AequilibraE 1.4.3

Pedro Camargo

Jun 17, 2025

CONTENTS

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 3.9, 3.10, 3.11 or 3.12. We recommend Python 3.10 or 3.11

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.

1.2 Dependencies

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

1.2.1 SpatiaLite

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.

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:

```python
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.

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
```

MacOS

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

```
brew install libspatialite
```

1.3 Hardware requirements

AequilibraE's requirements depend heavily on the size of the model you are using for computation. 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.

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.

Impressive 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 is a little different from 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.
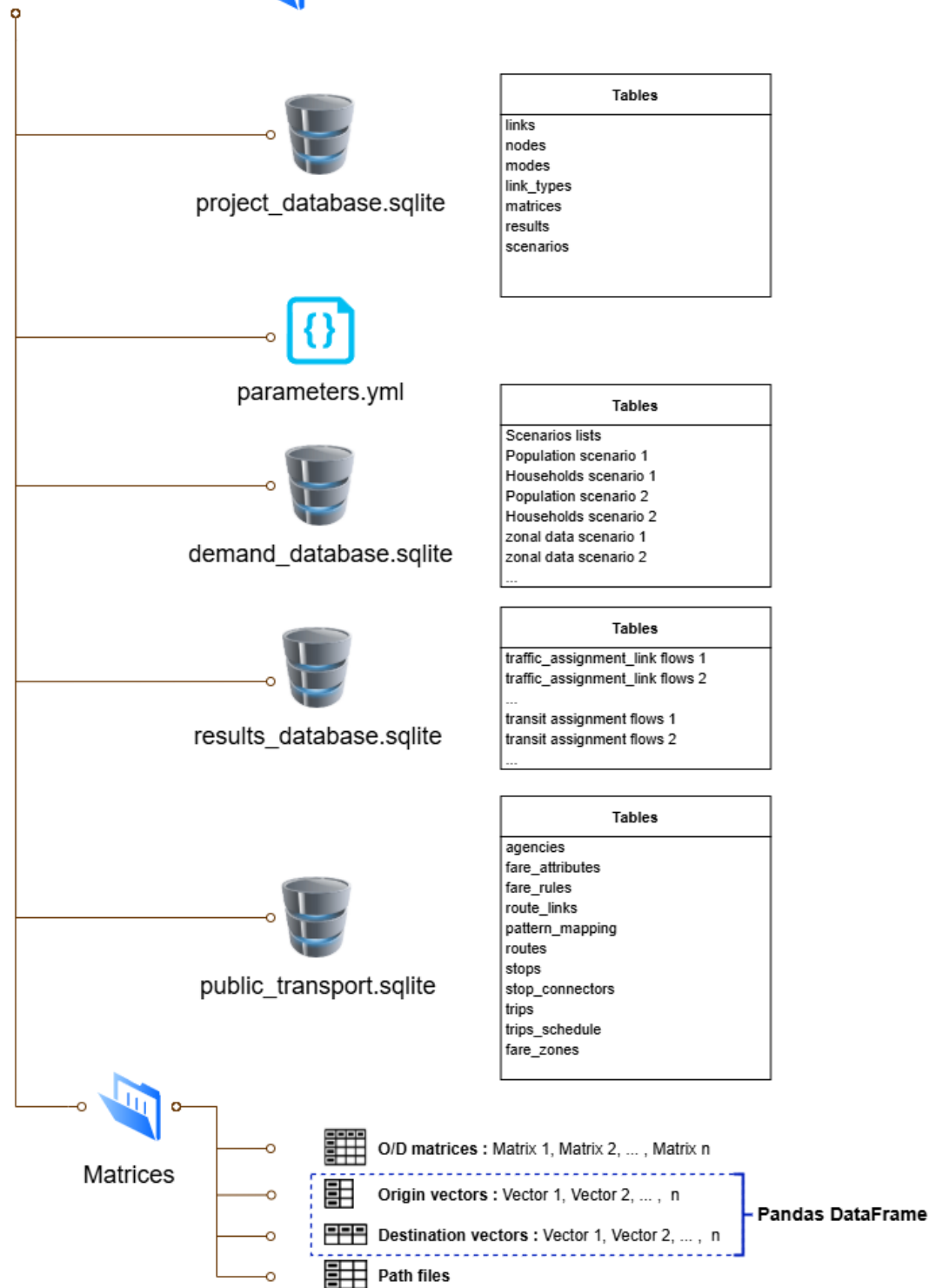
It is then important to highlight that AequilibraE, as a software, is divided in three very distinctive layers. The first, which is responsible for tables consistent with each other (including links and nodes, modes and link_types), are embedded in the data layer in the form of geo-spatial database triggers. 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 available for use with AequilibraE. This database 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.

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()*
    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.
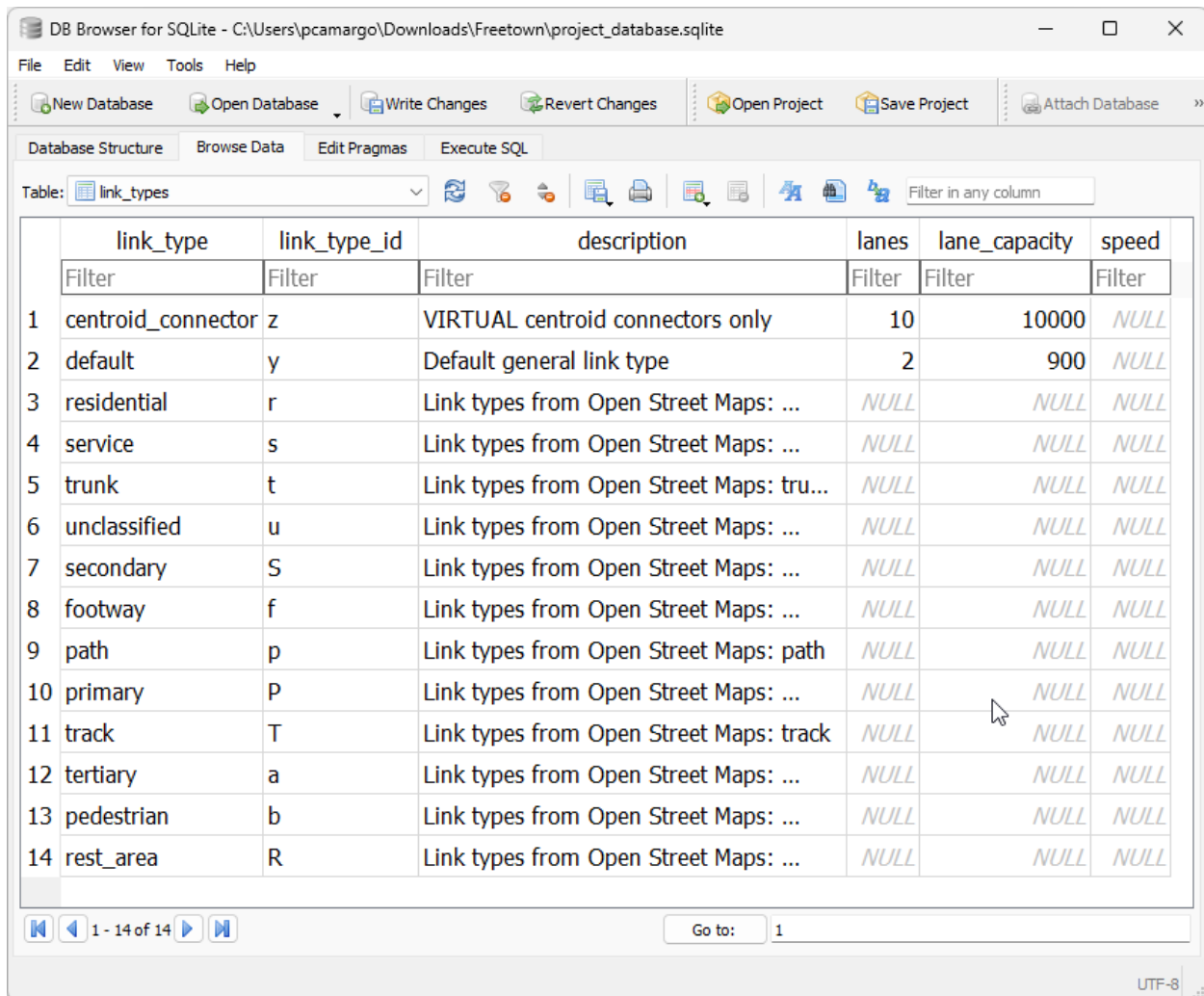
Reserved values

