as_module(file, module_name[, force])	Import a file as a Python module.

aequilibrae.utils.model_run_utils.**import_file_as_module**(*file: Path*, *module_name*, *force: bool = False*)

Import a file as a Python module.

Arguments

file (`pathlib.Path`): Path object pointing to the file to import

module_name: Name to give the imported module

force: Replace the module in `sys.modules` if it exists.

Returns
The imported module

aequilibrae.utils.python_signal

Classes

PythonSignal(object)	This class provides a pure python equivalent of the Signal passing infrastructure present in QGIS.

class aequilibrae.utils.python_signal.**PythonSignal**(*object*)

This class provides a pure python equivalent of the Signal passing infrastructure present in QGIS. It takes updates in the same format as the QGIS progress bar manager used in QAequilibrae and translates them into TQDM syntax.

The expected structure of update data is a list where the first element is string describing the desired action:

['action', *args]

The currently supported formats for actions are listed here:

1. ['finished'] - close out the current progress bar
2. ['refresh'] - force the current progress bar to refresh
3. ['reset'] - reset the current progress bar
4. ['start', num_elements: int, desc: str] - start a new progress bar
5. ['set_position', pos: int] - set the position of the current progress bar
6. ['set_text', desc: str] - set the description of the current progress bar
7. ['update', pos: int, desc: str] - set both pos and desc of current progress bar

emit(*val*)

aequilibrae.utils.qgis_utils

aequilibrae.utils.simwrapper

Modules

generate_simwrapper_config	Utilities to generate SimWrapper dashboard configuration for an AequilibraE project.
simwrapper_panel	
simwrapper_utils	

aequilibrae.utils.simwrapper.generate_simwrapper_config

Utilities to generate SimWrapper dashboard configuration for an AequilibraE project.

Usage

Listing 7: Generate SimWrapper dashboard for an open project

```
>>> from aequilibrae.project import Project
>>> from aequilibrae.utils.simwrapper.generate_simwrapper_config import␣
↪SimwrapperConfigGenerator
>>> prj = Project()
>>> prj.open('/path/to/project')
>>> gen = SimwrapperConfigGenerator(prj, output_dir='simwrapper')
>>> gen.write_yamls()
```

Notes

- *output_dir* must be inside the project directory

Functions

main([argv])	Command-line entry point for generating SimWrapper configs.

Classes

SimwrapperConfigGenerator(project[, ...])	Generates SimWrapper dashboard configuration for an AequilibraE project.

class aequilibrae.utils.simwrapper.generate_simwrapper_config.**SimwrapperConfigGenerator**(*project*, *output_dir='simwrap*
max_results_tabl
results_tables=Non
centroid_link_types=

Generates SimWrapper dashboard configuration for an AequilibraE project.

write_yamls()

Writes the SimWrapper dashboard YAML file.

aequilibrae.utils.simwrapper.generate_simwrapper_config.**main**(*argv=None*)

Command-line entry point for generating SimWrapper configs.

Example

aequilibrae-simwrapper –project /path/to/project –output-dir simwrapper

aequilibrae.utils.simwrapper.simwrapper_panel

Classes

AequilibraEMapPanel(title[, database, ...])	Panel for rendering interactive AequilibraE network maps.

continues on next page

Table 143 – continued from previous page

AequilibraEResultsMapPanel(title[, project, ...])	Panel for rendering interactive AequilibraE network maps with optional results styling.
ConvergencePanel(title, config[, height, width])	Panel wrapper for Vega/Vega-Lite specifications.
SimwrapperPanel(type, title[, height, width])	Base class for all SimWrapper panels.
TextPanel(title, data[, is_file, height, width])	Panel for displaying text content.
TilePanel(title, dataset[, height, width, ...])	Panel used to display tabular summary statistics.

class aequilibrae.utils.simwrapper.simwrapper_panel.**AequilibraEMapPanel**(*title*, *database='project_database.sqlite'*, *height=None*, *width=None*, *center=None*, *zoom=None*)

Panel for rendering interactive AequilibraE network maps.

Parameters

- ****title**** (str) – Title
- ****database**** (str, optional) – Project database
- ****height**** (int, optional) – Panel height
- ****width**** (int, optional) – Panel width
- ****center**** (list, optional) – Map center coordinates
- ****zoom**** (int, optional) – Initial zoom level

add_layer(*name*, *layer_dict*)

Adds a layer definition under the given name

set_defaults(*defaults_dict=None*)

Sets default visuals for map layers

set_extra_databases(*database_dict*)

Registers extra databases used by map

set_legend(*legend_list*)

Sets legend configuration for the map

to_dict()

Returns dictionary representation of the panel

class aequilibrae.utils.simwrapper.simwrapper_panel.**AequilibraEResultsMapPanel**(*title*, *project=None*, *project_database='project_dat*, *results_database='results_datab*, *colour_metric=None*, *width_metric=None*, *height=None*, *width=None*, *center=None*, *zoom=None*, *palette='Temps'*, *results_table=None*, *sql_filter=None*)

Panel for rendering interactive AequilibraE network maps with optional results styling.

Behavior - When *results_table* is provided the panel LEFT JOINs the map *links* layer with that

table from an extra *results* database and reads metric values from the joined table.

- When *results_table* is omitted the panel reads metrics directly from the project's *links* table in *project_database.sqlite* (no separate results DB/table required).
- Data ranges used for styling are computed from the lower/upper percentiles (default 5th/95th) and pretty-rounded with *pretty_round*.
- If a metric or the corresponding table/column is missing the panel falls back to the default data range *[0, 1]* (graceful fallback).

Arguments

title (`str`): panel title shown in the dashboard **project** (`Project`, *Optional*): open Project instance used to read metric values

and compute percentile ranges; if not provided ranges default to [0, 1]

project_database (`str`): main project database filename (default: `project_database.sqlite`) **results_database** (`str`): results database filename (default: `results_database.sqlite`);

used only when `results_table` is provided

colour_metric (`str`, *Optional*): column name used to style line colour. Look-up order:
results table (if provided) → project *links* table

width_metric (`str`, *Optional*): column name used to style line width (same lookup rules) **results_table** (`str`, *Optional*): name of the table in the results DB to join to *links*.

When omitted, metrics are read from the project *links* table instead.

palette (`str`): colour palette for *colour_metric* scaling (default: `Temps`) **height**, **width**, **center**, **zoom**: visual/layout parameters (see

AequilibraEMapPanel)

sql_filter (`str`, *Optional*): SQL filter expression applied to the `links` layer

Notes

- The panel will register an *extraDatabases* entry named `results` only when `results_table` is specified.
- Ranges are computed from percentiles (5th/95th) and rounded via *aequilibrae.utils.simwrapper.simwrapper_utils.pretty_round()*.

Examples

- Read a metric from the project `links` table
- Read a metric from a results table and join it to `links`

add_layer(*name*, *layer_dict*)

Adds a layer definition under the given name

build_legend(*colour_range*, *width_range*)

set_colour_styling(*data_range*)

set_defaults(*defaults_dict=None*)

Sets default visuals for map layers

set_extra_databases(*database_dict*)

Registers extra databases used by map

set_legend(*legend_list*)

Sets legend configuration for the map

set_width_styling(*data_range*)

to_dict()

Returns dictionary representation of the panel

class aequilibrae.utils.simwrapper.simwrapper_panel.**ConvergencePanel**(*title*, *config*, *height=None*, *width=None*)

Panel wrapper for Vega/Vega-Lite specifications.

to_dict()

Returns dictionary representation of the panel.

class aequilibrae.utils.simwrapper.simwrapper_panel.**SimwrapperPanel**(*type*, *title*, *height=None*, *width=None*)

Base class for all SimWrapper panels.

Parameters

- ****type**** (`str`) – Panel type
- ****title**** (`str`) – Title to show in the dashboard
- ****height**** (`int`, optional) – Panel height
- ****width**** (`int`, optional) – Panel width

to_dict()

Returns dictionary representation of the panel.

class aequilibrae.utils.simwrapper.simwrapper_panel.**TextPanel**(*title*, *data*, *is_file=False*, *height=None*, *width=None*)

Panel for displaying text content.

Parameters

- ****title**** (str) – Title
- ****data**** (str) – Text content or file path
- ****is_file**** (bool, optional) – Whether data is a file reference
- ****height**** (int, optional) – Panel height
- ****width**** (int, optional) – Panel width

to_dict()

Returns dictionary representation of the panel.

class aequilibrae.utils.simwrapper.simwrapper_panel.**TilePanel**(*title*, *dataset*, *height=None*, *width=None*, *colors=None*)

Panel used to display tabular summary statistics.

Parameters

- ****title**** (str) – Title
- ****dataset**** (str or list) – CSV path or inline dataset
- ****height**** (int, optional) – Panel height
- ****width**** (int, optional) – Panel width

to_dict()

Returns dictionary representation of the panel.

aequilibrae.utils.simwrapper.simwrapper_utils

Functions

export_convergence_csv(results_dataframe, ...)	Write assignment convergence for all results tables into data_dir/assignment_convergence.csv.
get_links_bounds_box(project)	Compute box around all coordinates in links table of project.
get_project_center(project)	Finds center coordinates of project
get_project_zoom(project)	Finds a reasonable zoom level based on project links' reach
parse_convergence_json(json_string)	Parse a (possibly double-encoded) procedure-report JSON string.
pretty_round(value[, direction])	Round a value to a 'pretty' number (1, 2, 5 multiples of powers of 10).

aequilibrae.utils.simwrapper.simwrapper_utils.**export_convergence_csv**(*results_dataframe*, *data_dir*)

Write assignment convergence for all results tables into data_dir/assignment_convergence.csv.

Returns Path to written CSV, or None if there was no convergence data. Raises ValueError when iteration/rgap lengths mismatch for a scenario.

aequilibrae.utils.simwrapper.simwrapper_utils.**get_links_bounds_box**(*project*)

Compute box around all coordinates in links table of project. Queries spatial database to find max and min x and y coords across all link geometries to return overall network links' reach.

Returns bounding box values (xmin, ymin, xmax, ymax)

aequilibrae.utils.simwrapper.simwrapper_utils.**get_project_center**(*project*)

Finds center coordinates of project

aequilibrae.utils.simwrapper.simwrapper_utils.**get_project_zoom**(*project*)

Finds a reasonable zoom level based on project links' reach

aequilibrae.utils.simwrapper.simwrapper_utils.**parse_convergence_json**(*json_string*)

Parse a (possibly double-encoded) procedure-report JSON string.

Returns (iteration_list, rgap_list). For invalid/missing input returns ([], []).

aequilibrae.utils.simwrapper.simwrapper_utils.**pretty_round**(*value*, *direction='up'*)

Round a value to a 'pretty' number (1, 2, 5 multiples of powers of 10).

Arguments

value (`float`): value to round **direction** (`str`): 'up' to ceil, 'down' to floor

Returns

float: the rounded pretty number

aequilibrae.utils.spatialite_utils

Functions

connect_spatialite(path_to_file[, missing_ok])	
ensure_spatialite_binaries()	
is_not_windows()	
is_spatialite(conn)	
is_windows()	
load_spatialite_extension(conn)	
set_known_spatialite_folder(spatialite_folder)	
spatialize_db(conn[, logger])	

aequilibrae.utils.spatialite_utils.**connect_spatialite**(*path_to_file: PathLike*, *missing_ok: bool = False*) → Connection

aequilibrae.utils.spatialite_utils.**ensure_spatialite_binaries**() → None

aequilibrae.utils.spatialite_utils.**is_not_windows**()

aequilibrae.utils.spatialite_utils.**is_spatialite**(*conn*)

aequilibrae.utils.spatialite_utils.**is_windows**()

aequilibrae.utils.spatialite_utils.**load_spatialite_extension**(*conn: Connection*)

aequilibrae.utils.spatialite_utils.**set_known_spatialite_folder**(*spatialite_folder: PathLike*)

aequilibrae.utils.spatialite_utils.**spatialize_db**(*conn*, *logger=None*)

PYTHON MODULE INDEX

a

aequilibrae, 239
aequilibrae.context, 265
aequilibrae.distribution, 265
aequilibrae.distribution.gravity_application, 268
aequilibrae.distribution.gravity_calibration, 270
aequilibrae.distribution.ipf, 271
aequilibrae.distribution.ipf_core, 272
aequilibrae.distribution.synthetic_gravity_model, 272
aequilibrae.log, 272
aequilibrae.matrix, 273
aequilibrae.matrix.aequilibrae_matrix, 281
aequilibrae.matrix.coo_demand, 287
aequilibrae.matrix.sparse_matrix, 288
aequilibrae.parameters, 288
aequilibrae.paths, 289
aequilibrae.paths.all_or_nothing, 314
aequilibrae.paths.AoN, 314
aequilibrae.paths.assignment_paths, 314
aequilibrae.paths.connectivity_analysis, 315
aequilibrae.paths.graph, 316
aequilibrae.paths.graph_building, 323
aequilibrae.paths.linear_approximation, 323
aequilibrae.paths.multi_threaded_aon, 323
aequilibrae.paths.multi_threaded_paths, 323
aequilibrae.paths.multi_threaded_skimming, 324
aequilibrae.paths.network_skimming, 324
aequilibrae.paths.optimal_strategies, 325
aequilibrae.paths.public_transport, 325
aequilibrae.paths.results, 325
aequilibrae.paths.results.assignment_results, 329
aequilibrae.paths.results.path_results, 330
aequilibrae.paths.results.skim_results, 332
aequilibrae.paths.route_choice, 333
aequilibrae.paths.sub_area, 337
aequilibrae.paths.traffic_assignment, 337
aequilibrae.paths.traffic_class, 346
aequilibrae.paths.vdf, 347
aequilibrae.project, 347
aequilibrae.project.about, 360
aequilibrae.project.basic_table, 362
aequilibrae.project.data, 362
aequilibrae.project.data.matrices, 365
aequilibrae.project.data.matrix_record, 366
aequilibrae.project.data.result_record, 366
aequilibrae.project.data.results, 367
aequilibrae.project.data_loader, 369
aequilibrae.project.database_connection, 369
aequilibrae.project.field_editor, 369
aequilibrae.project.network, 370
aequilibrae.project.network.connector_creation, 382
aequilibrae.project.network.gmns_builder, 384
aequilibrae.project.network.gmns_exporter, 385
aequilibrae.project.network.haversine, 385
aequilibrae.project.network.link, 385
aequilibrae.project.network.link_type, 387
aequilibrae.project.network.link_types, 387
aequilibrae.project.network.links, 389
aequilibrae.project.network.mode, 390
aequilibrae.project.network.modes, 390
aequilibrae.project.network.network, 392
aequilibrae.project.network.node, 394
aequilibrae.project.network.nodes, 395
aequilibrae.project.network.osm, 397
aequilibrae.project.network.osm.model_area_gridding, 397
aequilibrae.project.network.osm.osm_builder, 397
aequilibrae.project.network.osm.osm_downloader, 398
aequilibrae.project.network.osm.osm_params, 398
aequilibrae.project.network.osm.place_getter, 398

aequilibrae.project.network.period, 399
aequilibrae.project.network.periods, 399
aequilibrae.project.network.safe_class, 401
aequilibrae.project.project, 401
aequilibrae.project.project_cleaning, 403
aequilibrae.project.project_creation, 404
aequilibrae.project.scenario, 404
aequilibrae.project.table_loader, 405
aequilibrae.project.tools, 405
aequilibrae.project.tools.cli, 409
aequilibrae.project.tools.migration_manager, 410
aequilibrae.project.tools.network_simplifier, 413
aequilibrae.project.zone, 414
aequilibrae.project.zoning, 415
aequilibrae.reference_files, 417
aequilibrae.transit, 417
aequilibrae.transit.column_order, 422
aequilibrae.transit.constants, 422
aequilibrae.transit.date_tools, 422
aequilibrae.transit.functions, 423
aequilibrae.transit.functions.breaking_links_from_stop_access, 423
aequilibrae.transit.functions.compute_line_bearing, 424
aequilibrae.transit.functions.get_srid, 424
aequilibrae.transit.gtfs_loader, 424
aequilibrae.transit.gtfs_writer, 425
aequilibrae.transit.gtfs_writer.agency_writer, 425
aequilibrae.transit.gtfs_writer.fare_writer, 425
aequilibrae.transit.gtfs_writer.routes_writer, 425
aequilibrae.transit.gtfs_writer.shape_writer, 426
aequilibrae.transit.gtfs_writer.stop_times_writer, 426
aequilibrae.transit.gtfs_writer.stops_writer, 426
aequilibrae.transit.gtfs_writer.trips_writer, 426
aequilibrae.transit.lib_gtfs, 426
aequilibrae.transit.parse_csv, 428
aequilibrae.transit.route_map_matcher, 428
aequilibrae.transit.route_system, 429
aequilibrae.transit.route_system_reader, 429
aequilibrae.transit.route_system_reader.agency_reader, 430
aequilibrae.transit.route_system_reader.pattern_reader, 430
aequilibrae.transit.route_system_reader.routes_reader, 430
aequilibrae.transit.route_system_reader.stop_reader, 430
aequilibrae.transit.route_system_reader.stop_times_reader, 430
aequilibrae.transit.route_system_reader.trips_reader, 431
aequilibrae.transit.transit, 431
aequilibrae.transit.transit_elements, 433
aequilibrae.transit.transit_elements.agency, 437
aequilibrae.transit.transit_elements.basic_element, 437
aequilibrae.transit.transit_elements.fare, 437
aequilibrae.transit.transit_elements.fare_rule, 438
aequilibrae.transit.transit_elements.link, 438
aequilibrae.transit.transit_elements.mode_correspondence, 439
aequilibrae.transit.transit_elements.pattern, 439
aequilibrae.transit.transit_elements.route, 440
aequilibrae.transit.transit_elements.service, 440
aequilibrae.transit.transit_elements.stop, 441
aequilibrae.transit.transit_elements.trip, 441
aequilibrae.transit.transit_graph_builder, 442
aequilibrae.utils, 446
aequilibrae.utils.aeq_signal, 446
aequilibrae.utils.core_setter, 446
aequilibrae.utils.create_delaunay_network, 446
aequilibrae.utils.create_example, 447
aequilibrae.utils.db_utils, 447
aequilibrae.utils.find_table_fields, 451
aequilibrae.utils.geo_index, 451
aequilibrae.utils.geo_utils, 452
aequilibrae.utils.get_table, 452
aequilibrae.utils.interface, 453
aequilibrae.utils.interface.worker_thread, 453
aequilibrae.utils.list_tables_in_db, 453
aequilibrae.utils.model_run_utils, 453
aequilibrae.utils.python_signal, 454
aequilibrae.utils.qgis_utils, 454
aequilibrae.utils.simwrapper, 454
aequilibrae.utils.simwrapper.generate_simwrapper_config,

454
aequilibrae.utils.simwrapper.simwrapper_panel, 455
aequilibrae.utils.simwrapper.simwrapper_utils, 459
aequilibrae.utils.spatialite_utils, 460

INDEX

A

about (*aequilibrae.Project property*), 257
about (*aequilibrae.project.Project property*), 356
about (*aequilibrae.project.project.Project property*), 403
about (*aequilibrae.project.Scenario attribute*), 357
about (*aequilibrae.project.scenario.Scenario attribute*), 404
About (*class in aequilibrae.project*), 347
About (*class in aequilibrae.project.about*), 361
activate() (*aequilibrae.Project method*), 256
activate() (*aequilibrae.project.Project method*), 355
activate() (*aequilibrae.project.project.Project method*), 401
activate_project() (*in module aequilibrae.context*), 265
add() (*aequilibrae.project.field_editor.FieldEditor method*), 370
add() (*aequilibrae.project.FieldEditor method*), 349
add() (*aequilibrae.project.network.Modes method*), 375
add() (*aequilibrae.project.network.modes.Modes method*), 391
add_centroid() (*aequilibrae.project.Zone method*), 357
add_centroid() (*aequilibrae.project.zone.Zone method*), 414
add_centroids() (*aequilibrae.project.Zoning method*), 359
add_centroids() (*aequilibrae.project.zoning.Zoning method*), 415
add_class() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
add_class() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
add_class() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 343
add_class() (*aequilibrae.paths.TrafficAssignment method*), 303
add_class() (*aequilibrae.paths.TransitAssignment method*), 307
add_class() (*aequilibrae.TrafficAssignment method*), 260
add_column() (*in module aequilibrae.utils.db_utils*), 451
add_column_unless_exists() (*in module aequilibrae.utils.db_utils*), 451
add_demand() (*aequilibrae.paths.route_choice.RouteChoice method*), 333
add_demand() (*aequilibrae.paths.RouteChoice method*), 296
add_df() (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand method*), 287
add_df() (*aequilibrae.matrix.GeneralisedCOODemand method*), 280
add_info_field() (*aequilibrae.project.About method*), 348
add_info_field() (*aequilibrae.project.about.About method*), 361
add_layer() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEMapPanel method*), 456
add_layer() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPa method*), 458
add_matrix() (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand method*), 287
add_matrix() (*aequilibrae.matrix.GeneralisedCOODemand method*), 280
add_mode() (*aequilibrae.project.network.Link method*), 371
add_mode() (*aequilibrae.project.network.link.Link method*), 386
add_preload() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
add_preload() (*aequilibrae.paths.TrafficAssignment method*), 303
add_preload() (*aequilibrae.TrafficAssignment method*), 260
add_subcommand_from_function() (*in module aequilibrae.project.tools.cli*), 409

add_triggers() (*in module aequilibrae.project.project_creation*), 404
add_zones() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 444
add_zones() (*aequilibrae.transit.TransitGraphBuilder method*), 420
aequilibrae
    module, 239
AequilibraEConnection (*class in aequilibrae.utils.db_utils*), 448
aequilibrae.context
    module, 265
aequilibrae.distribution
    module, 265
aequilibrae.distribution.gravity_application
    module, 268
aequilibrae.distribution.gravity_calibration
    module, 270
aequilibrae.distribution.ipf
    module, 271
aequilibrae.distribution.ipf_core
    module, 272
aequilibrae.distribution.synthetic_gravity_model
    module, 272
aequilibrae.log
    module, 272
AequilibraEMapPanel (*class in aequilibrae.utils.simwrapper.simwrapper_panel*), 456
aequilibrae.matrix
    module, 273
AequilibraeMatrix (*class in aequilibrae*), 239
AequilibraeMatrix (*class in aequilibrae.matrix*), 273
AequilibraeMatrix (*class in aequilibrae.matrix.aequilibrae_matrix*), 281
aequilibrae.matrix.aequilibrae_matrix
    module, 281
aequilibrae.matrix.coo_demand
    module, 287
aequilibrae.matrix.sparse_matrix
    module, 288
aequilibrae.parameters
    module, 288
aequilibrae.paths
    module, 289
aequilibrae.paths.all_or_nothing
    module, 314
aequilibrae.paths.AoN
    module, 314
aequilibrae.paths.assignment_paths
    module, 314
aequilibrae.paths.connectivity_analysis
    module, 315
aequilibrae.paths.graph
    module, 316
aequilibrae.paths.graph_building
    module, 323
aequilibrae.paths.linear_approximation
    module, 323
aequilibrae.paths.multi_threaded_aon
    module, 323
aequilibrae.paths.multi_threaded_paths
    module, 323
aequilibrae.paths.multi_threaded_skimming
    module, 324
aequilibrae.paths.network_skimming
    module, 324
aequilibrae.paths.optimal_strategies
    module, 325
aequilibrae.paths.public_transport
    module, 325
aequilibrae.paths.results
    module, 325
aequilibrae.paths.results.assignment_results
    module, 329
aequilibrae.paths.results.path_results
    module, 330
aequilibrae.paths.results.skim_results
    module, 332
aequilibrae.paths.route_choice
    module, 333
aequilibrae.paths.sub_area
    module, 337
aequilibrae.paths.traffic_assignment
    module, 337
aequilibrae.paths.traffic_class
    module, 346
aequilibrae.paths.vdf
    module, 347
aequilibrae.project
    module, 347
aequilibrae.project.about
    module, 360
aequilibrae.project.basic_table
    module, 362
aequilibrae.project.data
    module, 362
aequilibrae.project.data_loader
    module, 369
aequilibrae.project.database_connection
    module, 369
aequilibrae.project.data.matrices
    module, 365
aequilibrae.project.data.matrix_record
    module, 366
aequilibrae.project.data.result_record
    module, 366

aequilibrae.project.data.results
    module, 367
aequilibrae.project.field_editor
    module, 369
aequilibrae.project.network
    module, 370
aequilibrae.project.network.connector_creation
    module, 382
aequilibrae.project.network.gmns_builder
    module, 384
aequilibrae.project.network.gmns_exporter
    module, 385
aequilibrae.project.network.haversine
    module, 385
aequilibrae.project.network.link
    module, 385
aequilibrae.project.network.link_type
    module, 387
aequilibrae.project.network.link_types
    module, 387
aequilibrae.project.network.links
    module, 389
aequilibrae.project.network.mode
    module, 390
aequilibrae.project.network.modes
    module, 390
aequilibrae.project.network.network
    module, 392
aequilibrae.project.network.node
    module, 394
aequilibrae.project.network.nodes
    module, 395
aequilibrae.project.network.osm
    module, 397
aequilibrae.project.network.osm.model_area_graph_building
    module, 397
aequilibrae.project.network.osm.osm_builder
    module, 397
aequilibrae.project.network.osm.osm_downloader
    module, 398
aequilibrae.project.network.osm.osm_params
    module, 398
aequilibrae.project.network.osm.place_getter
    module, 398
aequilibrae.project.network.period
    module, 399
aequilibrae.project.network.periods
    module, 399
aequilibrae.project.network.safe_class
    module, 401
aequilibrae.project.project
    module, 401
aequilibrae.project.project_cleaning
    module, 403
aequilibrae.project.project_creation
    module, 404
aequilibrae.project.scenario
    module, 404
aequilibrae.project.table_loader
    module, 405
aequilibrae.project.tools
    module, 405
aequilibrae.project.tools.cli
    module, 409
aequilibrae.project.tools.migration_manager
    module, 410
aequilibrae.project.tools.network_simplifier
    module, 413
aequilibrae.project.zone
    module, 414
aequilibrae.project.zoning
    module, 415
aequilibrae.reference_files
    module, 417
AequilibraEResultsMapPanel (*class in aequilibrae.utils.simwrapper.simwrapper_panel*), 456
aequilibrae.transit
    module, 417
aequilibrae.transit.column_order
    module, 422
aequilibrae.transit.constants
    module, 422
aequilibrae.transit.date_tools
    module, 422
aequilibrae.transit.functions
    module, 423
aequilibrae.transit.functions.breaking_links_for_stop_ac
    module, 423
aequilibrae.transit.functions.compute_line_bearing
    module, 424
aequilibrae.transit.functions.get_srid
    module, 424
aequilibrae.transit.gtfs_loader
    module, 424
aequilibrae.transit.gtfs_writer
    module, 425
aequilibrae.transit.gtfs_writer.agency_writer
    module, 425
aequilibrae.transit.gtfs_writer.fare_writer
    module, 425
aequilibrae.transit.gtfs_writer.routes_writer
    module, 425
aequilibrae.transit.gtfs_writer.shape_writer
    module, 426
aequilibrae.transit.gtfs_writer.stop_times_writer
    module, 426
aequilibrae.transit.gtfs_writer.stops_writer
    module, 426

aequilibrae.transit.gtfs_writer.trips_writer
    module, 426
aequilibrae.transit.lib_gtfs
    module, 426
aequilibrae.transit.parse_csv
    module, 428
aequilibrae.transit.route_map_matcher
    module, 428
aequilibrae.transit.route_system
    module, 429
aequilibrae.transit.route_system_reader
    module, 429
aequilibrae.transit.route_system_reader.agency_reader
    module, 430
aequilibrae.transit.route_system_reader.pattern_reader
    module, 430
aequilibrae.transit.route_system_reader.routes_reader
    module, 430
aequilibrae.transit.route_system_reader.stop_reader
    module, 430
aequilibrae.transit.route_system_reader.stop_times_reader
    module, 430
aequilibrae.transit.route_system_reader.trips_reader
    module, 431
aequilibrae.transit.transit
    module, 431
aequilibrae.transit.transit_elements
    module, 433
aequilibrae.transit.transit_elements.agency
    module, 437
aequilibrae.transit.transit_elements.basic_element
    module, 437
aequilibrae.transit.transit_elements.fare
    module, 437
aequilibrae.transit.transit_elements.fare_rule
    module, 438
aequilibrae.transit.transit_elements.link
    module, 438
aequilibrae.transit.transit_elements.mode_correspondence
    module, 439
aequilibrae.transit.transit_elements.pattern
    module, 439
aequilibrae.transit.transit_elements.route
    module, 440
aequilibrae.transit.transit_elements.service
    module, 440
aequilibrae.transit.transit_elements.stop
    module, 441
aequilibrae.transit.transit_elements.trip
    module, 441
aequilibrae.transit.transit_graph_builder
    module, 442
aequilibrae.utils
    module, 446
aequilibrae.utils.aeq_signal
    module, 446
aequilibrae.utils.core_setter
    module, 446
aequilibrae.utils.create_delaunay_network
    module, 446
aequilibrae.utils.create_example
    module, 447
aequilibrae.utils.db_utils
    module, 447
aequilibrae.utils.find_table_fields
    module, 451
aequilibrae.utils.geo_index
    module, 451
aequilibrae.utils.geo_utils
    module, 452
aequilibrae.utils.get_table
    module, 452
aequilibrae.utils.interface
    module, 453
aequilibrae.utils.interface.worker_thread
    module, 453
aequilibrae.utils.list_tables_in_db
    module, 453
aequilibrae.utils.model_run_utils
    module, 453
aequilibrae.utils.python_signal
    module, 454
aequilibrae.utils.qgis_utils
    module, 454
aequilibrae.utils.simwrapper
    module, 454
aequilibrae.utils.simwrapper.generate_simwrapper_config
    module, 454
aequilibrae.utils.simwrapper.simwrapper_panel
    module, 455
aequilibrae.utils.simwrapper.simwrapper_utils
    module, 459
aequilibrae.utils.spatialite_utils
    module, 460
agencies (*aequilibrae.transit.constants.Constants attribute*), 422
Agency (*class in aequilibrae.transit.transit_elements*), 433
Agency (*class in aequilibrae.transit.transit_elements.agency*), 437
algorithm (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 338
algorithm (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
algorithm (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
algorithm (*aequilibrae.paths.TrafficAssignment attribute*), 306

algorithm (*aequilibrae.paths.TransitAssignment attribute*), 309
algorithms_available() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
algorithms_available() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
algorithms_available() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
algorithms_available() (*aequilibrae.paths.TrafficAssignment method*), 303
algorithms_available() (*aequilibrae.paths.TransitAssignment method*), 307
algorithms_available() (*aequilibrae.TrafficAssignment method*), 260
all_algorithms (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
all_algorithms (*aequilibrae.paths.RouteChoice attribute*), 300
all_algorithms (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
all_algorithms (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
all_algorithms (*aequilibrae.paths.TrafficAssignment attribute*), 306
all_algorithms (*aequilibrae.paths.TransitAssignment attribute*), 309
all_algorithms (*aequilibrae.TrafficAssignment attribute*), 263
all_fields() (*aequilibrae.project.field_editor.FieldEditor method*), 370
all_fields() (*aequilibrae.project.FieldEditor method*), 349
all_modes() (*aequilibrae.project.network.Modes method*), 375
all_modes() (*aequilibrae.project.network.modes.Modes method*), 391
all_types() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
all_types() (*aequilibrae.project.network.LinkTypes method*), 372
all_zones() (*aequilibrae.project.Zoning method*), 359
all_zones() (*aequilibrae.project.zoning.Zoning method*), 415
allOrNothing (*class in aequilibrae*), 264
allOrNothing (*class in aequilibrae.paths*), 312
allOrNothing (*class in aequilibrae.paths.all_or_nothing*), 314
APPLIED (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
APPLIED (*aequilibrae.project.tools.MigrationStatus attribute*), 408
apply() (*aequilibrae.distribution.gravity_application.GravityApplication method*), 270
apply() (*aequilibrae.distribution.GravityApplication method*), 266
apply() (*aequilibrae.GravityApplication method*), 249
apply() (*aequilibrae.project.tools.Migration method*), 405
apply() (*aequilibrae.project.tools.migration_manager.Migration method*), 410
as_integer_ratio() (*aequilibrae.project.tools.migration_manager.MigrationStatus method*), 412
as_integer_ratio() (*aequilibrae.project.tools.MigrationStatus method*), 407
assemble_shape() (*aequilibrae.transit.route_map_matcher.RouteMapMatcher method*), 429
assign() (*aequilibrae.paths.HyperpathGenerating method*), 293
assign_matrix() (*aequilibrae.utils.create_delaunay_network.DelaunayAnalysis method*), 447
assignment (*aequilibrae.paths.linear_approximation.LinearApproximation attribute*), 323
assignment (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 338
assignment (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
assignment (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
assignment (*aequilibrae.paths.TrafficAssignment attribute*), 306
assignment (*aequilibrae.paths.TransitAssignment attribute*), 309
AssignmentBase (*class in aequilibrae.paths.traffic_assignment*), 338
AssignmentPaths (*class in aequilibrae*), 245
AssignmentPaths (*class in aequilibrae.paths*), 289
AssignmentPaths (*class in aequilibrae.paths.assignment_paths*), 314
AssignmentResults (*class in aequilibrae*), 245
AssignmentResults (*class in aequilibrae.paths*), 289
AssignmentResults (*class in aequilibrae.paths.results*), 325
AssignmentResults (*class in aequilibrae.paths.results.assignment_results*), 329
AssignmentResultsBase (*class in aequilibrae.paths.results.assignment_results*), 329
AssignmentResultsTable (*class in aequili-*

brae.paths.assignment_paths), 315
available_skims() (*aequilibrae.Graph method*), 246
available_skims() (*aequilibrae.paths.Graph method*), 291
available_skims() (*aequilibrae.paths.graph.Graph method*), 316
available_skims() (*aequilibrae.paths.graph.GraphBase method*), 318
available_skims() (*aequilibrae.paths.graph.TransitGraph method*), 320
available_skims() (*aequilibrae.paths.TransitGraph method*), 310