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
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.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.
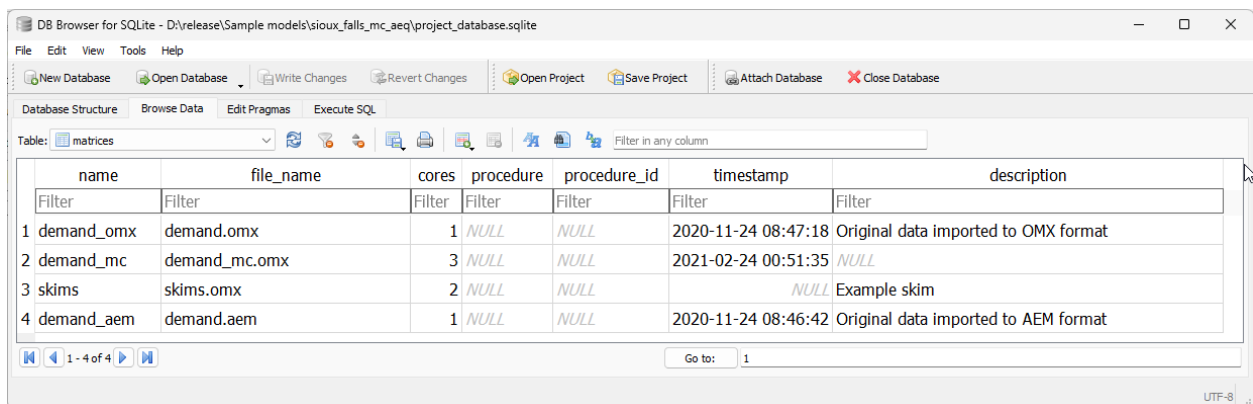
See also

- *aequilibrae.project.Zone()*
  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.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.
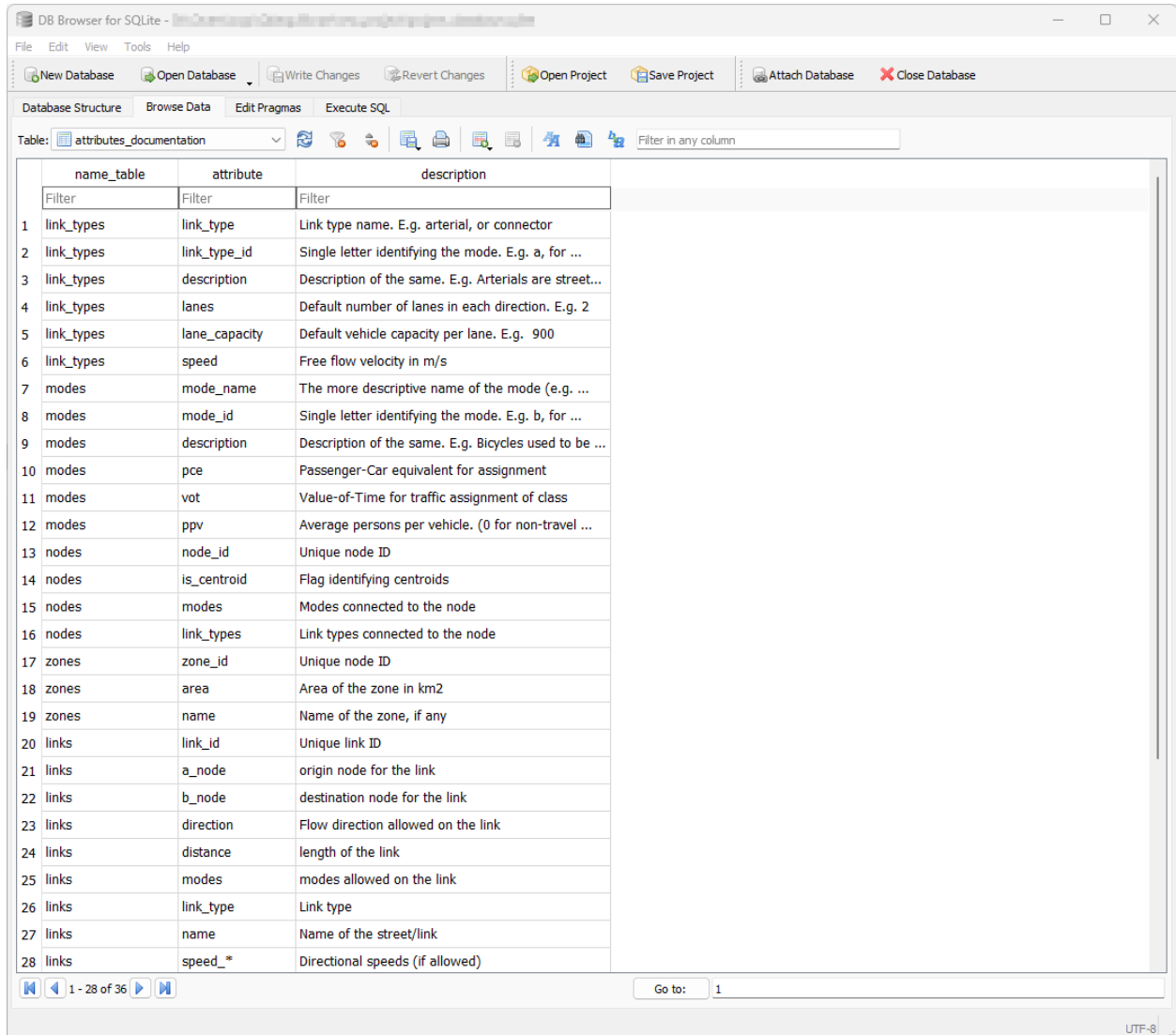
See also

- **aequilibrae.project.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:

> **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
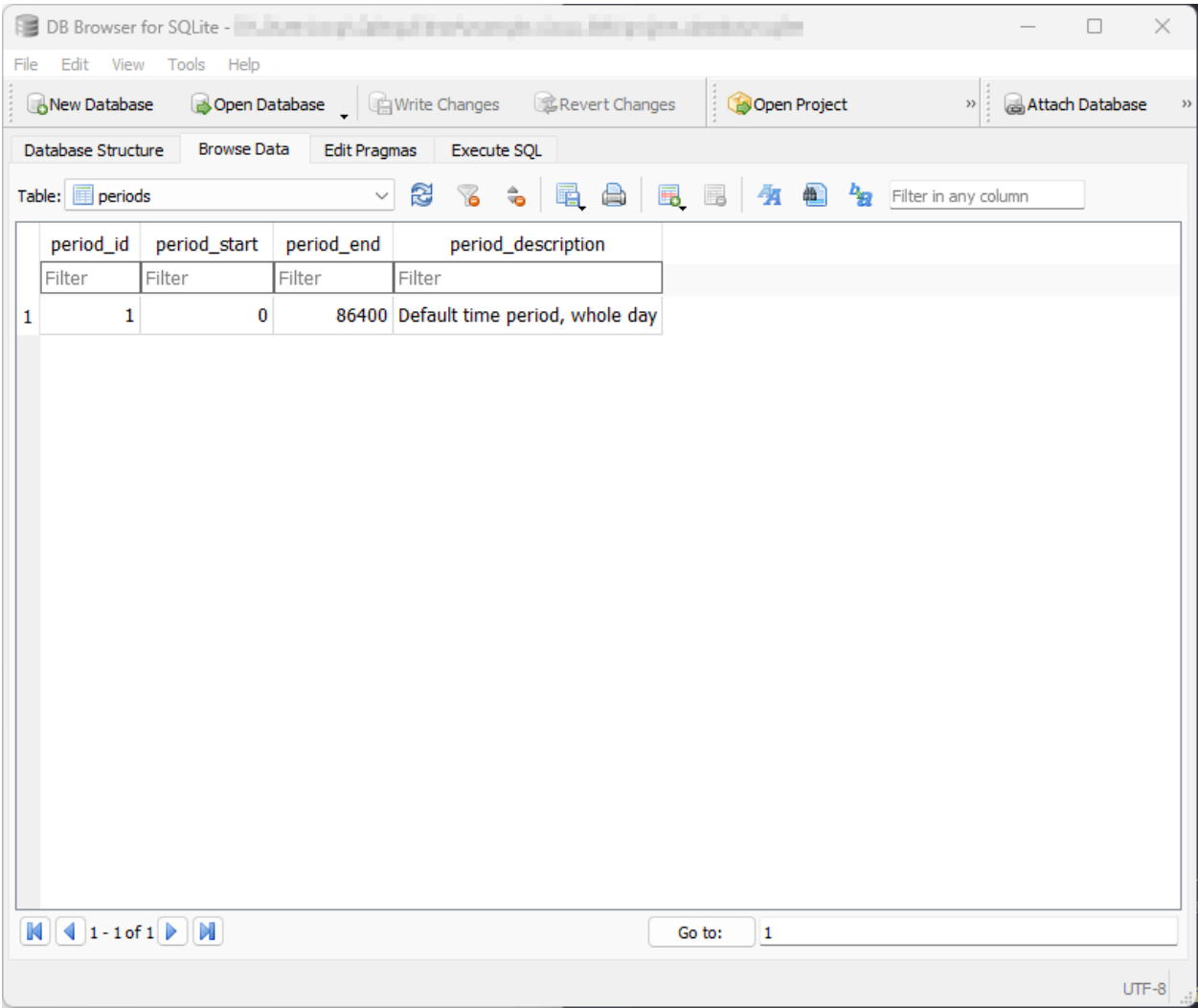
2.2.2 Project database SQL data model

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.

```sql
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');


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 attribues 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 attibutes 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 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');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES(
→'link_types','speed', 'Free flow velocity 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,
                                   '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' );
```

(continues on next page)

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

matrices table structure

The *matrices* table holds infromation 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 the the procedure that generated the result (e.g.: Traffic Assignment).

The **procedure_id** field holds an unique alpha-numeric identifier for this prodecure.

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)␣
```

(continues on next page)

```
↪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 **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
table_name*	TEXT	NO	
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 (table_name       TEXT     NOT NULL PRIMARY KEY,
                                     procedure        TEXT     NOT NULL,
                                     procedure_id     TEXT     NOT NULL,
                                     procedure_report TEXT     NOT NULL,
                                     timestamp        DATETIME DEFAULT current_
↪timestamp,
                                     description      TEXT);
```

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(
↪'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 `Run-timeError` will be raised when `project.run` is accessed. The arguments are applied via `functools.partial` and replace the objects within the module.

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

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

Assignment

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

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

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

Distribution

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

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

Network

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

Link Fields

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

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

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

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

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**.

```
gmns:
  critical_dist: 2
  node:
    equivalency: ···
    fields: ···
  link:
    equivalency: ···
    fields: ···
  use_definition:
    fields: ···
    equivalency: ···
```

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()*
>   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()*
>   Class documentation
>
> - *aequilibrae.transit.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 fuill name of the transit agency

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

service_date indicates the date for the indicate 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 attribues 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 attibutes 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 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 especifies the fare price

currency_code especifies 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 lenght 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,
```

<div align="right">(continues on next page)</div>

```
                                 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', 'Free flow velocity 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␣
→update RESTRICT ON delete RESTRICT,
                                  line_seg_idx  INTEGER,
                                  trav_time     NUMERIC  NOT NULL,
                                  freq          NUMERIC  NOT NULL,
                                  o_line_id     TEXT,
                                  d_line_id     TEXT,
                                  transfer_id   TEXT,
                                  CHECK(TYPEOF(link_id) == 'integer')
                                  UNIQUE(link_id, period_id) ON CONFLICT ABORT
                                  CHECK(link_id > 0)
                                  CHECK(TYPEOF(a_node) == 'integer')
                                  CHECK(TYPEOF(b_node) == 'integer')
                                  CHECK(TYPEOF(direction) == 'integer')
                                  CHECK(LENGTH(modes)>0)
                                  CHECK(LENGTH(direction)==1));
```

(continues on next page)

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

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

CREATE INDEX idx_period_links ON links (period_id);

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

CREATE INDEX idx_link_anode ON links (a_node);

CREATE INDEX idx_link_bnode ON links (b_node);

CREATE INDEX idx_link_modes ON links (modes);

CREATE INDEX idx_link_link_type ON links (link_type);

CREATE INDEX idx_links_a_node_b_node ON links (a_node, b_node);

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

modes table structure

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

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

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

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

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

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

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

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

(* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists modes (mode_name   VARCHAR UNIQUE NOT NULL,
                                  mode_id     VARCHAR UNIQUE NOT NULL       PRIMARY␣
→KEY,

                                  description VARCHAR,
                                  pce         NUMERIC       NOT NULL DEFAULT 1.0,
                                  vot         NUMERIC       NOT NULL DEFAULT 0,
                                  ppv         NUMERIC       NOT NULL DEFAULT 1.0
                                  CHECK(LENGTH(mode_id)==1));

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

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

(continues on next page)

```
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/Desination 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', 'Desination node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('stop', 's',
→'Stop node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('alighting', 'a
→', 'Alighting node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('boarding', 'b
→', 'Boarding node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('walking', 'w',
→ 'Walking node type');

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

```
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 segement 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' );
```

results table structure

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

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
table_name*	TEXT	NO	
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 (table_name      TEXT    NOT NULL PRIMARY KEY,
                                     procedure       TEXT    NOT NULL,
                                     procedure_id    TEXT    NOT NULL,
                                     procedure_report TEXT   NOT NULL,
                                     timestamp       DATETIME DEFAULT current_
→timestamp,
                                     description     TEXT);
```

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 idenfifies a stop, statio, or station entrance

agency_id identifies the agency fot 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 transporation 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,
```

(continues on next page)

```
    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
);
```

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.

Project Data Component

Allows the manipulation of all elements within the homonym table in the project

```
project.network.Links
project.network.Nodes
project.network.LinkTypes
project.network.Modes
project.network.Periods
project.Zoning
```

Project Data Items

Correspond to ONE object in the homonym data component class

```
project.network.Link
project.network.Node
project.network.LinkType
project.network.Mode
project.network.Period
project.Zone
```

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()*
  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

>>> # 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()`**

> 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 modifiy 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
>>> 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()_**
>   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 everytime 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()***
>   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")

# 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
↪funcionalities", "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.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 entires 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 wiesly once the deletion of data in the log file can't be undone.
>>> project_log.clear()

>>> project.close()
```

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

project.matrices

This method ia 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_omx")
>>> 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()

# Similar to the `get_record` function, we have the `get_matrix`, which allows you to
# get an AequilibraE matrix.
>>> matrices.get_matrix("demand_aem")

>>> project.close()
```

See also

- *aequilibrae.project.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
↪funcionalities", "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.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()*
    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()`**
>   Class documentation
>
> - *Periods table*
>   Table documentation

AequilibraE Matrix

AequilibraE matrices are very useful objects that allow you to make the most with AequilibraE. 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

>>> file = os.path.join(my_folder_path, "path_to_my_matrix.aem")
>>> 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(file_name=file, zones=num_zones, matrix_names=names)

# `memory_only` parameter can be changed to `True` case you want to save the matrix
↪in disk.

# 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
['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 storing project data, AequilibraE matrices are objects necessary to run procedures, such as traffic assignment. To do so, one must create a computational view of the matrix, which allows AequilibraE matrices to be used in parallelized algorithms. It is possible to create a computational view for more than one matrix at a time.

Case you're using matricial data from an OMX file, this step is mandatory to load the data to memory, otherwise the matrix is useless in other procedures.

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

You can also export AequilibraE matrices to another file formats, such as CSV and OMX. When exporting to a OMX file, you can choose the cores os the matrix you want to save, although this is not the case for CSV file, in which all cores will be exported as separate columns in the output file.