B

backup() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
base_path (*aequilibrae.project.Scenario attribute*), 357
base_path (*aequilibrae.project.scenario.Scenario attribute*), 404
BasicPTElement (*class in aequilibrae.transit.transit_elements.basic_element*), 437
BasicTable (*class in aequilibrae.project.basic_table*), 362
batches() (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand method*), 287
batches() (*aequilibrae.matrix.GeneralisedCOODemand method*), 280
best_shape() (*aequilibrae.transit.transit_elements.Pattern method*), 434
best_shape() (*aequilibrae.transit.transit_elements.pattern.Pattern method*), 439
bit_count() (*aequilibrae.project.tools.migration_manager.MigrationStatus method*), 412
bit_count() (*aequilibrae.project.tools.MigrationStatus method*), 408
bit_length() (*aequilibrae.project.tools.migration_manager.MigrationStatus method*), 412
bit_length() (*aequilibrae.project.tools.MigrationStatus method*), 408
blobopen() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
bpr_parameters (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
bpr_parameters (*aequilibrae.paths.TrafficAssignment attribute*), 306
bpr_parameters (*aequilibrae.TrafficAssignment attribute*), 263
build_from_layer() (*aequilibrae.utils.geo_index.GeoIndex method*), 452
build_geo() (*aequilibrae.transit.transit_elements.Link method*), 434
build_geo() (*aequilibrae.transit.transit_elements.link.Link method*), 439
build_graphs() (*aequilibrae.project.Network method*), 350
build_graphs() (*aequilibrae.project.network.Network method*), 375
build_graphs() (*aequilibrae.project.network.network.Network method*), 392
build_legend() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPar method*), 458
build_pt_preload() (*aequilibrae.transit.Transit method*), 417
build_pt_preload() (*aequilibrae.transit.transit.Transit method*), 431
builds_map_matchers() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 426
bulk_connector_creation() (*in module aequilibrae.project.network.connector_creation*), 382

C

calculate_biconjugate_direction() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323
calculate_conjugate_stepsize() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323
calculate_stepsize() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323
calibrate() (*aequilibrae.distribution.gravity_calibration.GravityCalibration method*), 271
calibrate() (*aequilibrae.distribution.GravityCalibration method*), 267
calibrate() (*aequilibrae.GravityCalibration method*), 250
capacity (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
capacity (*aequilibrae.paths.TrafficAssignment attribute*), 306
capacity_field (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
capacity_field (*aequilibrae.paths.TrafficAssignment attribute*), 306
check_convergence() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323

check_exists() (*aequilibrae.Matrices method*), 252
check_exists() (*aequilibrae.project.data.Matrices method*), 362
check_exists() (*aequilibrae.project.data.matrices.Matrices method*), 365
check_exists() (*aequilibrae.project.data.Results method*), 363
check_exists() (*aequilibrae.project.data.results.Results method*), 367
check_exists() (*aequilibrae.project.Matrices method*), 350
check_file_indices() (*aequilibrae.Project method*), 256
check_file_indices() (*aequilibrae.project.Project method*), 355
check_file_indices() (*aequilibrae.project.project.Project method*), 401
check_skim_cols() (*aequilibrae.paths.HyperpathGenerating method*), 293
classes (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 338
classes (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
classes (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
classes (*aequilibrae.paths.TrafficAssignment attribute*), 306
classes (*aequilibrae.paths.TransitAssignment attribute*), 309
clean() (*in module aequilibrae.project.project_cleaning*), 403
clear() (*aequilibrae.Log method*), 251
clear() (*aequilibrae.log.Log method*), 273
clear() (*aequilibrae.project.Log method*), 349
clear_database() (*aequilibrae.Matrices method*), 252
clear_database() (*aequilibrae.project.data.Matrices method*), 362
clear_database() (*aequilibrae.project.data.matrices.Matrices method*), 365
clear_database() (*aequilibrae.project.data.Results method*), 363
clear_database() (*aequilibrae.project.data.results.Results method*), 367
clear_database() (*aequilibrae.project.Matrices method*), 350
cli() (*in module aequilibrae.project.tools.cli*), 409
clone_scenario() (*aequilibrae.Project method*), 256
clone_scenario() (*aequilibrae.project.Project method*), 355
clone_scenario() (*aequilibrae.project.project.Project method*), 401
close() (*aequilibrae.AequilibraeMatrix method*), 239
close() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 281
close() (*aequilibrae.matrix.AequilibraeMatrix method*), 273
close() (*aequilibrae.Project method*), 256
close() (*aequilibrae.project.Project method*), 355
close() (*aequilibrae.project.project.Project method*), 402
close() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
collapse_links_into_nodes() (*aequilibrae.project.NetworkSimplifier method*), 353
collapse_links_into_nodes() (*aequilibrae.project.tools.network_simplifier.NetworkSimplifier method*), 413
collapse_links_into_nodes() (*aequilibrae.project.tools.NetworkSimplifier method*), 409
ColumnDef (*class in aequilibrae.utils.db_utils*), 451
columns() (*aequilibrae.AequilibraeMatrix method*), 239
columns() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 281
columns() (*aequilibrae.matrix.AequilibraeMatrix method*), 273
commit() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
commit_and_close (*class in aequilibrae.utils.db_utils*), 451
computational_view() (*aequilibrae.AequilibraeMatrix method*), 240
computational_view() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 281
computational_view() (*aequilibrae.matrix.AequilibraeMatrix method*), 274
compute_line_bearing() (*in module aequilibrae.transit.functions.compute_line_bearing*), 424
compute_path() (*aequilibrae.Graph method*), 246
compute_path() (*aequilibrae.PathResults method*), 255
compute_path() (*aequilibrae.paths.Graph method*), 291
compute_path() (*aequilibrae.paths.graph.Graph method*), 316
compute_path() (*aequilibrae.paths.graph.GraphBase method*), 318
compute_path() (*aequilibrae.paths.graph.TransitGraph method*), 321
compute_path() (*aequilibrae.paths.PathResults method*), 296
compute_path() (*aequilibrae.paths.results.path_results.PathResults method*), 331
compute_path() (*aequilibrae.paths.results.PathResults method*), 326

compute_path() (*aequilibrae.paths.TransitGraph method*), 310
compute_skim_cols() (*aequilibrae.paths.HyperpathGenerating method*), 293
compute_skims() (*aequilibrae.Graph method*), 247
compute_skims() (*aequilibrae.paths.Graph method*), 291
compute_skims() (*aequilibrae.paths.graph.Graph method*), 316
compute_skims() (*aequilibrae.paths.graph.GraphBase method*), 318
compute_skims() (*aequilibrae.paths.graph.TransitGraph method*), 321
compute_skims() (*aequilibrae.paths.TransitGraph method*), 311
config (*aequilibrae.paths.graph.TransitGraph property*), 322
config (*aequilibrae.paths.TransitGraph property*), 312
config (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder property*), 445
config (*aequilibrae.transit.TransitGraphBuilder property*), 421
congested_time (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
congested_time (*aequilibrae.paths.TrafficAssignment attribute*), 306
conjugate() (*aequilibrae.project.tools.migration_manager.MigrationStatus method*), 413
conjugate() (*aequilibrae.project.tools.MigrationStatus method*), 408
connect_mode() (*aequilibrae.project.network.Node method*), 378
connect_mode() (*aequilibrae.project.network.node.Node method*), 395
connect_mode() (*aequilibrae.project.Zone method*), 358
connect_mode() (*aequilibrae.project.zone.Zone method*), 414
connect_mode() (*aequilibrae.project.Zoning method*), 359
connect_mode() (*aequilibrae.project.zoning.Zoning method*), 415
connect_spatialite() (*in module aequilibrae.utils.spatialite_utils*), 460
connectivity (*aequilibrae.paths.connectivity_analysis.ConnectivityAnalysis attribute*), 315
connectivity (*aequilibrae.paths.ConnectivityAnalysis attribute*), 291
ConnectivityAnalysis (*class in aequilibrae.paths*), 290
ConnectivityAnalysis (*class in aequilibrae.paths.connectivity_analysis*), 315
connector_creation() (*in module aequilibrae.project.network.connector_creation*), 383
Constants (*class in aequilibrae.transit.constants*), 422
contents() (*aequilibrae.Log method*), 251
contents() (*aequilibrae.log.Log method*), 273
contents() (*aequilibrae.project.Log method*), 349
ConvergencePanel (*class in aequilibrae.utils.simwrapper.simwrapper_panel*), 458
convert_demand_matrix_from_zone_to_node_ids() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 444
convert_demand_matrix_from_zone_to_node_ids() (*aequilibrae.transit.TransitGraphBuilder method*), 420
convex_hull() (*aequilibrae.project.Network method*), 351
convex_hull() (*aequilibrae.project.network.Network method*), 376
convex_hull() (*aequilibrae.project.network.network.Network method*), 392
COO (*class in aequilibrae.matrix*), 279
COO (*class in aequilibrae.matrix.sparse_matrix*), 288
copy() (*aequilibrae.AequilibraeMatrix method*), 240
copy() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 282
copy() (*aequilibrae.matrix.AequilibraeMatrix method*), 274
copy_link() (*aequilibrae.project.network.Links method*), 373
copy_link() (*aequilibrae.project.network.links.Links method*), 389
cores (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
cores (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 338
cores (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
cores (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
cores (*aequilibrae.paths.TrafficAssignment attribute*), 306
cores (*aequilibrae.paths.TransitAssignment attribute*), 309
correct_geometries() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 384
count_centroids() (*aequilibrae.project.Network method*), 351
count_centroids() (*aequilibrae.project.network.Network method*), 376
count_centroids() (*aequilibrae.project.network.network.Network method*),

392
count_links() (*aequilibrae.project.Network method*), 351
count_links() (*aequilibrae.project.network.Network method*), 376
count_links() (*aequilibrae.project.network.network.Network method*), 392
count_nodes() (*aequilibrae.project.Network method*), 351
count_nodes() (*aequilibrae.project.network.Network method*), 376
count_nodes() (*aequilibrae.project.network.network.Network method*), 392
coverage() (*aequilibrae.project.Zoning method*), 359
coverage() (*aequilibrae.project.zoning.Zoning method*), 416
create() (*aequilibrae.project.About method*), 348
create() (*aequilibrae.project.about.About method*), 361
create_additional_db_fields() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 444
create_additional_db_fields() (*aequilibrae.transit.TransitGraphBuilder method*), 420
create_aggregate() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
create_base_tables() (*in module aequilibrae.project.project_creation*), 404
create_collation() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
create_compressed_link_network_mapping() (*aequilibrae.Graph method*), 247
create_compressed_link_network_mapping() (*aequilibrae.paths.Graph method*), 291
create_compressed_link_network_mapping() (*aequilibrae.paths.graph.Graph method*), 316
create_compressed_link_network_mapping() (*aequilibrae.paths.graph.GraphBase method*), 319
create_compressed_link_network_mapping() (*aequilibrae.paths.graph.TransitGraph method*), 321
create_compressed_link_network_mapping() (*aequilibrae.paths.TransitGraph method*), 311
create_days_between() (*in module aequilibrae.transit.date_tools*), 422
create_empty() (*aequilibrae.AequilibraeMatrix method*), 241
create_empty() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 282
create_empty() (*aequilibrae.matrix.AequilibraeMatrix method*), 275
create_empty_scenario() (*aequilibrae.Project method*), 256
create_empty_scenario() (*aequilibrae.project.Project method*), 355
create_empty_scenario() (*aequilibrae.project.project.Project method*), 402
create_example() (*in module aequilibrae.utils.create_example*), 447
create_from_gmns() (*aequilibrae.project.Network method*), 351
create_from_gmns() (*aequilibrae.project.network.Network method*), 376
create_from_gmns() (*aequilibrae.project.network.network.Network method*), 393
create_from_omx() (*aequilibrae.AequilibraeMatrix method*), 241
create_from_omx() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 283
create_from_omx() (*aequilibrae.matrix.AequilibraeMatrix method*), 276
create_from_osm() (*aequilibrae.project.Network method*), 351
create_from_osm() (*aequilibrae.project.network.Network method*), 376
create_from_osm() (*aequilibrae.project.network.network.Network method*), 393
create_from_trip_list() (*aequilibrae.AequilibraeMatrix method*), 242
create_from_trip_list() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 283
create_from_trip_list() (*aequilibrae.matrix.AequilibraeMatrix method*), 276
create_function() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
create_graph() (*aequilibrae.transit.Transit method*), 417
create_graph() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 444
create_graph() (*aequilibrae.transit.TransitGraphBuilder method*), 420
create_graph() (*aequilibrae.transit.transit.Transit method*), 431
create_line_geometry() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder*

method), 445
create_line_geometry() (*aequilibrae.transit.TransitGraphBuilder method*), 420
create_network() (*aequilibrae.utils.create_delaunay_network.DelaunayAnalysis method*), 447
create_od_node_mapping() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
create_od_node_mapping() (*aequilibrae.transit.TransitGraphBuilder method*), 420
create_transit_database() (*aequilibrae.transit.Transit method*), 417
create_transit_database() (*aequilibrae.transit.transit.Transit method*), 432
create_window_function() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 448
create_zoning_layer() (*aequilibrae.project.Zoning method*), 359
create_zoning_layer() (*aequilibrae.project.zoning.Zoning method*), 416
cursor() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449

D

data (*aequilibrae.project.network.Links property*), 374
data (*aequilibrae.project.network.links.Links property*), 390
data (*aequilibrae.project.network.Nodes property*), 380
data (*aequilibrae.project.network.nodes.Nodes property*), 396
data (*aequilibrae.project.network.Periods property*), 381
data (*aequilibrae.project.network.periods.Periods property*), 400
data (*aequilibrae.project.Periods property*), 354
data (*aequilibrae.project.Zoning property*), 360
data (*aequilibrae.project.zoning.Zoning property*), 417
data (*aequilibrae.transit.transit_elements.Route property*), 435
data (*aequilibrae.transit.transit_elements.route.Route property*), 440
data (*aequilibrae.transit.transit_elements.Stop property*), 436
data (*aequilibrae.transit.transit_elements.stop.Stop property*), 441
data_fields() (*aequilibrae.project.network.Link method*), 371
data_fields() (*aequilibrae.project.network.link.Link method*), 386
data_fields() (*aequilibrae.project.network.Node method*), 378
data_fields() (*aequilibrae.project.network.node.Node method*), 395
data_fields() (*aequilibrae.project.network.Period method*), 380
data_fields() (*aequilibrae.project.network.period.Period method*), 399
data_fields() (*aequilibrae.project.Period method*), 353
database_connection() (*in module aequilibrae.project.database_connection*), 369
database_path() (*in module aequilibrae.project.database_connection*), 369
DatabaseError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
DataError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
DataLoader (*class in aequilibrae.project.data_loader*), 369
dates_available() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
day_of_week() (*in module aequilibrae.transit.date_tools*), 422
db_connection (*aequilibrae.Project property*), 257
db_connection (*aequilibrae.project.Project property*), 356
db_connection (*aequilibrae.project.project.Project property*), 403
db_connection_spatial (*aequilibrae.Project property*), 257
db_connection_spatial (*aequilibrae.project.Project property*), 356
db_connection_spatial (*aequilibrae.project.project.Project property*), 403
deactivate() (*aequilibrae.Project method*), 256
deactivate() (*aequilibrae.project.Project method*), 355
deactivate() (*aequilibrae.project.project.Project method*), 402
default (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451
default_capacities (*aequilibrae.transit.Transit attribute*), 418
default_capacities (*aequilibrae.transit.transit.Transit attribute*), 432
default_headers() (*in module aequilibrae.project.network.osm.osm_params*), 398
default_parameters (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
default_parameters (*aequilibrae.paths.RouteChoice attribute*), 301
default_pces (*aequilibrae.transit.Transit attribute*), 418
default_pces (*aequilibrae.transit.transit.Transit at-*

tribute), 432
default_period (*aequilibrae.project.network.Periods property*), 381
default_period (*aequilibrae.project.network.periods.Periods property*), 401
default_period (*aequilibrae.project.Periods property*), 354
default_types() (*aequilibrae.Graph method*), 247
default_types() (*aequilibrae.paths.Graph method*), 292
default_types() (*aequilibrae.paths.graph.Graph method*), 317
default_types() (*aequilibrae.paths.graph.GraphBase method*), 319
default_types() (*aequilibrae.paths.graph.TransitGraph method*), 321
default_types() (*aequilibrae.paths.TransitGraph method*), 311
DelaunayAnalysis (*class in aequilibrae.utils.create_delaunay_network*), 447
delete() (*aequilibrae.project.data.matrix_record.MatrixRecord method*), 366
delete() (*aequilibrae.project.data.result_record.ResultRecord method*), 367
delete() (*aequilibrae.project.network.Link method*), 371
delete() (*aequilibrae.project.network.link_type.LinkType method*), 387
delete() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
delete() (*aequilibrae.project.network.link.Link method*), 386
delete() (*aequilibrae.project.network.Links method*), 373
delete() (*aequilibrae.project.network.links.Links method*), 389
delete() (*aequilibrae.project.network.LinkType method*), 371
delete() (*aequilibrae.project.network.LinkTypes method*), 373
delete() (*aequilibrae.project.network.Modes method*), 375
delete() (*aequilibrae.project.network.modes.Modes method*), 391
delete() (*aequilibrae.project.Zone method*), 358
delete() (*aequilibrae.project.zone.Zone method*), 415
delete() (*aequilibrae.utils.geo_index.GeoIndex method*), 452
delete_record() (*aequilibrae.Matrices method*), 252
delete_record() (*aequilibrae.project.data.Matrices method*), 362
delete_record() (*aequilibrae.project.data.matrices.Matrices method*), 365
delete_record() (*aequilibrae.project.data.Results method*), 363
delete_record() (*aequilibrae.project.data.results.Results method*), 367
delete_record() (*aequilibrae.project.Matrices method*), 350
demand_index_names (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
demand_index_names (*aequilibrae.paths.RouteChoice attribute*), 301
denominator (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
denominator (*aequilibrae.project.tools.MigrationStatus attribute*), 408
description (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 339
description (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
description (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
description (*aequilibrae.paths.TrafficAssignment attribute*), 306
description (*aequilibrae.paths.TransitAssignment attribute*), 309
deserialize() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
df (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand attribute*), 287
df (*aequilibrae.matrix.GeneralisedCOODemand attribute*), 280
disconnect_mode() (*aequilibrae.project.Zone method*), 358
disconnect_mode() (*aequilibrae.project.zone.Zone method*), 415
doWork() (*aequilibrae.allOrNothing method*), 264
doWork() (*aequilibrae.NetworkSkimming method*), 253
doWork() (*aequilibrae.paths.all_or_nothing.allOrNothing method*), 314
doWork() (*aequilibrae.paths.allOrNothing method*), 313
doWork() (*aequilibrae.paths.connectivity_analysis.ConnectivityAnalysis method*), 315
doWork() (*aequilibrae.paths.ConnectivityAnalysis method*), 290
doWork() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323
doWork() (*aequilibrae.paths.network_skimming.NetworkSkimming method*), 324
doWork() (*aequilibrae.paths.NetworkSkimming method*), 294

doWork() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 384
doWork() (*aequilibrae.project.network.gmns_exporter.GMNSExporter method*), 385
doWork() (*aequilibrae.project.network.osm.osm_builder.OSMBuilder method*), 397
doWork() (*aequilibrae.project.network.osm.osm_downloader.OSMDownloader method*), 398
doWork() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
drop_mode() (*aequilibrae.project.network.Link method*), 371
drop_mode() (*aequilibrae.project.network.link.Link method*), 386

E

emit() (*aequilibrae.utils.python_signal.PythonSignal method*), 454
empty (*class in aequilibrae.transit.parse_csv*), 428
enable_load_extension() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
ensure_spatialite_binaries() (*in module aequilibrae.utils.spatialite_utils*), 460
equilibration (*aequilibrae.paths.linear_approximation.LinearApproximation attribute*), 323
Error (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
exclude_links() (*aequilibrae.Graph method*), 247
exclude_links() (*aequilibrae.paths.Graph method*), 292
exclude_links() (*aequilibrae.paths.graph.Graph method*), 317
exclude_links() (*aequilibrae.paths.graph.GraphBase method*), 319
exclude_links() (*aequilibrae.paths.graph.TransitGraph method*), 322
exclude_links() (*aequilibrae.paths.TransitGraph method*), 311
execute() (*aequilibrae.allOrNothing method*), 264
execute() (*aequilibrae.NetworkSkimming method*), 253
execute() (*aequilibrae.paths.all_or_nothing.allOrNothing method*), 314
execute() (*aequilibrae.paths.allOrNothing method*), 313
execute() (*aequilibrae.paths.connectivity_analysis.ConnectivityAnalysis method*), 315
execute() (*aequilibrae.paths.ConnectivityAnalysis method*), 290
execute() (*aequilibrae.paths.linear_approximation.LinearApproximation method*), 323
execute() (*aequilibrae.paths.network_skimming.NetworkSkimming method*), 324
execute() (*aequilibrae.paths.NetworkSkimming method*), 294
execute() (*aequilibrae.paths.optimal_strategies.OptimalStrategies method*), 325
execute() (*aequilibrae.paths.OptimalStrategies method*), 295
execute() (*aequilibrae.paths.route_choice.RouteChoice method*), 333
execute() (*aequilibrae.paths.RouteChoice method*), 297
execute() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
execute() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
execute() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
execute() (*aequilibrae.paths.TrafficAssignment method*), 303
execute() (*aequilibrae.paths.TransitAssignment method*), 308
execute() (*aequilibrae.TrafficAssignment method*), 260
execute() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
execute_from_pandas() (*aequilibrae.paths.route_choice.RouteChoice method*), 333
execute_from_pandas() (*aequilibrae.paths.RouteChoice method*), 297
execute_from_path_files() (*aequilibrae.paths.route_choice.RouteChoice method*), 333
execute_from_path_files() (*aequilibrae.paths.RouteChoice method*), 297
execute_import() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
execute_single() (*aequilibrae.paths.route_choice.RouteChoice method*), 333
execute_single() (*aequilibrae.paths.RouteChoice method*), 297
executemany() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
executescript() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
export() (*aequilibrae.AequilibraeMatrix method*), 242
export() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 284
export() (*aequilibrae.matrix.AequilibraeMatrix method*), 276
export_convergence_csv() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 459
export_to_gmns() (*aequilibrae.project.Network*

method), 352
export_to_gmns() (*aequilibrae.project.network.Network method*), 377
export_to_gmns() (*aequilibrae.project.network.network.Network method*), 393
extent() (*aequilibrae.project.basic_table.BasicTable method*), 362
extent() (*aequilibrae.project.Network method*), 352
extent() (*aequilibrae.project.network.Links method*), 373
extent() (*aequilibrae.project.network.links.Links method*), 389
extent() (*aequilibrae.project.network.Network method*), 377
extent() (*aequilibrae.project.network.network.Network method*), 393
extent() (*aequilibrae.project.network.Nodes method*), 379
extent() (*aequilibrae.project.network.nodes.Nodes method*), 396
extent() (*aequilibrae.project.network.Periods method*), 381
extent() (*aequilibrae.project.network.periods.Periods method*), 400
extent() (*aequilibrae.project.Periods method*), 354
extent() (*aequilibrae.project.Zoning method*), 359
extent() (*aequilibrae.project.zoning.Zoning method*), 416

F

f32_names (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand attribute*), 287
f32_names (*aequilibrae.matrix.GeneralisedCOODemand attribute*), 280
f64_names (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand attribute*), 287
f64_names (*aequilibrae.matrix.GeneralisedCOODemand attribute*), 280
Fare (*class in aequilibrae.transit.transit_elements*), 433
Fare (*class in aequilibrae.transit.transit_elements.fare*), 437
FareRule (*class in aequilibrae.transit.transit_elements*), 433
FareRule (*class in aequilibrae.transit.transit_elements.fare_rule*), 438
fares (*aequilibrae.transit.constants.Constants attribute*), 422
FieldEditor (*class in aequilibrae.project*), 348
FieldEditor (*class in aequilibrae.project.field_editor*), 369
fields (*aequilibrae.project.basic_table.BasicTable property*), 362
fields (*aequilibrae.project.network.link_types.LinkTypes property*), 388
fields (*aequilibrae.project.network.Links property*), 374
fields (*aequilibrae.project.network.links.Links property*), 390
fields (*aequilibrae.project.network.LinkTypes property*), 373
fields (*aequilibrae.project.network.Modes property*), 375
fields (*aequilibrae.project.network.modes.Modes property*), 392
fields (*aequilibrae.project.network.Nodes property*), 380
fields (*aequilibrae.project.network.nodes.Nodes property*), 396
fields (*aequilibrae.project.network.Periods property*), 381
fields (*aequilibrae.project.network.periods.Periods property*), 401
fields (*aequilibrae.project.Periods property*), 355
fields (*aequilibrae.project.Zoning property*), 360
fields (*aequilibrae.project.zoning.Zoning property*), 417
file (*aequilibrae.project.tools.Migration attribute*), 406
file (*aequilibrae.project.tools.migration_manager.Migration attribute*), 411
file_default (*aequilibrae.Parameters attribute*), 254
file_default (*aequilibrae.parameters.Parameters attribute*), 289
find_applicable() (*aequilibrae.project.tools.migration_manager.MigrationManager method*), 411
find_applicable() (*aequilibrae.project.tools.MigrationManager method*), 406
find_table_fields() (*in module aequilibrae.utils.find_table_fields*), 451
fit() (*aequilibrae.distribution.Ipf method*), 268
fit() (*aequilibrae.distribution.ipf.Ipf method*), 272
fit() (*aequilibrae.Ipf method*), 251
format_date() (*in module aequilibrae.transit.date_tools*), 422
free_flow_tt (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 339
free_flow_tt (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
free_flow_tt (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
free_flow_tt (*aequilibrae.paths.TrafficAssignment attribute*), 306
free_flow_tt (*aequilibrae.paths.TransitAssignment attribute*), 309
from_bytes() (*aequilibrae.project.tools.migration_manager.MigrationStatus class method*), 412
from_bytes() (*aequilibrae.project.tools.MigrationStatus class method*), 407

from_db() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder class method*), 444
from_db() (*aequilibrae.transit.TransitGraphBuilder class method*), 419
from_disk() (*aequilibrae.matrix.COO class method*), 280
from_disk() (*aequilibrae.matrix.Sparse class method*), 280
from_disk() (*aequilibrae.matrix.sparse_matrix.COO class method*), 288
from_disk() (*aequilibrae.matrix.sparse_matrix.Sparse class method*), 288
from_matrix() (*aequilibrae.matrix.COO class method*), 280
from_matrix() (*aequilibrae.matrix.sparse_matrix.COO class method*), 288
from_path() (*aequilibrae.Project class method*), 256
from_path() (*aequilibrae.project.Project class method*), 355
from_path() (*aequilibrae.project.project.Project class method*), 401
from_row() (*aequilibrae.transit.transit_elements.Agency method*), 433
from_row() (*aequilibrae.transit.transit_elements.agency.Agency method*), 437
from_row() (*aequilibrae.transit.transit_elements.basic_element.BasicPTElement method*), 437
from_row() (*aequilibrae.transit.transit_elements.Pattern method*), 434
from_row() (*aequilibrae.transit.transit_elements.pattern.Pattern method*), 439
from_row() (*aequilibrae.transit.transit_elements.Route method*), 435
from_row() (*aequilibrae.transit.transit_elements.route.Route method*), 440
from_row() (*aequilibrae.transit.transit_elements.Stop method*), 435
from_row() (*aequilibrae.transit.transit_elements.stop.Stop method*), 441
from_row() (*aequilibrae.transit.transit_elements.Trip method*), 436
from_row() (*aequilibrae.transit.transit_elements.trip.Trip method*), 442
func_assig_thread() (*aequilibrae.allOrNothing method*), 264
func_assig_thread() (*aequilibrae.paths.all_or_nothing.allOrNothing method*), 314
func_assig_thread() (*aequilibrae.paths.allOrNothing method*), 313
functions_available() (*aequilibrae.paths.VDF method*), 312
functions_available() (*aequilibrae.paths.vdf.VDF method*), 347
functions_available() (*aequilibrae.VDF method*), 264

G

GeneralisedCOODemand (*class in aequilibrae.matrix*), 280
GeneralisedCOODemand (*class in aequilibrae.matrix.coo_demand*), 287
GeoIndex (*class in aequilibrae.utils.geo_index*), 452
geometry_grid() (*in module aequilibrae.project.network.osm.model_area_gridding*), 397
get() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
get() (*aequilibrae.project.network.Links method*), 373
get() (*aequilibrae.project.network.links.Links method*), 389
get() (*aequilibrae.project.network.LinkTypes method*), 373
get() (*aequilibrae.project.network.Modes method*), 375
get() (*aequilibrae.project.network.modes.Modes method*), 391
get() (*aequilibrae.project.network.Nodes method*), 379
get() (*aequilibrae.project.network.nodes.Nodes method*), 396
get() (*aequilibrae.project.network.Periods method*), 381
get() (*aequilibrae.project.network.periods.Periods method*), 400
get() (*aequilibrae.project.Periods method*), 354
get() (*aequilibrae.project.Zoning method*), 359
get() (*aequilibrae.project.zoning.Zoning method*), 416
get_ab_lists() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 385
get_active_project() (*in module aequilibrae.context*), 265
get_aeq_direction() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 385
get_by_name() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
get_by_name() (*aequilibrae.project.network.LinkTypes method*), 373
get_by_name() (*aequilibrae.project.network.Modes method*), 375
get_by_name() (*aequilibrae.project.network.modes.Modes method*), 391
get_closest_zone() (*aequilibrae.project.Zoning method*), 359
get_closest_zone() (*aequilibrae.project.zoning.Zoning method*), 416

get_data() (*aequilibrae.project.data.matrix_record.MatrixRecord method*), 366
get_data() (*aequilibrae.project.data.result_record.ResultRecord method*), 367
get_geo_table() (*in module aequilibrae.utils.get_table*), 452
get_graph_to_network_mapping() (*aequilibrae.AssignmentResults method*), 245
get_graph_to_network_mapping() (*aequilibrae.paths.AssignmentResults method*), 289
get_graph_to_network_mapping() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
get_graph_to_network_mapping() (*aequilibrae.paths.results.AssignmentResults method*), 325
get_heuristics() (*aequilibrae.PathResults method*), 255
get_heuristics() (*aequilibrae.paths.PathResults method*), 296
get_heuristics() (*aequilibrae.paths.results.path_results.PathResults method*), 331
get_heuristics() (*aequilibrae.paths.results.PathResults method*), 327
get_link_field_type() (*aequilibrae.project.network.osm.osm_builder.OSMBuilder static method*), 397
get_link_fields() (*aequilibrae.project.network.osm.osm_builder.OSMBuilder static method*), 397
get_link_id() (*aequilibrae.transit.transit_elements.Link method*), 434
get_link_id() (*aequilibrae.transit.transit_elements.link.Link method*), 439
get_links_bounds_box() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 459
get_load_results() (*aequilibrae.AssignmentResults method*), 246
get_load_results() (*aequilibrae.paths.AssignmentResults method*), 289
get_load_results() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
get_load_results() (*aequilibrae.paths.results.assignment_results.TransitAssignmentResults method*), 330
get_load_results() (*aequilibrae.paths.results.AssignmentResults method*), 325
get_load_results() (*aequilibrae.paths.results.TransitAssignmentResults method*), 328
get_load_results() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
get_load_results() (*aequilibrae.paths.RouteChoice method*), 297
get_load_results() (*aequilibrae.paths.TransitAssignmentResults method*), 309
get_log_handler() (*in module aequilibrae.log*), 273
get_logger() (*in module aequilibrae.context*), 265
get_matrix() (*aequilibrae.AequilibraeMatrix method*), 242
get_matrix() (*aequilibrae.Matrices method*), 252
get_matrix() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 284
get_matrix() (*aequilibrae.matrix.AequilibraeMatrix method*), 277
get_matrix() (*aequilibrae.project.data.Matrices method*), 362
get_matrix() (*aequilibrae.project.data.matrices.Matrices method*), 365
get_matrix() (*aequilibrae.project.Matrices method*), 350
get_node_id() (*aequilibrae.transit.transit_elements.Stop method*), 435
get_node_id() (*aequilibrae.transit.transit_elements.stop.Stop method*), 441
get_osm_filter() (*aequilibrae.project.network.osm.osm_downloader.OSMDownloader method*), 398
get_path_for_destination() (*aequilibrae.AssignmentPaths method*), 245
get_path_for_destination() (*aequilibrae.paths.assignment_paths.AssignmentPaths method*), 314
get_path_for_destination() (*aequilibrae.paths.AssignmentPaths method*), 289
get_path_for_destination_from_files() (*aequilibrae.AssignmentPaths static method*), 245
get_path_for_destination_from_files() (*aequilibrae.paths.assignment_paths.AssignmentPaths static method*), 314
get_path_for_destination_from_files() (*aequilibrae.paths.AssignmentPaths static method*), 289
get_project_center() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 460
get_project_zoom() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 460
get_record() (*aequilibrae.Matrices method*), 252
get_record() (*aequilibrae.project.data.Matrices method*), 362
get_record() (*aequili-*

brae.project.data.matrices.Matrices method), 365
get_record() (*aequilibrae.project.data.Results method*), 363
get_record() (*aequilibrae.project.data.results.Results method*), 368
get_record() (*aequilibrae.project.Matrices method*), 350
get_results() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
get_results() (*aequilibrae.paths.RouteChoice method*), 297
get_results() (*aequilibrae.project.data.Results method*), 363
get_results() (*aequilibrae.project.data.results.Results method*), 368
get_schema() (*in module aequilibrae.utils.db_utils*), 451
get_select_link_loading_results() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
get_select_link_loading_results() (*aequilibrae.paths.RouteChoice method*), 298
get_select_link_od_matrix_results() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
get_select_link_od_matrix_results() (*aequilibrae.paths.RouteChoice method*), 298
get_skim_results() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
get_skim_results() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
get_skim_results() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
get_skim_results() (*aequilibrae.paths.TrafficAssignment method*), 303
get_skim_results() (*aequilibrae.paths.TransitAssignment method*), 308
get_skim_results() (*aequilibrae.TrafficAssignment method*), 260
get_sl_results() (*aequilibrae.AssignmentResults method*), 246
get_sl_results() (*aequilibrae.paths.AssignmentResults method*), 290
get_sl_results() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
get_sl_results() (*aequilibrae.paths.results.AssignmentResults method*), 325
get_srid() (*in module aequilibrae.transit.functions.get_srid*), 424
get_table() (*aequilibrae.transit.Transit method*), 417
get_table() (*aequilibrae.transit.transit.Transit method*), 432
get_table() (*in module aequilibrae.utils.get_table*), 453
get_traffic_class_names_and_id() (*aequilibrae.paths.assignment_paths.AssignmentResultsTable method*), 315
get_trip_id() (*aequilibrae.transit.transit_elements.Trip method*), 436
get_trip_id() (*aequilibrae.transit.transit_elements.trip.Trip method*), 442
getlimit() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
GMNSBuilder (*class in aequilibrae.project.network.gmns_builder*), 384
GMNSExporter (*class in aequilibrae.project.network.gmns_exporter*), 385
Graph (*class in aequilibrae*), 246
Graph (*class in aequilibrae.paths*), 291
Graph (*class in aequilibrae.paths.graph*), 316
graph_ab_idx (*aequilibrae.paths.graph.NetworkGraphIndices attribute*), 320
graph_ba_idx (*aequilibrae.paths.graph.NetworkGraphIndices attribute*), 320
GraphBase (*class in aequilibrae.paths.graph*), 318
graphs (*aequilibrae.project.network.Network attribute*), 377
graphs (*aequilibrae.project.network.network.Network attribute*), 394
graphs (*aequilibrae.transit.Transit attribute*), 418
graphs (*aequilibrae.transit.transit.Transit attribute*), 432
GravityApplication (*class in aequilibrae*), 248
GravityApplication (*class in aequilibrae.distribution*), 265
GravityApplication (*class in aequilibrae.distribution.gravity_application*), 268
GravityCalibration (*class in aequilibrae*), 250
GravityCalibration (*class in aequilibrae.distribution*), 266
GravityCalibration (*class in aequilibrae.distribution.gravity_calibration*), 270
GTFSReader (*class in aequilibrae.transit.gtfs_loader*), 424
GTFSRouteSystemBuilder (*class in aequilibrae.transit.lib_gtfs*), 426