```
>>> mat.export(os.path.join(my_folder_path, 'my_new_omx_file.omx'))
```

```
>>> mat.export(os.path.join(my_folder_path, 'my_new_csv_file.csv'))
```

The `export` method also allows you to change your mind and save your AequilibraE matrix into an AEM file, if it's only in memory.

```
>>> mat.export(os.path.join(my_folder_path, 'my_new_aem_file.aem'))
```

To avoid errors, 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, to remove it from memory and flush the data to disk, or to close the OMX file, if that's the case.

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

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

```
>>> mat = AequilibraeMatrix()
>>> mat.load(os.path.join(my_folder_path, 'my_new_aem_file.aem'))

>>> mat.get_matrix("only_ones")
memmap([[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.]])
```

> **Important**
>
> File extension for AequilibraE matrices is **AEM**.

> **See also**
>
> *aequilibrae.matrix.AequilibraeMatrix()*
>   Class documentatiom
>
> *Traffic Assignment without an AequilibraE Model*
>   Usage example

OpenMatrix (OMX)

AequilibraE can handle OMX files, but 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 does it work.

Creating an AequilibraE matrix from an OMX file is pretty straightforward.

```
>>> file_path = os.path.join(my_folder_path, "path_to_new_matrix.aem")
>>> omx_path = os.path.join(my_folder_path, "my_new_omx_file.omx")


>>> omx_mat = AequilibraeMatrix()
>>> omx_mat.create_from_omx(omx_path, file_path)


>>> mat.get_matrix("only_ones")
memmap([[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.]])
```

2.2.7 Examples

AequilibraE Project

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/

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

from aequilibrae.utils.create_example import create_example
```

```python
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

```python
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.

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

(continues on next page)

```python
from os.path import join
from aequilibrae.utils.create_example import create_example
import logging
import sys
```

```python
# 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

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

```python
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!

```python
# 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
```

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

We build our graphs

```python
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

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.

3.1.1 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.run*

```python
# 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
```

```python
# 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.

```python
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*.

```python
mode = "c"
```

```python
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.

```python
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*.

```python
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
```

(continues on next page)

```
\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(project)
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 sqlite3
import pandas as pd
import geopandas as gpd
```

Let's retrieve the results

```
res_path = join(project.project_base_path, "results_database.sqlite")
conn = sqlite3.connect(res_path)

results = pd.read_sql("SELECT * FROM my_run_module_example", conn).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 3. Run module

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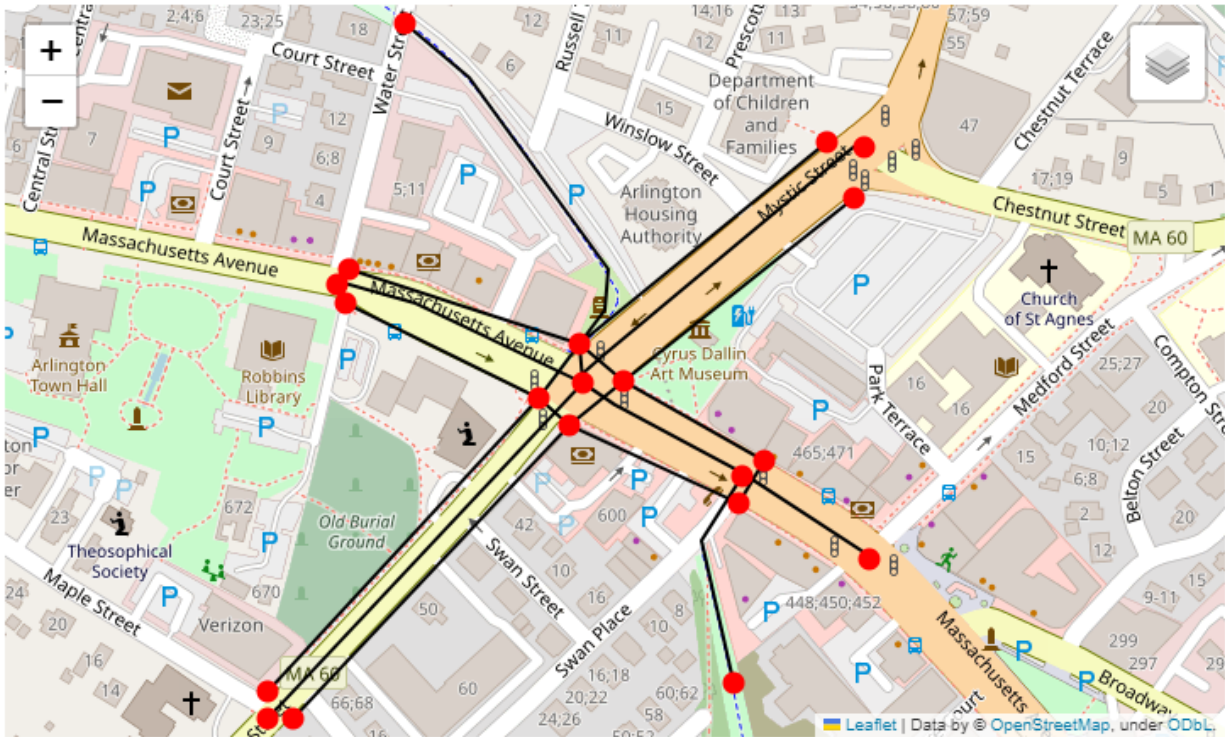
It is possible to import the following files from a GMNS source:

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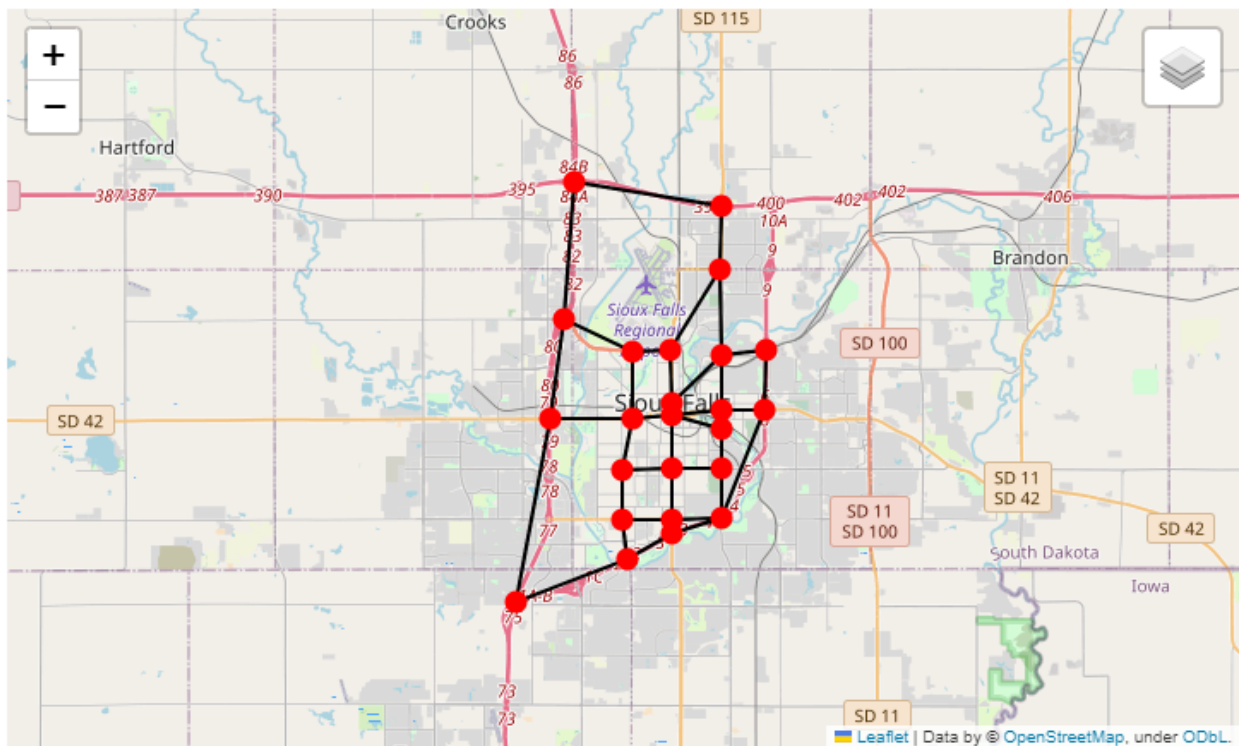
It is possible to export an AequilibraE network to the following tables in GMNS format:

- 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 optinal field listed in the GMNS node table specification.

You can find the GMNS specification here.

> **See also**
>
> - `aequilibrae.project.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 referes 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 `Geodesi-cLength`, 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.create_from_osm()*

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

from aequilibrae import Project
```

```python
# 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.

```python
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.

```python
# 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()*

```python
# 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
```

```python
# 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.

```python
# 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.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

```python
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

```python
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.

```python
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!

```python
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
```

```python
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()*

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

```python
from os.path import join
from aequilibrae.utils.create_example import create_example
from shapely.geometry import LineString, Point
import matplotlib.pyplot as plt
```

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

project = create_example(fldr)
```

```python
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

```python
# 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.

```python
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.

```python
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.

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

from aequilibrae.utils.create_example import create_example
import folium
```

```python
# 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

```python
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!

```python
# links.plot()
```

Let's create copies of our link layers for each mode

```python
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!

```python
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()*

```python
# 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
```

```python
# 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.

```python
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.

```python
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.create_from_gmns()*

```
# Imports
from uuid import uuid4
from os.path import join
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
link_file = "https://raw.githubusercontent.com/zephyr-data-specs/GMNS/main/examples/
↪Arlington_Signals/link.csv"
node_file = "https://raw.githubusercontent.com/zephyr-data-specs/GMNS/main/examples/
↪Arlington_Signals/node.csv"
use_group_file = "https://raw.githubusercontent.com/zephyr-data-specs/GMNS/main/
↪examples/Arlington_Signals/use_group.csv"
```

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

(continues on next page)

```
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.PathResults()*

```python
# 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
```

```python
# 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.

```python
# 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()

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

```python
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.LinkTypes()*

```python
# 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
```

```python
# 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

```python
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:

```python
link_types = df.link_type.unique()
```

And the existing link types are

```python
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 theses 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.NetworkSimplifier()*

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

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]
```

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

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

```python
# 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)
```

```python
# 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.

```python
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.

```python
net = NetworkSimplifier()
```

When we choose to simplify the network, we pass a graph object to the function, and the output of this operation is

```python
net.simplify(graph)
net.rebuild_network()
```

Let's plot the previous and actual networks!

```python
links_after = net.network.links.data
nodes_after = net.network.nodes.data
```

```python
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")
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()
```

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 funcions: `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 calibared a model.

Plase check other arguments and parameters that are passed to `GravityApplication` in its documentation.

> **See also**
>
> - ***aequilibrae.distribution.SyntheticGravityModel()***
>   Function documentation
>
> - ***aequilibrae.distribution.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 funcions: `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.

Plase check other arguments and parameters that are passed to `GravityCalibration` in its documentation.

> **See also**
>
> - *`aequilibrae.distribution.GravityCalibration()`*
>   Function documentation

5.1.3 `Ipf`

IPF is an acronym for Iterative Proportial 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()`*
>   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.
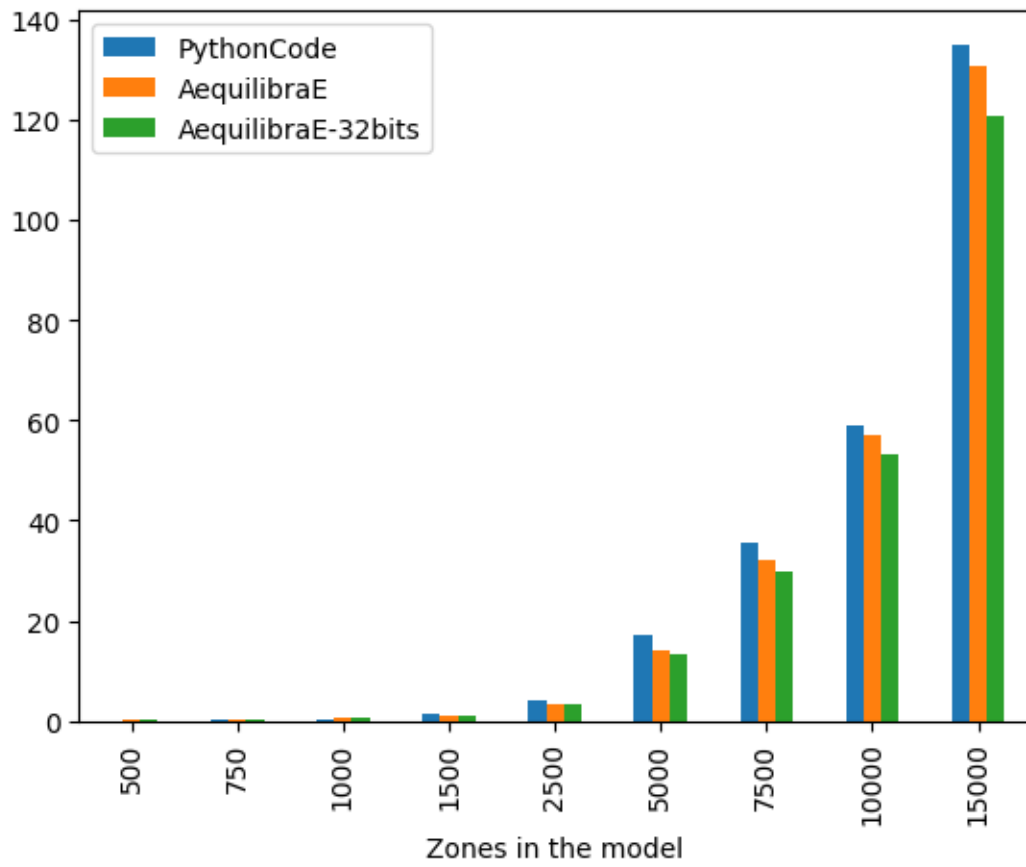
Fig. 1: AequilibraE's IPF runtime
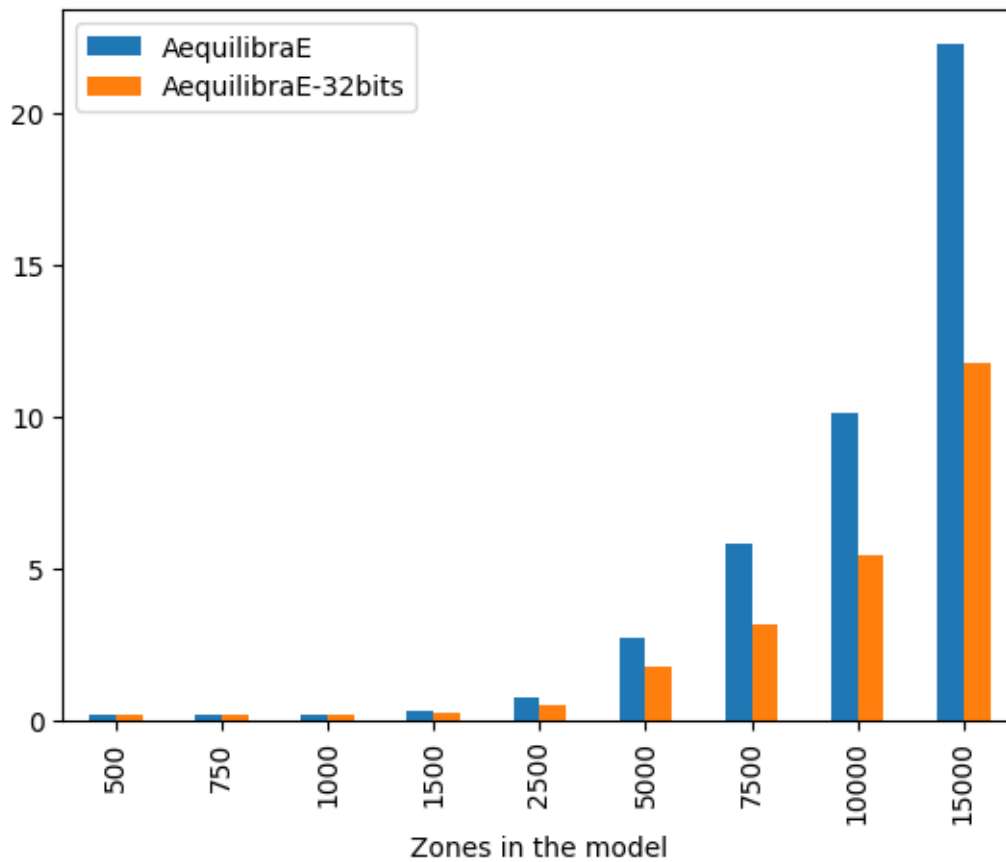
Fig. 2: AequilibraE's IPF runtime 32 vs 64 bits

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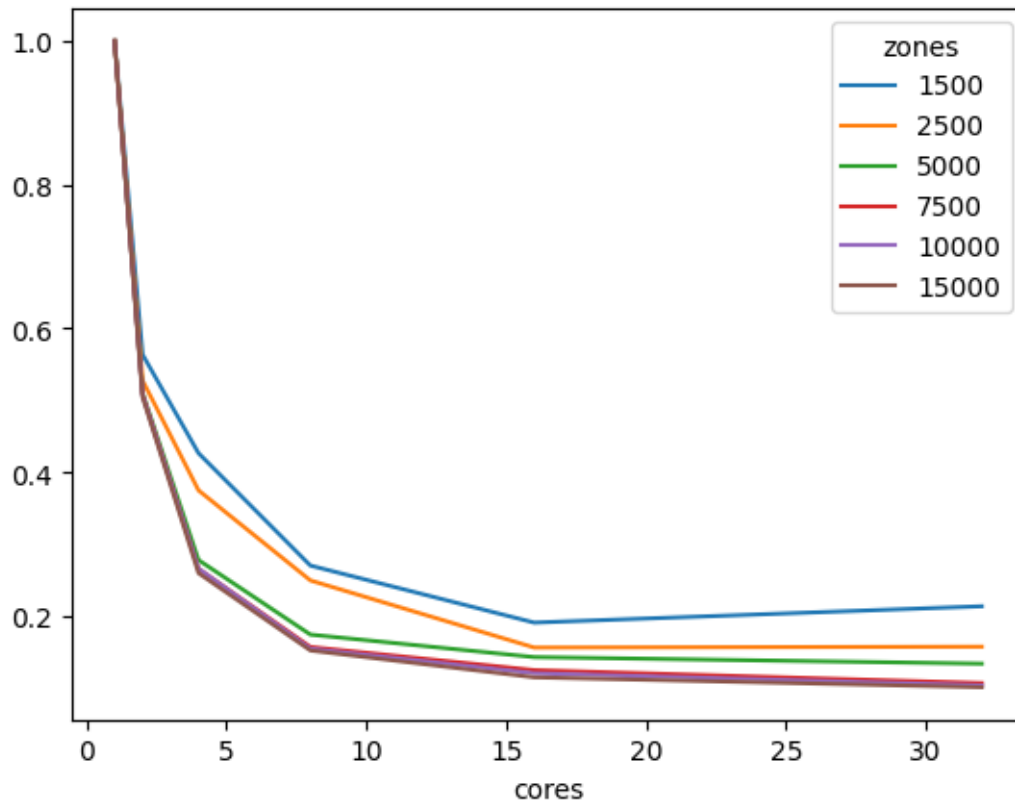
Fig. 3: number of cores used in IPF for 64 bit matrices

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.ipf_core import ipf_core
from tqdm import tqdm
```
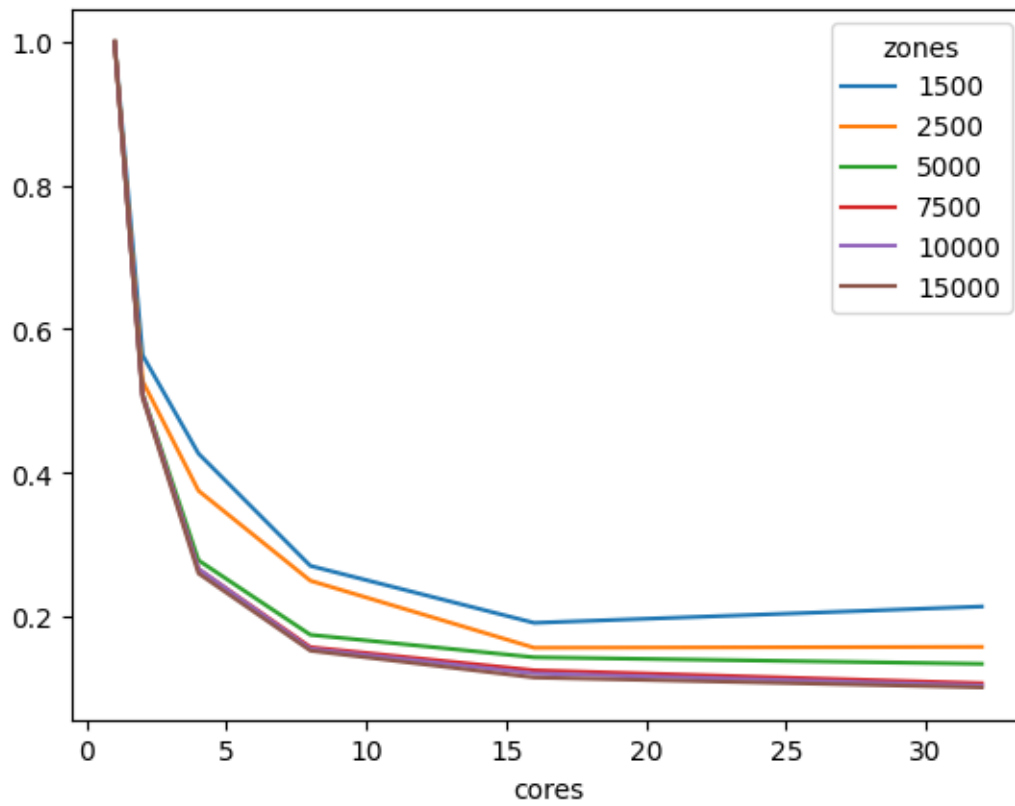
Fig. 4: number of cores used in IPF for 32 bit matrices

```
# 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)