H

has_column() (*in module aequilibrae.utils.db_utils*), 451
has_table() (*in module aequilibrae.utils.db_utils*), 451
haversine() (*in module aequilibrae.project.network.haversine*), 385
haversine() (*in module aequilibrae.utils.geo_utils*), 452

HyperpathGenerating (*class in aequilibrae.paths*), 293

I

id (*aequilibrae.project.tools.Migration attribute*), 406
id (*aequilibrae.project.tools.migration_manager.Migration attribute*), 411
idx (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451
imag (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
imag (*aequilibrae.project.tools.MigrationStatus attribute*), 408
import_file_as_module() (*in module aequilibrae.utils.model_run_utils*), 453
importing_network() (*aequilibrae.project.network.osm.osm_builder.OSMBuilder method*), 397
in_transaction (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
info (*aequilibrae.paths.traffic_class.TrafficClass property*), 347
info (*aequilibrae.paths.traffic_class.TransitClass property*), 347
info (*aequilibrae.paths.traffic_class.TransportClassBase property*), 347
info (*aequilibrae.paths.TrafficClass property*), 307
info (*aequilibrae.paths.TransitClass property*), 310
info (*aequilibrae.TrafficClass property*), 264
info() (*aequilibrae.paths.HyperpathGenerating method*), 294
info() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
info() (*aequilibrae.paths.RouteChoice method*), 298
info() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
info() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 340
info() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
info() (*aequilibrae.paths.TrafficAssignment method*), 303
info() (*aequilibrae.paths.TransitAssignment method*), 308
info() (*aequilibrae.TrafficAssignment method*), 260
initialize_graph() (*aequilibrae.transit.route_map_matcher.RouteMapMatcher method*), 429
initialize_tables() (*in module aequilibrae.project.project_creation*), 404
insert() (*aequilibrae.utils.geo_index.GeoIndex method*), 452
installation, 1
IntegrityError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
InterfaceError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
InternalError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
interrupt() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
Ipf (*class in aequilibrae*), 250
Ipf (*class in aequilibrae.distribution*), 267
Ipf (*class in aequilibrae.distribution.ipf*), 271
is_empty() (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand method*), 287
is_empty() (*aequilibrae.matrix.GeneralisedCOODemand method*), 280
is_not_windows() (*in module aequilibrae.utils.spatialite_utils*), 460
is_omx() (*aequilibrae.AequilibraeMatrix method*), 243
is_omx() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 284
is_omx() (*aequilibrae.matrix.AequilibraeMatrix method*), 277
is_pk (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451
is_spatialite() (*in module aequilibrae.utils.spatialite_utils*), 460
is_windows() (*in module aequilibrae.utils.spatialite_utils*), 460
isolation_level (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
iterdump() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449

K

k_nearest() (*in module aequilibrae.project.network.connector_creation*), 383
k_nearest_in_zone() (*in module aequilibrae.project.network.connector_creation*), 383

L

LinearApproximation (*class in aequilibrae.paths.linear_approximation*), 323
Link (*class in aequilibrae.project.network*), 370
Link (*class in aequilibrae.project.network.link*), 385
Link (*class in aequilibrae.transit.transit_elements*), 434
Link (*class in aequilibrae.transit.transit_elements.link*), 438
link_types (*aequilibrae.project.Network attribute*), 352
link_types (*aequilibrae.project.network.Network attribute*), 377

link_types (*aequilibrae.project.network.network.Network attribute*), 394
links (*aequilibrae.transit.constants.Constants attribute*), 422
Links (*class in aequilibrae.project.network*), 373
Links (*class in aequilibrae.project.network.links*), 389
LinkType (*class in aequilibrae.project.network*), 371
LinkType (*class in aequilibrae.project.network.link_type*), 387
LinkTypes (*class in aequilibrae.project.network*), 372
LinkTypes (*class in aequilibrae.project.network.link_types*), 387
list() (*aequilibrae.Matrices method*), 252
list() (*aequilibrae.project.data.Matrices method*), 362
list() (*aequilibrae.project.data.matrices.Matrices method*), 365
list() (*aequilibrae.project.data.Results method*), 364
list() (*aequilibrae.project.data.results.Results method*), 368
list() (*aequilibrae.project.Matrices method*), 350
list_columns() (*in module aequilibrae.utils.db_utils*), 451
list_examples() (*in module aequilibrae.utils.create_example*), 447
list_fields() (*aequilibrae.project.About method*), 348
list_fields() (*aequilibrae.project.about.About method*), 361
list_functions() (*in module aequilibrae.project.tools.cli*), 409
list_modes() (*aequilibrae.project.Network method*), 352
list_modes() (*aequilibrae.project.network.Network method*), 377
list_modes() (*aequilibrae.project.network.network.Network method*), 393
list_scenarios() (*aequilibrae.Project method*), 256
list_scenarios() (*aequilibrae.project.Project method*), 355
list_scenarios() (*aequilibrae.project.project.Project method*), 402
list_tables_in_db() (*in module aequilibrae.utils.db_utils*), 451
list_tables_in_db() (*in module aequilibrae.utils.list_tables_in_db*), 453
load() (*aequilibrae.AequilibraeMatrix method*), 243
load() (*aequilibrae.distribution.synthetic_gravity_model.SyntheticGravityModel method*), 272
load() (*aequilibrae.distribution.SyntheticGravityModel method*), 268
load() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 284
load() (*aequilibrae.matrix.AequilibraeMatrix method*), 277
load() (*aequilibrae.SyntheticGravityModel method*), 258
load() (*aequilibrae.transit.Transit method*), 417
load() (*aequilibrae.transit.transit.Transit method*), 432
load_data() (*aequilibrae.transit.gtfs_loader.GTFSReader method*), 424
load_date() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
load_default() (*aequilibrae.Parameters class method*), 254
load_default() (*aequilibrae.parameters.Parameters class method*), 289
load_extension() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
load_from_disk() (*aequilibrae.Graph method*), 247
load_from_disk() (*aequilibrae.paths.Graph method*), 292
load_from_disk() (*aequilibrae.paths.graph.Graph method*), 317
load_from_disk() (*aequilibrae.paths.graph.GraphBase method*), 319
load_from_disk() (*aequilibrae.paths.graph.TransitGraph method*), 322
load_from_disk() (*aequilibrae.paths.TransitGraph method*), 311
load_route_system() (*aequilibrae.transit.route_system.RouteSystem method*), 429
load_spatialite_extension() (*in module aequilibrae.utils.spatialite_utils*), 460
load_structure() (*aequilibrae.project.table_loader.TableLoader method*), 405
load_table() (*aequilibrae.project.data_loader.DataLoader method*), 369
load_table() (*aequilibrae.project.table_loader.TableLoader method*), 405
Log (*class in aequilibrae*), 251
Log (*class in aequilibrae.log*), 272
Log (*class in aequilibrae.project*), 349
log() (*aequilibrae.Project method*), 256
log() (*aequilibrae.project.Project method*), 355
log() (*aequilibrae.project.project.Project method*), 402
log_specification() (*aequilibrae.paths.route_choice.RouteChoice method*), 334
log_specification() (*aequilibrae.paths.RouteChoice method*), 298
log_specification() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338

log_specification() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
log_specification() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
log_specification() (*aequilibrae.paths.TrafficAssignment method*), 303
log_specification() (*aequilibrae.paths.TransitAssignment method*), 308
log_specification() (*aequilibrae.TrafficAssignment method*), 260
logger (*aequilibrae.Project property*), 257
logger (*aequilibrae.project.Project property*), 356
logger (*aequilibrae.project.project.Project property*), 403
logger (*aequilibrae.project.Scenario attribute*), 357
logger (*aequilibrae.project.scenario.Scenario attribute*), 404
lonlat (*aequilibrae.project.network.Nodes property*), 380
lonlat (*aequilibrae.project.network.nodes.Nodes property*), 396

M

main() (*in module aequilibrae.utils.simwrapper.generate_simwrapper_config*), 455
manual_transaction() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
map_match() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
map_match() (*aequilibrae.transit.transit_elements.Pattern method*), 434
map_match() (*aequilibrae.transit.transit_elements.pattern.Pattern method*), 439
map_match_route() (*aequilibrae.transit.route_map_matcher.RouteMapMatcher method*), 429
mark_all_as_seen() (*aequilibrae.project.tools.migration_manager.MigrationManager method*), 411
mark_all_as_seen() (*aequilibrae.project.tools.MigrationManager method*), 406
mark_as() (*aequilibrae.project.tools.Migration method*), 405
mark_as() (*aequilibrae.project.tools.migration_manager.Migration method*), 410
mark_as_seen() (*aequilibrae.project.tools.Migration method*), 406
mark_as_seen() (*aequilibrae.project.tools.migration_manager.Migration method*), 410
matrices (*aequilibrae.Project property*), 257
matrices (*aequilibrae.project.Project property*), 356
matrices (*aequilibrae.project.project.Project property*), 403
matrices (*aequilibrae.project.Scenario attribute*), 357
matrices (*aequilibrae.project.scenario.Scenario attribute*), 405
Matrices (*class in aequilibrae*), 252
Matrices (*class in aequilibrae.project*), 349
Matrices (*class in aequilibrae.project.data*), 362
Matrices (*class in aequilibrae.project.data.matrices*), 365
MatrixRecord (*class in aequilibrae.project.data.matrix_record*), 366
maybe_transform_srid() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 385
metre_crs_for_gdf() (*in module aequilibrae.utils.geo_utils*), 452
Migration (*class in aequilibrae.project.tools*), 405
Migration (*class in aequilibrae.project.tools.migration_manager*), 410
MigrationManager (*class in aequilibrae.project.tools*), 406
MigrationManager (*class in aequilibrae.project.tools.migration_manager*), 411
migrations (*aequilibrae.project.tools.migration_manager.MigrationManager attribute*), 412
migrations (*aequilibrae.project.tools.MigrationManager attribute*), 407
MigrationStatus (*class in aequilibrae.project.tools*), 407
MigrationStatus (*class in aequilibrae.project.tools.migration_manager*), 412
MISSING (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
MISSING (*aequilibrae.project.tools.MigrationStatus attribute*), 408
Mode (*class in aequilibrae.project.network*), 374
Mode (*class in aequilibrae.project.network.mode*), 390
Modes (*class in aequilibrae.project.network*), 374
Modes (*class in aequilibrae.project.network.modes*), 390
module
    aequilibrae, 239
    aequilibrae.context, 265
    aequilibrae.distribution, 265
    aequilibrae.distribution.gravity_application, 268
    aequilibrae.distribution.gravity_calibration, 270
    aequilibrae.distribution.ipf, 271
    aequilibrae.distribution.ipf_core, 272
    aequilibrae.distribution.synthetic_gravity_model, 272
    aequilibrae.log, 272

aequilibrae.matrix, 273
aequilibrae.matrix.aequilibrae_matrix, 281
aequilibrae.matrix.coo_demand, 287
aequilibrae.matrix.sparse_matrix, 288
aequilibrae.parameters, 288
aequilibrae.paths, 289
aequilibrae.paths.all_or_nothing, 314
aequilibrae.paths.AoN, 314
aequilibrae.paths.assignment_paths, 314
aequilibrae.paths.connectivity_analysis, 315
aequilibrae.paths.graph, 316
aequilibrae.paths.graph_building, 323
aequilibrae.paths.linear_approximation, 323
aequilibrae.paths.multi_threaded_aon, 323
aequilibrae.paths.multi_threaded_paths, 323
aequilibrae.paths.multi_threaded_skimming, 324
aequilibrae.paths.network_skimming, 324
aequilibrae.paths.optimal_strategies, 325
aequilibrae.paths.public_transport, 325
aequilibrae.paths.results, 325
aequilibrae.paths.results.assignment_results, 329
aequilibrae.paths.results.path_results, 330
aequilibrae.paths.results.skim_results, 332
aequilibrae.paths.route_choice, 333
aequilibrae.paths.sub_area, 337
aequilibrae.paths.traffic_assignment, 337
aequilibrae.paths.traffic_class, 346
aequilibrae.paths.vdf, 347
aequilibrae.project, 347
aequilibrae.project.about, 360
aequilibrae.project.basic_table, 362
aequilibrae.project.data, 362
aequilibrae.project.data_loader, 369
aequilibrae.project.database_connection, 369
aequilibrae.project.data.matrices, 365
aequilibrae.project.data.matrix_record, 366
aequilibrae.project.data.result_record, 366
aequilibrae.project.data.results, 367
aequilibrae.project.field_editor, 369
aequilibrae.project.network, 370
aequilibrae.project.network.connector_creation, 382
aequilibrae.project.network.gmns_builder, 384
aequilibrae.project.network.gmns_exporter, 385
aequilibrae.project.network.haversine, 385
aequilibrae.project.network.link, 385
aequilibrae.project.network.link_type, 387
aequilibrae.project.network.link_types, 387
aequilibrae.project.network.links, 389
aequilibrae.project.network.mode, 390
aequilibrae.project.network.modes, 390
aequilibrae.project.network.network, 392
aequilibrae.project.network.node, 394
aequilibrae.project.network.nodes, 395
aequilibrae.project.network.osm, 397
aequilibrae.project.network.osm.model_area_gridding, 397
aequilibrae.project.network.osm.osm_builder, 397
aequilibrae.project.network.osm.osm_downloader, 398
aequilibrae.project.network.osm.osm_params, 398
aequilibrae.project.network.osm.place_getter, 398
aequilibrae.project.network.period, 399
aequilibrae.project.network.periods, 399
aequilibrae.project.network.safe_class, 401
aequilibrae.project.project, 401
aequilibrae.project.project_cleaning, 403
aequilibrae.project.project_creation, 404
aequilibrae.project.scenario, 404
aequilibrae.project.table_loader, 405
aequilibrae.project.tools, 405
aequilibrae.project.tools.cli, 409
aequilibrae.project.tools.migration_manager, 410
aequilibrae.project.tools.network_simplifier, 413
aequilibrae.project.zone, 414
aequilibrae.project.zoning, 415
aequilibrae.reference_files, 417
aequilibrae.transit, 417
aequilibrae.transit.column_order, 422
aequilibrae.transit.constants, 422
aequilibrae.transit.date_tools, 422

aequilibrae.transit.functions, 423
aequilibrae.transit.functions.breaking_links_at_stops, 423
aequilibrae.transit.functions.compute_line_bearing, 424
aequilibrae.transit.functions.get_srid, 424
aequilibrae.transit.gtfs_loader, 424
aequilibrae.transit.gtfs_writer, 425
aequilibrae.transit.gtfs_writer.agency_writer, 425
aequilibrae.transit.gtfs_writer.fare_writer, 425
aequilibrae.transit.gtfs_writer.routes_writer, 425
aequilibrae.transit.gtfs_writer.shape_writer, 426
aequilibrae.transit.gtfs_writer.stop_times_writer, 426
aequilibrae.transit.gtfs_writer.stops_writer, 426
aequilibrae.transit.gtfs_writer.trips_writer, 426
aequilibrae.transit.lib_gtfs, 426
aequilibrae.transit.parse_csv, 428
aequilibrae.transit.route_map_matcher, 428
aequilibrae.transit.route_system, 429
aequilibrae.transit.route_system_reader, 429
aequilibrae.transit.route_system_reader.agency_reader, 430
aequilibrae.transit.route_system_reader.pattern_reader, 430
aequilibrae.transit.route_system_reader.routes_reader, 430
aequilibrae.transit.route_system_reader.stop_reader, 430
aequilibrae.transit.route_system_reader.stop_times_reader, 430
aequilibrae.transit.route_system_reader.trips_reader, 431
aequilibrae.transit.transit, 431
aequilibrae.transit.transit_elements, 433
aequilibrae.transit.transit_elements.agency, 437
aequilibrae.transit.transit_elements.basic_element, 437
aequilibrae.transit.transit_elements.fare, 437
aequilibrae.transit.transit_elements.fare_rule, 438
aequilibrae.transit.transit_elements.link, 438
aequilibrae.transit.transit_elements.mode_correspondence, 439
aequilibrae.transit.transit_elements.pattern, 439
aequilibrae.transit.transit_elements.route, 440
aequilibrae.transit.transit_elements.service, 440
aequilibrae.transit.transit_elements.stop, 441
aequilibrae.transit.transit_elements.trip, 441
aequilibrae.transit.transit_graph_builder, 442
aequilibrae.utils, 446
aequilibrae.utils.aeq_signal, 446
aequilibrae.utils.core_setter, 446
aequilibrae.utils.create_delaunay_network, 446
aequilibrae.utils.create_example, 447
aequilibrae.utils.db_utils, 447
aequilibrae.utils.find_table_fields, 451
aequilibrae.utils.geo_index, 451
aequilibrae.utils.geo_utils, 452
aequilibrae.utils.get_table, 452
aequilibrae.utils.interface, 453
aequilibrae.utils.interface.worker_thread, 453
aequilibrae.utils.list_tables_in_db, 453
aequilibrae.utils.model_run_utils, 453
aequilibrae.utils.python_signal, 454
aequilibrae.utils.qgis_utils, 454
aequilibrae.utils.simwrapper, 454
aequilibrae.utils.simwrapper.generate_simwrapper_config, 454
aequilibrae.utils.simwrapper.simwrapper_panel, 455
aequilibrae.utils.simwrapper.simwrapper_utils, 459
aequilibrae.utils.spatialite_utils, 460
MultiThreadedAoN (*class in aequilibrae.paths*), 294
MultiThreadedAoN (*class in aequilibrae.paths.multi_threaded_aon*), 323
MultiThreadedNetworkSkimming (*class in aequilibrae.paths*), 294
MultiThreadedNetworkSkimming (*class in aequilibrae.paths.multi_threaded_skimming*), 324
MultiThreadedPaths (*class in aequilibrae.paths.multi_threaded_paths*), 324