```
        # 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'>
```
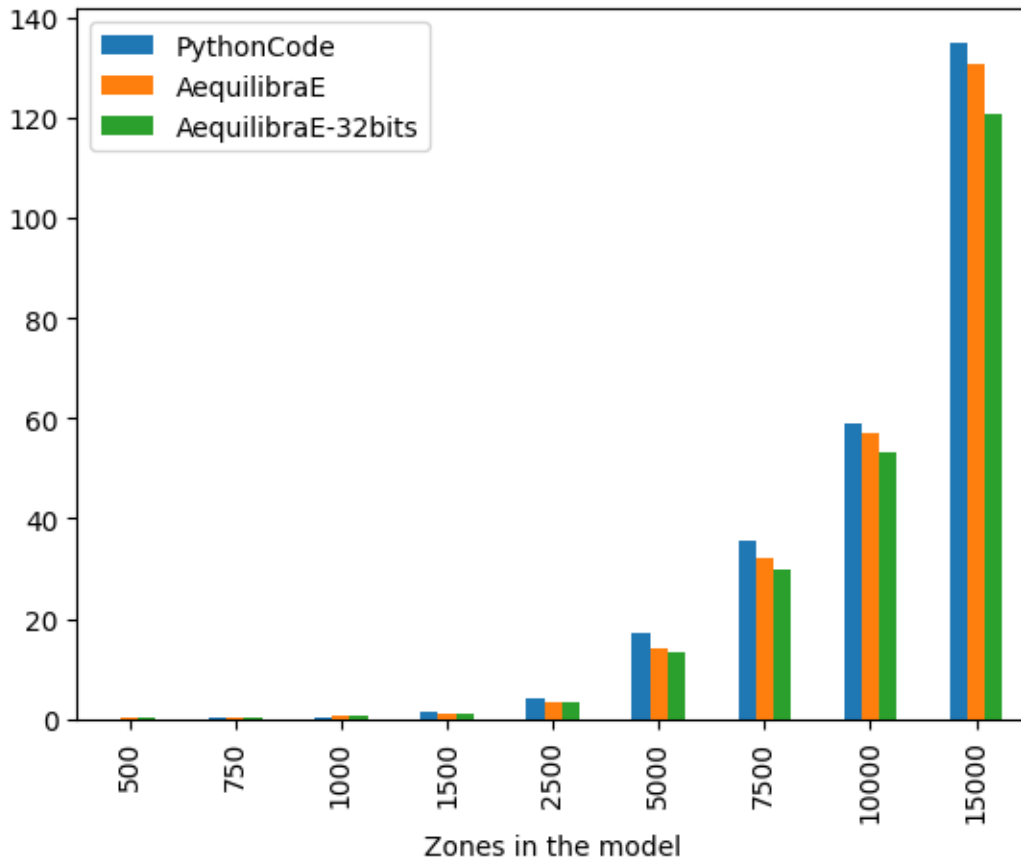
```
bench_df.groupby(["Zones in the model"]).mean()
```

```
#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)

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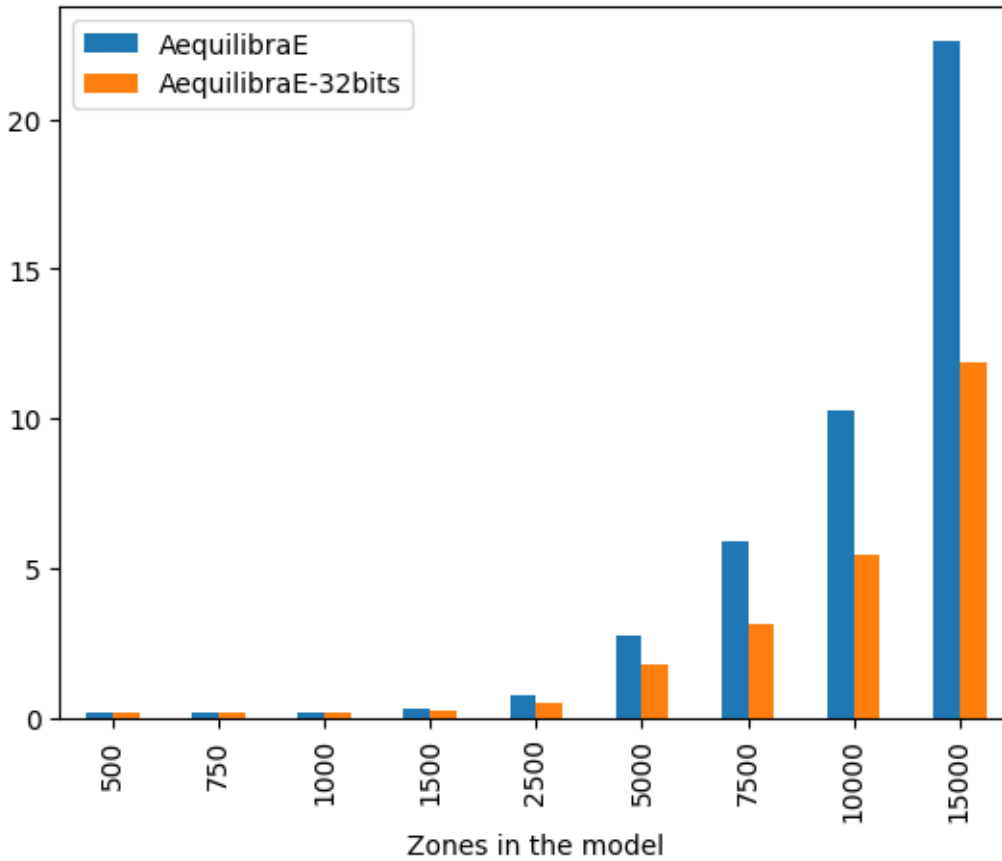
```
bench_df_parallel.groupby(["Zones in the model"]).mean()
```

```
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]
```

```python
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
```



```python
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):
```
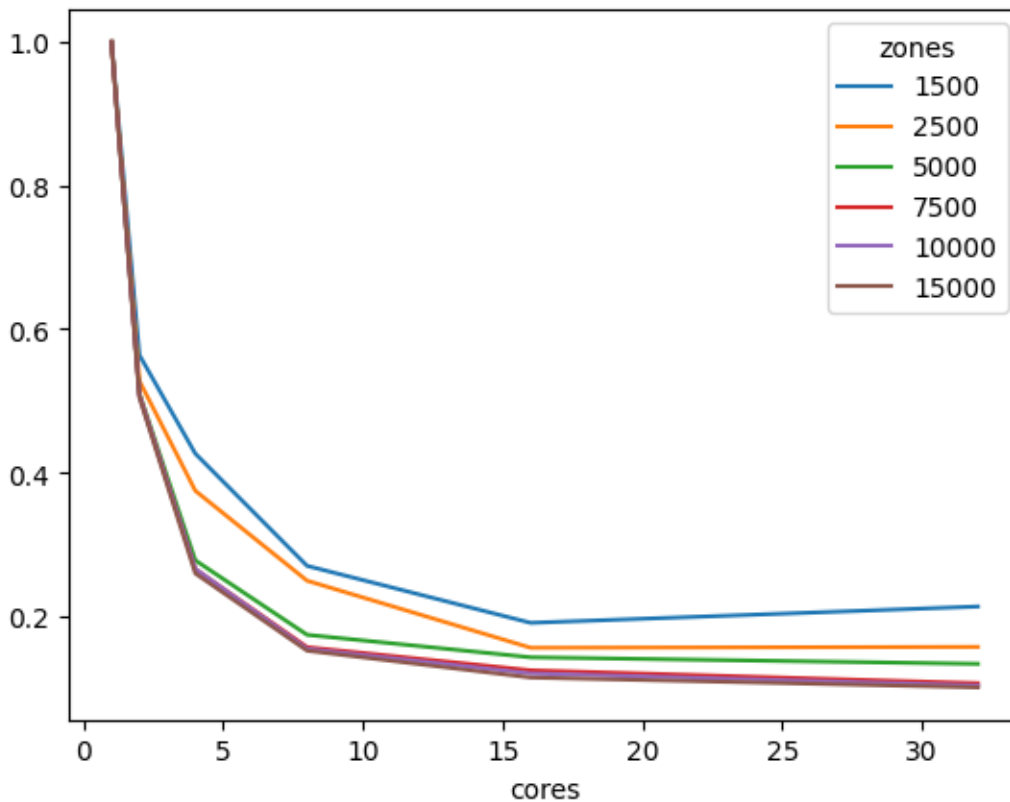
```
        mat1 = np.random.rand(zones, zones)
        target_prod = np.random.rand(zones)
        target_atra = np.random.rand(zones)
        target_atra *= target_prod.sum()/target_atra.sum()

        aeq_mat = np.array(deepcopy(mat1), np.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
```
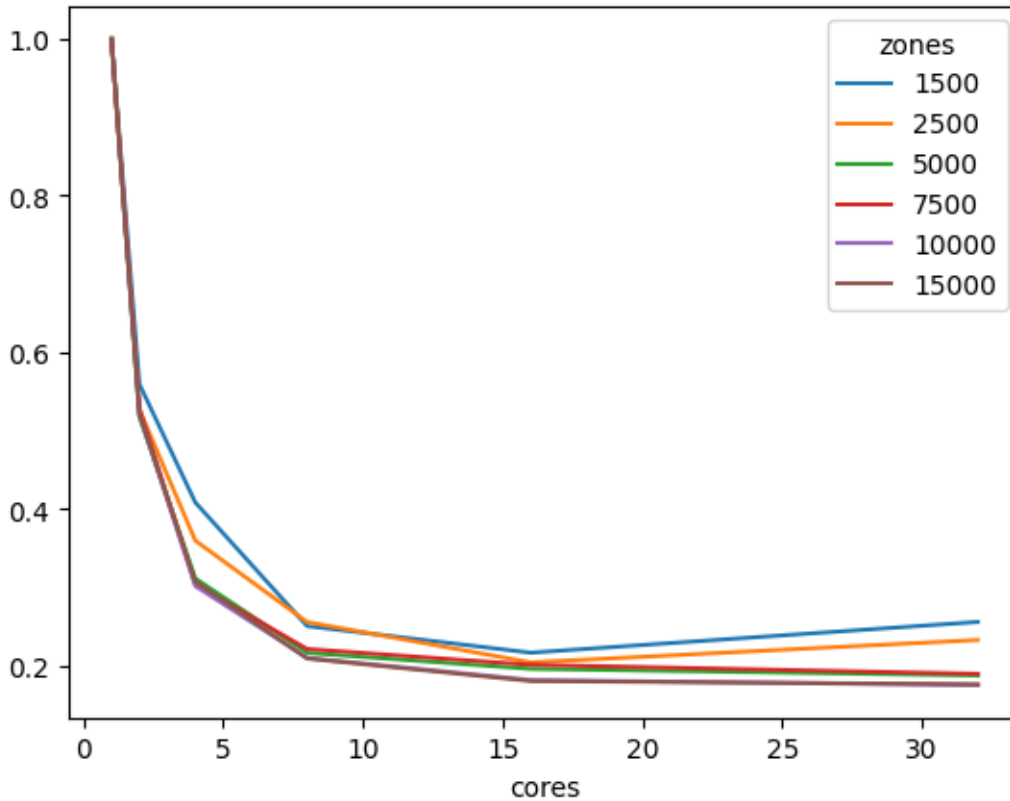
5.3 Examples