N

name (*aequilibrae.project.Scenario attribute*), 357
name (*aequilibrae.project.scenario.Scenario attribute*), 405

name (*aequilibrae.project.tools.Migration attribute*), 406
name (*aequilibrae.project.tools.migration_manager.Migration attribute*), 411
name (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451
nan_to_num() (*aequilibrae.AequilibraeMatrix method*), 243
nan_to_num() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 285
nan_to_num() (*aequilibrae.matrix.AequilibraeMatrix method*), 277
nearest() (*aequilibrae.utils.geo_index.GeoIndex method*), 452
network (*aequilibrae.Project property*), 257
network (*aequilibrae.project.Project property*), 356
network (*aequilibrae.project.project.Project property*), 403
network (*aequilibrae.project.Scenario attribute*), 357
network (*aequilibrae.project.scenario.Scenario attribute*), 405
Network (*class in aequilibrae.project*), 350
Network (*class in aequilibrae.project.network*), 375
Network (*class in aequilibrae.project.network.network*), 392
network_ab_idx (*aequilibrae.paths.graph.NetworkGraphIndices attribute*), 320
network_ba_idx (*aequilibrae.paths.graph.NetworkGraphIndices attribute*), 320
network_migration_file (*aequilibrae.project.tools.migration_manager.MigrationManager attribute*), 412
network_migration_file (*aequilibrae.project.tools.MigrationManager attribute*), 407
NetworkGraphIndices (*class in aequilibrae.paths.graph*), 320
NetworkSimplifier (*class in aequilibrae.project*), 353
NetworkSimplifier (*class in aequilibrae.project.tools*), 409
NetworkSimplifier (*class in aequilibrae.project.tools.network_simplifier*), 413
NetworkSkimming (*class in aequilibrae*), 252
NetworkSkimming (*class in aequilibrae.paths*), 294
NetworkSkimming (*class in aequilibrae.paths.network_skimming*), 324
new() (*aequilibrae.Project method*), 257
new() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
new() (*aequilibrae.project.network.Links method*), 374
new() (*aequilibrae.project.network.links.Links method*), 389
new() (*aequilibrae.project.network.LinkTypes method*), 373
new() (*aequilibrae.project.network.Modes method*), 375
new() (*aequilibrae.project.network.modes.Modes method*), 391
new() (*aequilibrae.project.Project method*), 356
new() (*aequilibrae.project.project.Project method*), 402
new() (*aequilibrae.project.Zoning method*), 360
new() (*aequilibrae.project.zoning.Zoning method*), 416
new_centroid() (*aequilibrae.project.network.Nodes method*), 379
new_centroid() (*aequilibrae.project.network.nodes.Nodes method*), 396
new_gtfs_builder() (*aequilibrae.transit.Transit method*), 418
new_gtfs_builder() (*aequilibrae.transit.transit.Transit method*), 432
new_period() (*aequilibrae.project.network.Periods method*), 381
new_period() (*aequilibrae.project.network.periods.Periods method*), 400
new_period() (*aequilibrae.project.Periods method*), 354
new_record() (*aequilibrae.Matrices method*), 252
new_record() (*aequilibrae.project.data.Matrices method*), 362
new_record() (*aequilibrae.project.data.matrices.Matrices method*), 365
new_record() (*aequilibrae.project.data.Results method*), 364
new_record() (*aequilibrae.project.data.results.Results method*), 368
new_record() (*aequilibrae.project.Matrices method*), 350
no_demand() (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand method*), 287
no_demand() (*aequilibrae.matrix.GeneralisedCOODemand method*), 280
Node (*class in aequilibrae.project.network*), 378
Node (*class in aequilibrae.project.network.node*), 394
Nodes (*class in aequilibrae.project.network*), 379
Nodes (*class in aequilibrae.project.network.nodes*), 395
nodes_to_indices (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand attribute*), 287
nodes_to_indices (*aequilibrae.matrix.GeneralisedCOODemand attribute*), 280
noop() (*in module aequilibrae.utils.aeq_signal*), 446
normalise_mode_strings() (*in module aequilibrae.project.network.connector_creation*), 384
not_null (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451

NotSupportedError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
numerator (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
numerator (*aequilibrae.project.tools.MigrationStatus attribute*), 408

O

one_day_before() (*in module aequilibrae.transit.date_tools*), 422
one_to_all() (*in module aequilibrae.paths*), 313
open() (*aequilibrae.Project method*), 257
open() (*aequilibrae.project.Project method*), 356
open() (*aequilibrae.project.project.Project method*), 402
OperationalError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
OptimalStrategies (*class in aequilibrae.paths*), 295
OptimalStrategies (*class in aequilibrae.paths.optimal_strategies*), 325
OSMBuilder (*class in aequilibrae.project.network.osm.osm_builder*), 397
OSMDownloader (*class in aequilibrae.project.network.osm.osm_downloader*), 398
overpass_request() (*aequilibrae.project.network.osm.osm_downloader.OSMDownloader method*), 398

P

parameters (*aequilibrae.Project property*), 258
parameters (*aequilibrae.project.Project property*), 357
parameters (*aequilibrae.project.project.Project property*), 403
Parameters (*class in aequilibrae*), 253
Parameters (*class in aequilibrae.parameters*), 288
parse_convergence_json() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 460
parse_csv() (*in module aequilibrae.transit.parse_csv*), 428
path_computation() (*in module aequilibrae.paths*), 313
path_to_file (*aequilibrae.Project property*), 258
path_to_file (*aequilibrae.project.Project property*), 357
path_to_file (*aequilibrae.project.project.Project property*), 403
path_to_file (*aequilibrae.project.Scenario attribute*), 357
path_to_file (*aequilibrae.project.scenario.Scenario attribute*), 405
PathResults (*class in aequilibrae*), 254
PathResults (*class in aequilibrae.paths*), 295
PathResults (*class in aequilibrae.paths.results*), 326
PathResults (*class in aequilibrae.paths.results.path_results*), 330
Pattern (*class in aequilibrae.transit.transit_elements*), 434
PatternStatus (*class in aequilibrae.transit.transit_elements.pattern*), 439
pattern_lookup (*aequilibrae.transit.constants.Constants attribute*), 422
patterns (*aequilibrae.transit.constants.Constants attribute*), 422
Period (*class in aequilibrae.project*), 353
Period (*class in aequilibrae.project.network*), 380
Period (*class in aequilibrae.project.network.period*), 399
Periods (*class in aequilibrae.project*), 354
Periods (*class in aequilibrae.project.network*), 380
Periods (*class in aequilibrae.project.network.periods*), 400
placegetter() (*in module aequilibrae.project.network.osm*), 397
placegetter() (*in module aequilibrae.project.network.osm.place_getter*), 398
populate() (*aequilibrae.transit.transit_elements.Fare method*), 433
populate() (*aequilibrae.transit.transit_elements.fare_rule.FareRule method*), 438
populate() (*aequilibrae.transit.transit_elements.fare.Fare method*), 438
populate() (*aequilibrae.transit.transit_elements.FareRule method*), 433
populate() (*aequilibrae.transit.transit_elements.Route method*), 435
populate() (*aequilibrae.transit.transit_elements.route.Route method*), 440
post_process() (*aequilibrae.paths.sub_area.SubAreaAnalysis method*), 337
post_process() (*aequilibrae.paths.SubAreaAnalysis method*), 301
preloads (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
preloads (*aequilibrae.paths.TrafficAssignment attribute*), 306
prepare() (*aequilibrae.AssignmentResults method*), 246
prepare() (*aequilibrae.PathResults method*), 255
prepare() (*aequilibrae.paths.AssignmentResults method*), 290
prepare() (*aequilibrae.paths.multi_threaded_aon.MultiThreadedAoN method*), 323
prepare() (*aequilibrae.paths.multi_threaded_skimming.MultiThreadedNetw method*), 324
prepare() (*aequilibrae.paths.MultiThreadedAoN method*), 294
prepare() (*aequilibrae.paths.MultiThreadedNetworkSkimming method*), 294
prepare() (*aequilibrae.paths.PathResults method*), 296

prepare() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
prepare() (*aequilibrae.paths.results.assignment_results.AssignmentResultsBase method*), 329
prepare() (*aequilibrae.paths.results.assignment_results.TransitAssignmentResults method*), 330
prepare() (*aequilibrae.paths.results.AssignmentResults method*), 325
prepare() (*aequilibrae.paths.results.path_results.PathResults method*), 331
prepare() (*aequilibrae.paths.results.PathResults method*), 327
prepare() (*aequilibrae.paths.results.skim_results.SkimResults method*), 332
prepare() (*aequilibrae.paths.results.SkimResults method*), 328
prepare() (*aequilibrae.paths.results.TransitAssignmentResults method*), 328
prepare() (*aequilibrae.paths.route_choice.RouteChoice method*), 335
prepare() (*aequilibrae.paths.RouteChoice method*), 298
prepare() (*aequilibrae.paths.SkimResults method*), 301
prepare() (*aequilibrae.paths.TransitAssignmentResults method*), 310
prepare() (*aequilibrae.SkimResults method*), 258
prepare_() (*aequilibrae.paths.multi_threaded_paths.MultiThreadedPaths method*), 324
prepare_() (*aequilibrae.paths.multi_threaded_skimming.MultiThreadedNetworkSkimming method*), 324
prepare_() (*aequilibrae.paths.MultiThreadedNetworkSkimming method*), 294
prepare_graph() (*aequilibrae.Graph method*), 247
prepare_graph() (*aequilibrae.paths.Graph method*), 292
prepare_graph() (*aequilibrae.paths.graph.Graph method*), 317
prepare_graph() (*aequilibrae.paths.graph.GraphBase method*), 319
prepare_graph() (*aequilibrae.paths.graph.TransitGraph method*), 322
prepare_graph() (*aequilibrae.paths.TransitGraph method*), 311
pretty_round() (*in module aequilibrae.utils.simwrapper.simwrapper_utils*), 460
ProgrammingError (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
Project (*class in aequilibrae*), 256
Project (*class in aequilibrae.project*), 355
Project (*class in aequilibrae.project.project*), 401
project_base_path (*aequilibrae.Project property*), 258
project_base_path (*aequilibrae.project.Project property*), 357
project_base_path (*aequilibrae.project.project.Project property*), 403
project_parameters (*aequilibrae.Project property*), 258
project_parameters (*aequilibrae.project.Project property*), 357
project_parameters (*aequilibrae.project.project.Project property*), 403
protected_fields (*aequilibrae.project.Network attribute*), 352
protected_fields (*aequilibrae.project.network.Network attribute*), 377
protected_fields (*aequilibrae.project.network.network.Network attribute*), 394
pt_con (*aequilibrae.transit.Transit attribute*), 418
pt_con (*aequilibrae.transit.transit.Transit attribute*), 432
PythonSignal (*class in aequilibrae.utils.python_signal*), 454

R

random_name() (*aequilibrae.AequilibraeMatrix static method*), 239
random_name() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix static method*), 281
random_name() (*aequilibrae.matrix.AequilibraeMatrix static method*), 273
read_agencies() (*in module aequilibrae.transit.route_system_reader*), 429
read_agencies() (*in module aequilibrae.transit.route_system_reader.agency_reader*), 430
read_and_close() (*in module aequilibrae.utils.db_utils*), 451
read_path_file() (*aequilibrae.AssignmentPaths method*), 245
read_path_file() (*aequilibrae.paths.assignment_paths.AssignmentPaths method*), 315
read_path_file() (*aequilibrae.paths.AssignmentPaths method*), 289
read_patterns() (*in module aequilibrae.transit.route_system_reader*), 429
read_patterns() (*in module aequilibrae.transit.route_system_reader.pattern_reader*), 430
read_routes() (*in module aequilibrae.transit.route_system_reader*), 429
read_routes() (*in module aequilibrae.transit.route_system_reader.routes_reader*), 430
read_sql() (*in module aequilibrae.utils.db_utils*), 451
read_stop_times() (*in module aequilibrae.transit.route_system_reader*), 429

read_stop_times() (*in module aequilibrae.transit.route_system_reader.stop_times_reader*), 431
read_stops() (*in module aequilibrae.transit.route_system_reader*), 429
read_stops() (*in module aequilibrae.transit.route_system_reader.stop_reader*), 430
read_trips() (*in module aequilibrae.transit.route_system_reader*), 429
read_trips() (*in module aequilibrae.transit.route_system_reader.trips_reader*), 431
real (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
real (*aequilibrae.project.tools.MigrationStatus attribute*), 409
rebuild_network() (*aequilibrae.project.NetworkSimplifier method*), 353
rebuild_network() (*aequilibrae.project.tools.network_simplifier.NetworkSimplifier method*), 414
rebuild_network() (*aequilibrae.project.tools.NetworkSimplifier method*), 409
recreate_columns() (*in module aequilibrae.project.project_creation*), 404
refresh() (*aequilibrae.project.network.Links method*), 374
refresh() (*aequilibrae.project.network.links.Links method*), 390
refresh() (*aequilibrae.project.network.Nodes method*), 379
refresh() (*aequilibrae.project.network.nodes.Nodes method*), 396
refresh() (*aequilibrae.project.network.Periods method*), 381
refresh() (*aequilibrae.project.network.periods.Periods method*), 400
refresh() (*aequilibrae.project.Periods method*), 354
refresh_fields() (*aequilibrae.project.network.Links method*), 374
refresh_fields() (*aequilibrae.project.network.links.Links method*), 390
refresh_fields() (*aequilibrae.project.network.Nodes method*), 379
refresh_fields() (*aequilibrae.project.network.nodes.Nodes method*), 396
refresh_fields() (*aequilibrae.project.network.Periods method*), 381
refresh_fields() (*aequilibrae.project.network.periods.Periods method*), 400
refresh_fields() (*aequilibrae.project.Periods method*), 354
refresh_geo_index() (*aequilibrae.project.Zoning method*), 360
refresh_geo_index() (*aequilibrae.project.zoning.Zoning method*), 416
reload() (*aequilibrae.Matrices method*), 252
reload() (*aequilibrae.project.data.Matrices method*), 363
reload() (*aequilibrae.project.data.matrices.Matrices method*), 366
reload() (*aequilibrae.project.data.Results method*), 364
reload() (*aequilibrae.project.data.results.Results method*), 369
reload() (*aequilibrae.project.Matrices method*), 350
remove() (*aequilibrae.project.field_editor.FieldEditor method*), 370
remove() (*aequilibrae.project.FieldEditor method*), 349
remove() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder class method*), 444
remove() (*aequilibrae.transit.TransitGraphBuilder class method*), 419
remove_config() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder static method*), 444
remove_config() (*aequilibrae.transit.TransitGraphBuilder static method*), 419
remove_edges() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder static method*), 444
remove_edges() (*aequilibrae.transit.TransitGraphBuilder static method*), 419
remove_graphs() (*aequilibrae.transit.Transit method*), 418
remove_graphs() (*aequilibrae.transit.transit.Transit method*), 432
remove_triggers() (*in module aequilibrae.project.project_creation*), 404
remove_vertices() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder static method*), 444
remove_vertices() (*aequilibrae.transit.TransitGraphBuilder static method*), 420
renumber() (*aequilibrae.project.network.Node method*), 379
renumber() (*aequilibrae.project.network.node.Node method*), 395
renumber() (*aequilibrae.project.network.Period method*), 380
renumber() (*aequilibrae.project.network.period.Period method*), 399

renumber() (*aequilibrae.project.Period method*), 353
reorder_fields() (*aequilibrae.project.network.gmns_exporter.GMNSExporter method*), 385
report() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
report() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
report() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
report() (*aequilibrae.paths.TrafficAssignment method*), 303
report() (*aequilibrae.paths.TransitAssignment method*), 308
report() (*aequilibrae.TrafficAssignment method*), 261
req_link_flds (*aequilibrae.project.Network attribute*), 352
req_link_flds (*aequilibrae.project.network.Network attribute*), 378
req_link_flds (*aequilibrae.project.network.network.Network attribute*), 394
req_node_flds (*aequilibrae.project.Network attribute*), 353
req_node_flds (*aequilibrae.project.network.Network attribute*), 378
req_node_flds (*aequilibrae.project.network.network.Network attribute*), 394
reset() (*aequilibrae.AssignmentResults method*), 246
reset() (*aequilibrae.PathResults method*), 255
reset() (*aequilibrae.paths.AssignmentResults method*), 290
reset() (*aequilibrae.paths.PathResults method*), 296
reset() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
reset() (*aequilibrae.paths.results.assignment_results.AssignmentResultsBase method*), 330
reset() (*aequilibrae.paths.results.assignment_results.TransitAssignmentResults method*), 330
reset() (*aequilibrae.paths.results.AssignmentResults method*), 326
reset() (*aequilibrae.paths.results.path_results.PathResults method*), 331
reset() (*aequilibrae.paths.results.PathResults method*), 327
reset() (*aequilibrae.paths.results.TransitAssignmentResults method*), 328
reset() (*aequilibrae.paths.TransitAssignmentResults method*), 310
reset() (*aequilibrae.utils.geo_index.GeoIndex method*), 452
resolve_recusive_dict() (*in module aequilibrae.project.network.gmns_builder*), 385
restore_default() (*aequilibrae.Parameters method*), 254
restore_default() (*aequilibrae.parameters.Parameters method*), 289
ResultRecord (*class in aequilibrae.project.data.result_record*), 366
results (*aequilibrae.Project property*), 258
results (*aequilibrae.project.Project property*), 357
results (*aequilibrae.project.project.Project property*), 403
results (*aequilibrae.project.Scenario attribute*), 357
results (*aequilibrae.project.scenario.Scenario attribute*), 405
Results (*class in aequilibrae.project.data*), 363
Results (*class in aequilibrae.project.data.results*), 367
results() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
results() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
results() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
results() (*aequilibrae.paths.TrafficAssignment method*), 304
results() (*aequilibrae.paths.TransitAssignment method*), 308
results() (*aequilibrae.TrafficAssignment method*), 261
results_connection (*aequilibrae.Project property*), 258
results_connection (*aequilibrae.project.Project property*), 357
results_connection (*aequilibrae.project.project.Project property*), 403
reverse() (*aequilibrae.Graph method*), 248
reverse() (*aequilibrae.paths.Graph method*), 292
reverse() (*aequilibrae.paths.graph.Graph method*), 317
reverse() (*aequilibrae.paths.graph.GraphBase method*), 320
reverse() (*aequilibrae.paths.graph.TransitGraph method*), 322
reverse() (*aequilibrae.paths.TransitGraph method*), 312
rollback() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 449
Route (*class in aequilibrae.transit.transit_elements*), 434
Route (*class in aequilibrae.transit.transit_elements.route*), 440
RouteChoice (*class in aequilibrae.paths*), 296
RouteChoice (*class in aequilibrae.paths.route_choice*), 333
RouteMapMatcher (*class in aequilibrae.transit.route_map_matcher*), 428
routes (*aequilibrae.transit.constants.Constants attribute*), 422
RouteSystem (*class in aequilibrae.transit.route_system*), 429
row_factory (*aequili-*

brae.utils.db_utils.AequilibraEConnection attribute), 450
rows() (*aequilibrae.AequilibraeMatrix method*), 243
rows() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 285
rows() (*aequilibrae.matrix.AequilibraeMatrix method*), 278
run (*aequilibrae.Project property*), 258
run (*aequilibrae.project.Project property*), 357
run (*aequilibrae.project.project.Project property*), 403
run() (*aequilibrae.paths.HyperpathGenerating method*), 294
run() (*in module aequilibrae.project.tools.cli*), 409
run_queries_from_sql_file() (*in module aequilibrae.project.project_creation*), 404

S

safe_connect() (*in module aequilibrae.utils.db_utils*), 451
SafeClass (*class in aequilibrae.project.network.safe_class*), 401
save() (*aequilibrae.AequilibraeMatrix method*), 244
save() (*aequilibrae.distribution.synthetic_gravity_model.SyntheticGravityModel method*), 272
save() (*aequilibrae.distribution.SyntheticGravityModel method*), 268
save() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 286
save() (*aequilibrae.matrix.AequilibraeMatrix method*), 278
save() (*aequilibrae.project.data.matrix_record.MatrixRecord method*), 366
save() (*aequilibrae.project.data.result_record.ResultRecord method*), 367
save() (*aequilibrae.project.field_editor.FieldEditor method*), 370
save() (*aequilibrae.project.FieldEditor method*), 349
save() (*aequilibrae.project.network.Link method*), 371
save() (*aequilibrae.project.network.link_type.LinkType method*), 387
save() (*aequilibrae.project.network.link_types.LinkTypes method*), 388
save() (*aequilibrae.project.network.link.Link method*), 387
save() (*aequilibrae.project.network.Links method*), 374
save() (*aequilibrae.project.network.links.Links method*), 390
save() (*aequilibrae.project.network.LinkType method*), 372
save() (*aequilibrae.project.network.LinkTypes method*), 373
save() (*aequilibrae.project.network.Mode method*), 374
save() (*aequilibrae.project.network.mode.Mode method*), 390
save() (*aequilibrae.project.network.Node method*), 379
save() (*aequilibrae.project.network.node.Node method*), 395
save() (*aequilibrae.project.network.Nodes method*), 380
save() (*aequilibrae.project.network.nodes.Nodes method*), 396
save() (*aequilibrae.project.network.Period method*), 380
save() (*aequilibrae.project.network.period.Period method*), 399
save() (*aequilibrae.project.network.Periods method*), 381
save() (*aequilibrae.project.network.periods.Periods method*), 400
save() (*aequilibrae.project.Period method*), 353
save() (*aequilibrae.project.Periods method*), 354
save() (*aequilibrae.project.Zone method*), 358
save() (*aequilibrae.project.zone.Zone method*), 415
save() (*aequilibrae.project.Zoning method*), 360
save() (*aequilibrae.project.zoning.Zoning method*), 416
save() (*aequilibrae.SyntheticGravityModel method*), 259
save() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
save() (*aequilibrae.transit.TransitGraphBuilder method*), 421
save_compressed_correspondence() (*aequilibrae.Graph method*), 248
save_compressed_correspondence() (*aequilibrae.paths.Graph method*), 292
save_compressed_correspondence() (*aequilibrae.paths.graph.Graph method*), 317
save_compressed_correspondence() (*aequilibrae.paths.graph.GraphBase method*), 320
save_compressed_correspondence() (*aequilibrae.paths.graph.TransitGraph method*), 322
save_compressed_correspondence() (*aequilibrae.paths.TransitGraph method*), 312
save_config() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
save_config() (*aequilibrae.transit.TransitGraphBuilder method*), 421
save_edges() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
save_edges() (*aequilibrae.transit.TransitGraphBuilder method*), 421
save_graphs() (*aequilibrae.transit.Transit method*), 418
save_graphs() (*aequilibrae.transit.transit.Transit method*), 432
save_link_flows() (*aequilibrae.paths.route_choice.RouteChoice method*), 335
save_link_flows() (*aequilibrae.paths.RouteChoice method*), 298
save_modes_to_aeq() (*aequili-*

brae.project.network.gmns_builder.GMNSBuilder method), 385
save_path_files (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
save_path_files (*aequilibrae.paths.TrafficAssignment attribute*), 306
save_path_files() (*aequilibrae.paths.route_choice.RouteChoice method*), 335
save_path_files() (*aequilibrae.paths.RouteChoice method*), 299
save_results() (*aequilibrae.paths.HyperpathGenerating method*), 294
save_results() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
save_results() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
save_results() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
save_results() (*aequilibrae.paths.TrafficAssignment method*), 304
save_results() (*aequilibrae.paths.TransitAssignment method*), 308
save_results() (*aequilibrae.TrafficAssignment method*), 261
save_select_link_flows() (*aequilibrae.paths.route_choice.RouteChoice method*), 335
save_select_link_flows() (*aequilibrae.paths.RouteChoice method*), 299
save_select_link_flows() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
save_select_link_flows() (*aequilibrae.paths.TrafficAssignment method*), 304
save_select_link_flows() (*aequilibrae.TrafficAssignment method*), 261
save_select_link_matrices() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
save_select_link_matrices() (*aequilibrae.paths.TrafficAssignment method*), 304
save_select_link_matrices() (*aequilibrae.TrafficAssignment method*), 261
save_select_link_results() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
save_select_link_results() (*aequilibrae.paths.TrafficAssignment method*), 304
save_select_link_results() (*aequilibrae.TrafficAssignment method*), 261
save_skims() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 341
save_skims() (*aequilibrae.paths.TrafficAssignment method*), 304
save_skims() (*aequilibrae.TrafficAssignment method*), 261
save_to_database() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 385
save_to_database() (*aequilibrae.transit.transit_elements.Agency method*), 433
save_to_database() (*aequilibrae.transit.transit_elements.agency.Agency method*), 437
save_to_database() (*aequilibrae.transit.transit_elements.Fare method*), 433
save_to_database() (*aequilibrae.transit.transit_elements.fare_rule.FareRule method*), 438
save_to_database() (*aequilibrae.transit.transit_elements.fare.Fare method*), 438
save_to_database() (*aequilibrae.transit.transit_elements.FareRule method*), 434
save_to_database() (*aequilibrae.transit.transit_elements.Link method*), 434
save_to_database() (*aequilibrae.transit.transit_elements.link.Link method*), 439
save_to_database() (*aequilibrae.transit.transit_elements.Pattern method*), 434
save_to_database() (*aequilibrae.transit.transit_elements.pattern.Pattern method*), 439
save_to_database() (*aequilibrae.transit.transit_elements.Route method*), 435
save_to_database() (*aequilibrae.transit.transit_elements.route.Route method*), 440
save_to_database() (*aequilibrae.transit.transit_elements.Stop method*), 436
save_to_database() (*aequilibrae.transit.transit_elements.stop.Stop method*), 441

save_to_database() (*aequilibrae.transit.transit_elements.Trip method*), 436
save_to_database() (*aequilibrae.transit.transit_elements.trip.Trip method*), 442
save_to_disk() (*aequilibrae.Graph method*), 248
save_to_disk() (*aequilibrae.paths.Graph method*), 292
save_to_disk() (*aequilibrae.paths.graph.Graph method*), 317
save_to_disk() (*aequilibrae.paths.graph.GraphBase method*), 320
save_to_disk() (*aequilibrae.paths.graph.TransitGraph method*), 322
save_to_disk() (*aequilibrae.paths.TransitGraph method*), 312
save_to_disk() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
save_to_project() (*aequilibrae.distribution.gravity_application.GravityApplication method*), 270
save_to_project() (*aequilibrae.distribution.GravityApplication method*), 266
save_to_project() (*aequilibrae.distribution.Ipf method*), 268
save_to_project() (*aequilibrae.distribution.ipf.Ipf method*), 272
save_to_project() (*aequilibrae.GravityApplication method*), 249
save_to_project() (*aequilibrae.Ipf method*), 251
save_to_project() (*aequilibrae.NetworkSkimming method*), 253
save_to_project() (*aequilibrae.paths.network_skimming.NetworkSkimming method*), 325
save_to_project() (*aequilibrae.paths.NetworkSkimming method*), 295
save_types_to_aeq() (*aequilibrae.project.network.gmns_builder.GMNSBuilder method*), 385
save_vertices() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
save_vertices() (*aequilibrae.transit.TransitGraphBuilder method*), 421
Scenario (*class in aequilibrae.project*), 357
Scenario (*class in aequilibrae.project.scenario*), 404
select_link_flows() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
select_link_flows() (*aequilibrae.paths.TrafficAssignment method*), 304
select_link_flows() (*aequilibrae.TrafficAssignment method*), 262
serialize() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 450
Service (*class in aequilibrae.transit.transit_elements*), 435
Service (*class in aequilibrae.transit.transit_elements.service*), 440
set_agency_identifier() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
set_algorithm() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
set_algorithm() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_algorithm() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 344
set_algorithm() (*aequilibrae.paths.TrafficAssignment method*), 304
set_algorithm() (*aequilibrae.paths.TransitAssignment method*), 308
set_algorithm() (*aequilibrae.TrafficAssignment method*), 262
set_allow_map_match() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
set_authorizer() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 450
set_blocked_centroid_flows() (*aequilibrae.Graph method*), 248
set_blocked_centroid_flows() (*aequilibrae.paths.Graph method*), 292
set_blocked_centroid_flows() (*aequilibrae.paths.graph.Graph method*), 317
set_blocked_centroid_flows() (*aequilibrae.paths.graph.GraphBase method*), 320
set_blocked_centroid_flows() (*aequilibrae.paths.graph.TransitGraph method*), 322
set_blocked_centroid_flows() (*aequilibrae.paths.TransitGraph method*), 312
set_capacities() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
set_capacity_field() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_capacity_field() (*aequilibrae.paths.TrafficAssignment method*), 305
set_capacity_field() (*aequilibrae.TrafficAssignment*

method), 262
set_choice_set_generation() (*aequilibrae.paths.route_choice.RouteChoice method*), 335
set_choice_set_generation() (*aequilibrae.paths.RouteChoice method*), 299
set_classes() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
set_classes() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_classes() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 345
set_classes() (*aequilibrae.paths.TrafficAssignment method*), 305
set_classes() (*aequilibrae.paths.TransitAssignment method*), 308
set_classes() (*aequilibrae.TrafficAssignment method*), 262
set_colour_styling() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
set_cores() (*aequilibrae.AssignmentResults method*), 246
set_cores() (*aequilibrae.NetworkSkimming method*), 253
set_cores() (*aequilibrae.paths.AssignmentResults method*), 290
set_cores() (*aequilibrae.paths.connectivity_analysis.ConnectivityAnalysis method*), 315
set_cores() (*aequilibrae.paths.ConnectivityAnalysis method*), 290
set_cores() (*aequilibrae.paths.network_skimming.NetworkSkimming method*), 325
set_cores() (*aequilibrae.paths.NetworkSkimming method*), 295
set_cores() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
set_cores() (*aequilibrae.paths.results.assignment_results.AssignmentResultsBase method*), 330
set_cores() (*aequilibrae.paths.results.assignment_results.TransitAssignmentResults method*), 330
set_cores() (*aequilibrae.paths.results.AssignmentResults method*), 326
set_cores() (*aequilibrae.paths.results.TransitAssignmentResults method*), 328
set_cores() (*aequilibrae.paths.route_choice.RouteChoice method*), 336
set_cores() (*aequilibrae.paths.RouteChoice method*), 300
set_cores() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
set_cores() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_cores() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 345
set_cores() (*aequilibrae.paths.TrafficAssignment method*), 305
set_cores() (*aequilibrae.paths.TransitAssignment method*), 309
set_cores() (*aequilibrae.paths.TransitAssignmentResults method*), 310
set_cores() (*aequilibrae.TrafficAssignment method*), 262
set_cores() (*in module aequilibrae.utils.core_setter*), 446
set_data() (*aequilibrae.project.data.result_record.ResultRecord method*), 367
set_date() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
set_defaults() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEMapPanel method*), 456
set_defaults() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
set_demand_matrix_core() (*aequilibrae.paths.traffic_class.TransitClass method*), 347
set_demand_matrix_core() (*aequilibrae.paths.TransitClass method*), 310
set_description() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 427
set_extra_databases() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEMapPanel method*), 456
set_extra_databases() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
set_feed() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 428
set_feed_path() (*aequilibrae.transit.gtfs_loader.GTFSReader method*),

424
set_fixed_cost() (*aequilibrae.paths.traffic_class.TrafficClass method*), 346
set_fixed_cost() (*aequilibrae.paths.TrafficClass method*), 307
set_fixed_cost() (*aequilibrae.TrafficClass method*), 263
set_frequency_field() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 345
set_frequency_field() (*aequilibrae.paths.TransitAssignment method*), 309
set_graph() (*aequilibrae.Graph method*), 248
set_graph() (*aequilibrae.paths.Graph method*), 293
set_graph() (*aequilibrae.paths.graph.Graph method*), 318
set_graph() (*aequilibrae.paths.graph.GraphBase method*), 320
set_graph() (*aequilibrae.paths.graph.TransitGraph method*), 322
set_graph() (*aequilibrae.paths.TransitGraph method*), 312
set_heuristic() (*aequilibrae.PathResults method*), 255
set_heuristic() (*aequilibrae.paths.PathResults method*), 296
set_heuristic() (*aequilibrae.paths.results.path_results.PathResults method*), 331
set_heuristic() (*aequilibrae.paths.results.PathResults method*), 327
set_index() (*aequilibrae.AequilibraeMatrix method*), 245
set_index() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 286
set_index() (*aequilibrae.matrix.AequilibraeMatrix method*), 279
set_known_spatialite_folder() (*in module aequilibrae.utils.spatialite_utils*), 460
set_legend() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEMapPanel method*), 456
set_legend() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
set_maximum_speeds() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 428
set_modes() (*aequilibrae.project.network.Link method*), 371
set_modes() (*aequilibrae.project.network.link.Link method*), 387
set_path_file_format() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_path_file_format() (*aequilibrae.paths.TrafficAssignment method*), 305
set_path_file_format() (*aequilibrae.TrafficAssignment method*), 262
set_pce() (*aequilibrae.paths.traffic_class.TrafficClass method*), 346
set_pce() (*aequilibrae.paths.TrafficClass method*), 307
set_pce() (*aequilibrae.TrafficClass method*), 264
set_pces() (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder method*), 428
set_progress_handler() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 450
set_save_path_files() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_save_path_files() (*aequilibrae.paths.TrafficAssignment method*), 305
set_save_path_files() (*aequilibrae.TrafficAssignment method*), 262
set_save_routes() (*aequilibrae.paths.route_choice.RouteChoice method*), 336
set_save_routes() (*aequilibrae.paths.RouteChoice method*), 300
set_select_links() (*aequilibrae.paths.route_choice.RouteChoice method*), 336
set_select_links() (*aequilibrae.paths.RouteChoice method*), 300
set_select_links() (*aequilibrae.paths.traffic_class.TrafficClass method*), 346
set_select_links() (*aequilibrae.paths.TrafficClass method*), 307
set_select_links() (*aequilibrae.TrafficClass method*), 264
set_skimming() (*aequilibrae.Graph method*), 248
set_skimming() (*aequilibrae.paths.Graph method*), 293
set_skimming() (*aequilibrae.paths.graph.Graph method*), 318
set_skimming() (*aequilibrae.paths.graph.GraphBase method*), 320
set_skimming() (*aequilibrae.paths.graph.TransitGraph method*), 322
set_skimming() (*aequilibrae.paths.TransitGraph method*), 312
set_skimming_fields() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 345
set_skimming_fields() (*aequilibrae.paths.TransitAssignment method*), 309

set_time_field() (*aequilibrae.paths.traffic_assignment.AssignmentBase method*), 338
set_time_field() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_time_field() (*aequilibrae.paths.traffic_assignment.TransitAssignment method*), 345
set_time_field() (*aequilibrae.paths.TrafficAssignment method*), 305
set_time_field() (*aequilibrae.paths.TransitAssignment method*), 309
set_time_field() (*aequilibrae.project.Network method*), 352
set_time_field() (*aequilibrae.project.network.Network method*), 377
set_time_field() (*aequilibrae.project.network.network.Network method*), 394
set_time_field() (*aequilibrae.TrafficAssignment method*), 262
set_trace_callback() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 450
set_vdf() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 342
set_vdf() (*aequilibrae.paths.TrafficAssignment method*), 305
set_vdf() (*aequilibrae.TrafficAssignment method*), 262
set_vdf_parameters() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 343
set_vdf_parameters() (*aequilibrae.paths.TrafficAssignment method*), 305
set_vdf_parameters() (*aequilibrae.TrafficAssignment method*), 262
set_vot() (*aequilibrae.paths.traffic_class.TrafficClass method*), 346
set_vot() (*aequilibrae.paths.TrafficClass method*), 307
set_vot() (*aequilibrae.TrafficClass method*), 264
set_width_styling() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
setDescription() (*aequilibrae.AequilibraeMatrix method*), 244
setDescription() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 286
setDescription() (*aequilibrae.matrix.AequilibraeMatrix method*), 278
setlimit() (*aequilibrae.utils.db_utils.AequilibraEConnection method*), 450
setName() (*aequilibrae.AequilibraeMatrix method*), 244
setName() (*aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix method*), 286
setName() (*aequilibrae.matrix.AequilibraeMatrix method*), 279
shape (*aequilibrae.matrix.COO attribute*), 280
shape (*aequilibrae.matrix.coo_demand.GeneralisedCOODemand attribute*), 287
shape (*aequilibrae.matrix.GeneralisedCOODemand attribute*), 280
shape (*aequilibrae.matrix.sparse_matrix.COO attribute*), 288
shape (*aequilibrae.transit.parse_csv.empty attribute*), 428
signal (*aequilibrae.allOrNothing attribute*), 264
signal (*aequilibrae.NetworkSkimming attribute*), 253
signal (*aequilibrae.paths.all_or_nothing.allOrNothing attribute*), 314
signal (*aequilibrae.paths.allOrNothing attribute*), 313
signal (*aequilibrae.paths.linear_approximation.LinearApproximation attribute*), 323
signal (*aequilibrae.paths.network_skimming.NetworkSkimming attribute*), 325
signal (*aequilibrae.paths.NetworkSkimming attribute*), 295
signal (*aequilibrae.project.Network attribute*), 353
signal (*aequilibrae.project.network.Network attribute*), 378
signal (*aequilibrae.project.network.network.Network attribute*), 394
signal (*aequilibrae.project.network.osm.osm_builder.OSMBuilder attribute*), 397
signal (*aequilibrae.project.network.osm.osm_downloader.OSMDownloader attribute*), 398
signal (*aequilibrae.project.NetworkSimplifier attribute*), 353
signal (*aequilibrae.project.tools.network_simplifier.NetworkSimplifier attribute*), 414
signal (*aequilibrae.project.tools.NetworkSimplifier attribute*), 409
signal (*aequilibrae.transit.gtfs_loader.GTFSReader attribute*), 424
signal (*aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder attribute*), 428
SignalObject (*class in aequilibrae.utils.aeq_signal*), 446
simplify() (*aequilibrae.project.NetworkSimplifier method*), 353
simplify() (*aequilibrae.project.tools.network_simplifier.NetworkSimplifier method*), 414
simplify() (*aequilibrae.project.tools.NetworkSimplifier method*), 409
SimwrapperConfigGenerator (*class in aequilibrae.utils.simwrapper.generate_simwrapper_config*), 455
SimwrapperPanel (*class in aequili-*

brae.utils.simwrapper.simwrapper_panel), 458
skim_congested() (*aequilibrae.paths.traffic_assignment.TrafficAssignment method*), 343
skim_congested() (*aequilibrae.paths.traffic_class.TrafficClass method*), 347
skim_congested() (*aequilibrae.paths.TrafficAssignment method*), 306
skim_congested() (*aequilibrae.paths.TrafficClass method*), 307
skim_congested() (*aequilibrae.TrafficAssignment method*), 263
skim_congested() (*aequilibrae.TrafficClass method*), 264
skimmable_fields() (*aequilibrae.project.Network method*), 352
skimmable_fields() (*aequilibrae.project.network.Network method*), 377
skimmable_fields() (*aequilibrae.project.network.network.Network method*), 394
skimming_single_origin() (*in module aequilibrae.paths*), 313
SkimResults (*class in aequilibrae*), 258
SkimResults (*class in aequilibrae.paths*), 301
SkimResults (*class in aequilibrae.paths.results*), 327
SkimResults (*class in aequilibrae.paths.results.skim_results*), 332
SKIPPED (*aequilibrae.project.tools.migration_manager.MigrationStatus attribute*), 413
SKIPPED (*aequilibrae.project.tools.MigrationStatus attribute*), 408
sl_compact_link_loads (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
sl_link_loads (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
Sparse (*class in aequilibrae.matrix*), 280
Sparse (*class in aequilibrae.matrix.sparse_matrix*), 288
spatialize_db() (*in module aequilibrae.utils.spatialite_utils*), 460
split_links_at_stops() (*in module aequilibrae.transit.functions.breaking_links_for_stop_access*), 423
sql (*aequilibrae.project.network.Links attribute*), 374
sql (*aequilibrae.project.network.links.Links attribute*), 390
sql (*aequilibrae.project.network.Nodes attribute*), 380
sql (*aequilibrae.project.network.nodes.Nodes attribute*), 397
sql (*aequilibrae.project.network.Periods attribute*), 381
sql (*aequilibrae.project.network.periods.Periods attribute*), 401
sql (*aequilibrae.project.Periods attribute*), 355
srid (*aequilibrae.transit.constants.Constants attribute*), 422
status() (*aequilibrae.project.tools.Migration method*), 406
status() (*aequilibrae.project.tools.migration_manager.Migration method*), 410
status() (*aequilibrae.project.tools.migration_manager.MigrationManager method*), 411
status() (*aequilibrae.project.tools.MigrationManager method*), 407
Stop (*class in aequilibrae.transit.transit_elements*), 435
Stop (*class in aequilibrae.transit.transit_elements.stop*), 441
stops (*aequilibrae.transit.constants.Constants attribute*), 422
SubAreaAnalysis (*class in aequilibrae.paths*), 301
SubAreaAnalysis (*class in aequilibrae.paths.sub_area*), 337
SyntheticGravityModel (*class in aequilibrae*), 258
SyntheticGravityModel (*class in aequilibrae.distribution*), 268
SyntheticGravityModel (*class in aequilibrae.distribution.synthetic_gravity_model*), 272

T

TableLoader (*class in aequilibrae.project.table_loader*), 405
temp_factory (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
TextPanel (*class in aequilibrae.utils.simwrapper.simwrapper_panel*), 458
TilePanel (*class in aequilibrae.utils.simwrapper.simwrapper_panel*), 459
time_field (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 339
time_field (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
time_field (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
time_field (*aequilibrae.paths.TrafficAssignment attribute*), 306
time_field (*aequilibrae.paths.TransitAssignment attribute*), 309
to_bytes() (*aequilibrae.project.tools.migration_manager.MigrationStatus method*), 413
to_bytes() (*aequilibrae.project.tools.MigrationStatus method*), 408

to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEMapPanel method*), 456
to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.AequilibraEResultsMapPanel method*), 458
to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.ConvergencePanel method*), 458
to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.SimWrapperPanel method*), 458
to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.TextPanel method*), 459
to_dict() (*aequilibrae.utils.simwrapper.simwrapper_panel.TitlePanel method*), 459
to_disk() (*aequilibrae.matrix.COO method*), 280
to_disk() (*aequilibrae.matrix.Sparse method*), 280
to_disk() (*aequilibrae.matrix.sparse_matrix.COO method*), 288
to_disk() (*aequilibrae.matrix.sparse_matrix.Sparse method*), 288
to_scipy() (*aequilibrae.matrix.COO method*), 280
to_scipy() (*aequilibrae.matrix.sparse_matrix.COO method*), 288
to_seconds() (*in module aequilibrae.transit.date_tools*), 422
to_time_string() (*in module aequilibrae.transit.date_tools*), 422
to_transit_graph() (*aequilibrae.transit.transit_graph_builder.TransitGraphBuilder method*), 445
to_transit_graph() (*aequilibrae.transit.TransitGraphBuilder method*), 421
total_changes (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 451
total_flow (*aequilibrae.paths.traffic_assignment.AssignmentBase attribute*), 339
total_flow (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
total_flow (*aequilibrae.paths.traffic_assignment.TransitAssignment attribute*), 345
total_flow (*aequilibrae.paths.TrafficAssignment attribute*), 306
total_flow (*aequilibrae.paths.TransitAssignment attribute*), 309
total_flows() (*aequilibrae.AssignmentResults method*), 246
total_flows() (*aequilibrae.paths.AssignmentResults method*), 290
total_flows() (*aequilibrae.paths.results.assignment_results.AssignmentResults method*), 329
total_flows() (*aequilibrae.paths.results.AssignmentResults method*), 326
TrafficAssignment (*class in aequilibrae*), 259
TrafficAssignment (*class in aequilibrae.paths*), 301
TrafficAssignmentMapPanel (*class in aequilibrae.paths.traffic_assignment*), 339
TrafficConvergencePanel (*class in aequilibrae*), 263
TrafficClass (*class in aequilibrae.paths*), 306
TrafficSimWrapperPanel (*class in aequilibrae.paths.traffic_class*), 346
TrafficClassIdentifier (*class in aequilibrae.paths.assignment_paths*), 315
TreeFunction (*aequilibrae.Project property*), 258
transit (*aequilibrae.project.Project property*), 357
transit (*aequilibrae.project.project.Project property*), 403
transit (*aequilibrae.project.Scenario attribute*), 357
transit (*aequilibrae.project.scenario.Scenario attribute*), 405
transit (*aequilibrae.transit.Transit attribute*), 418
transit (*aequilibrae.transit.transit.Transit attribute*), 432
Transit (*class in aequilibrae.transit*), 417
Transit (*class in aequilibrae.transit.transit*), 431
transit_connection (*aequilibrae.Project property*), 258
transit_connection (*aequilibrae.project.Project property*), 357
transit_connection (*aequilibrae.project.project.Project property*), 403
transit_links (*aequilibrae.transit.constants.Constants attribute*), 422
transit_migration_file (*aequilibrae.project.tools.migration_manager.MigrationManager attribute*), 412
transit_migration_file (*aequilibrae.project.tools.MigrationManager attribute*), 407
TransitAssignment (*class in aequilibrae.paths*), 307
TransitAssignment (*class in aequilibrae.paths.traffic_assignment*), 343
TransitAssignmentResults (*class in aequilibrae.paths*), 309
TransitAssignmentResults (*class in aequilibrae.paths.results*), 328
TransitAssignmentResults (*class in aequilibrae.paths.results.assignment_results*), 330
TransitClass (*class in aequilibrae.paths*), 310
TransitClass (*class in aequilibrae.paths.traffic_class*), 347
TransitGraph (*class in aequilibrae.paths*), 310
TransitGraph (*class in aequilibrae.paths.graph*), 320
TransitGraphBuilder (*class in aequilibrae.transit*), 418
TransitGraphBuilder (*class in aequilibrae.transit.transit_graph_builder*), 442
TransportClassBase (*class in aequilibrae.paths.traffic_class*), 347
Trip (*class in aequilibrae.transit.transit_elements*), 436

Trip (*class in aequilibrae.transit.transit_elements.trip*), 441
trips (*aequilibrae.transit.constants.Constants attribute*), 422
type (*aequilibrae.project.tools.Migration attribute*), 406
type (*aequilibrae.project.tools.migration_manager.Migration attribute*), 411
type (*aequilibrae.utils.db_utils.ColumnDef attribute*), 451

U

update_cores() (*aequilibrae.project.data.matrix_record.MatrixRecord method*), 366
update_database() (*aequilibrae.Matrices method*), 252
update_database() (*aequilibrae.project.data.Matrices method*), 363
update_database() (*aequilibrae.project.data.matrices.Matrices method*), 366
update_database() (*aequilibrae.project.data.Results method*), 364
update_database() (*aequilibrae.project.data.results.Results method*), 369
update_database() (*aequilibrae.project.Matrices method*), 350
update_direction_field() (*aequilibrae.project.network.gmns_exporter.GMNSExporter method*), 385
update_field_names() (*aequilibrae.project.network.gmns_exporter.GMNSExporter method*), 385
update_modes_fields() (*aequilibrae.project.network.gmns_exporter.GMNSExporter method*), 385
update_path_trace() (*in module aequilibrae.paths*), 313
update_trace() (*aequilibrae.PathResults method*), 255
update_trace() (*aequilibrae.paths.PathResults method*), 296
update_trace() (*aequilibrae.paths.results.path_results.PathResults method*), 332
update_trace() (*aequilibrae.paths.results.PathResults method*), 327
upgrade() (*aequilibrae.Project method*), 257
upgrade() (*aequilibrae.project.Project method*), 356
upgrade() (*aequilibrae.project.project.Project method*), 402
upgrade() (*aequilibrae.project.tools.migration_manager.MigrationManager method*), 411
upgrade() (*aequilibrae.project.tools.MigrationManager method*), 407
use_scenario() (*aequilibrae.Project method*), 257
use_scenario() (*aequilibrae.project.Project method*), 356
use_scenario() (*aequilibrae.project.project.Project method*), 403

V

VDF (*class in aequilibrae*), 264
VDF (*class in aequilibrae.paths*), 312
VDF (*class in aequilibrae.paths.vdf*), 347
vdf_parameters (*aequilibrae.paths.traffic_assignment.TrafficAssignment attribute*), 343
vdf_parameters (*aequilibrae.paths.TrafficAssignment attribute*), 306

W

Warning (*aequilibrae.utils.db_utils.AequilibraEConnection attribute*), 450
where (*aequilibrae.paths.route_choice.RouteChoice attribute*), 337
WorkerThread (*class in aequilibrae.utils.interface.worker_thread*), 453
write_agencies() (*in module aequilibrae.transit.gtfs_writer*), 425
write_agencies() (*in module aequilibrae.transit.gtfs_writer.agency_writer*), 425
write_back() (*aequilibrae.Parameters method*), 254
write_back() (*aequilibrae.parameters.Parameters method*), 289
write_back() (*aequilibrae.project.About method*), 348
write_back() (*aequilibrae.project.about.About method*), 361
write_fares() (*in module aequilibrae.transit.gtfs_writer*), 425
write_fares() (*in module aequilibrae.transit.gtfs_writer.fare_writer*), 425
write_GTFS() (*aequilibrae.transit.route_system.RouteSystem method*), 429
write_routes() (*in module aequilibrae.transit.gtfs_writer*), 425
write_routes() (*in module aequilibrae.transit.gtfs_writer.routes_writer*), 425
write_shapes() (*in module aequilibrae.transit.gtfs_writer*), 425
write_shapes() (*in module aequilibrae.transit.gtfs_writer.shape_writer*), 426
write_stop_times() (*in module aequilibrae.transit.gtfs_writer*), 425
write_stop_times() (*in module aequilibrae.transit.gtfs_writer.stop_times_writer*), 426
write_stops() (*in module aequilibrae.transit.gtfs_writer*), 425
write_stops() (*in module aequilibrae.transit.gtfs_writer.stops_writer*), 426

write_trips() (*in module aequilibrae.transit.gtfs_writer*), 425
write_trips() (*in module aequilibrae.transit.gtfs_writer.trips_writer*), 426
write_yamls() (*aequilibrae.utils.simwrapper.generate_simwrapper_config.SimwrapperConfigGenerator method*), 455

Z

Zone (*class in aequilibrae.project*), 357
Zone (*class in aequilibrae.project.zone*), 414
zoning (*aequilibrae.Project property*), 258
zoning (*aequilibrae.project.Project property*), 357
zoning (*aequilibrae.project.project.Project property*), 403
zoning (*aequilibrae.project.Scenario attribute*), 357
zoning (*aequilibrae.project.scenario.Scenario attribute*), 405
Zoning (*class in aequilibrae.project*), 358
Zoning (*class in aequilibrae.project.zoning*), 415