5.3.1 Distribution Procedures

Running IPF with NumPy array

In this example, we show how to use `aequilibrae.distribution.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_benchmark.

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.ipf_core()`

```
# Imports
import numpy as np

from aequilibrae.distribution.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.AequilibraeMatrix()*
- *aequilibrae.distribution.Ipf()*

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

import numpy as np
```

```python
import pandas as pd

from aequilibrae.distribution import Ipf
from aequilibrae.matrix import AequilibraeMatrix
```

```python
folder = gettempdir()
```

```python
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]
```

```python
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()
```

```python
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)
```

```python
args = {
    "matrix": mtx,
    "vectors": vectors,
    "row_field": "productions",
    "column_field": "attractions",
    "nan_as_zero": True,
}
fratar = Ipf(**args)
fratar.fit()
```

```python
fratar.output.matrix_view
```

```python
for line in fratar.report:
    print(line)
```

5.4 References

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 arbritrary 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.PathResults()***
>   Class documentation
>
> - *Path computation*
>   Usage example
>
> - *Network skimming*
>   Usage example

6.1 AequilibraE Graphs

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

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

>>> g = Graph()

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

6.1.1 Directionality

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 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_example(project_path, "coquimbo")

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

6.1.3 Manipulating graphs in memory

As mentioned before, 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 inidex 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
```

(continues on next page)

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

6.1.4 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"])
```

6.1.5 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()***
>   Class documentation
> - ***aequilibrae.paths.TransitGraph()***
>   Class documentation

6.1.6 Topological simplification

When using the AequilibraE graph to perform topological simplification of the graph, 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)
```

6.2 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()*

```python
# 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.

```python
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.

```python
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.

```python
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

```python
net.head()
```

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

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

… and then set the graph's configurations.

```python
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.PathResults()*

Imports

```python
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")
```

```python
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.

```python
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.

```python
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:

```python
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.

```python
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.

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

```python
links.explore(color="blue", style_kwds={'weight':5})
```

```python
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.NetworkSkimming()*

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

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

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

```python
import logging
import sys
```

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

```python
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

```python
import numpy as np
```

Let's build all graphs

```python
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

```python
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()
```

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 some users, so this section of the documentation will include discussions of the mechanics of some of these procedures and some light discussion on its 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)

```
>>> 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 classe 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`, or `INRETS`

```
>>> 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

For now, the VDF functions available in AequilibraE are

- BPR[1]

$$CongestedTime_i = FreeFlowTime_i * (1 + \alpha * (\frac{Volume_i}{Capacity_i})^\beta)$$

- Spiess' conical[2]

$$CongestedTime_i = FreeFlowTime_i * (2 + \sqrt[2]{\alpha^2 * (1 - \frac{Volume_i}{Capacity_i})^2 + \beta^2} - \alpha * (1 - \frac{Volume_i}{Capacity_i}) - \beta)$$

- and French INRETS (alpha < 1)

Before capacity

$$CongestedTime_i = FreeFlowTime_i * \frac{1.1 - (\alpha * \frac{Volume_i}{Capacity_i})}{1.1 - \frac{Volume_i}{Capacity_i}}$$

and after capacity

$$CongestedTime_i = FreeFlowTime_i * \frac{1.1 - \alpha}{0.1} * (\frac{Volume_i}{Capacity_i})^2$$

More functions will be added as needed/requested/possible.

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)
```

(continues on next page)

[1] Hampton Roads Transportation Planning Organization, Regional Travel Demand Model V2 (2020). Available in: https://www.hrtpo.org/uploads/docs/2020_HamptonRoads_Modelv2_MethodologyReport.pdf

[2] Spiess, H. (1990) "Technical Note—Conical Volume-Delay Functions." Transportation Science, 24(2): 153-158. Available in: https://doi.org/10.1287/trsc.24.2.153

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

7.1.4 Executing an Assignment

Finally, run traffic assignment!

```
>>> assig.execute()
```

7.1.5 References

7.2 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.

7.2.1 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.

7.2.2 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

7.2.3 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*.

7.2.4 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.

7.2.5 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.

7.2.6 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.

7.2.7 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

7.2.8 References

7.3 Traffic Assignment Validation

Similar to other complex algorithms that handle a large amount 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 1.1.0.

As shown below, the results produced by AequilibraE are within expected, although some differences have been found, particularly for Winnipeg. 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
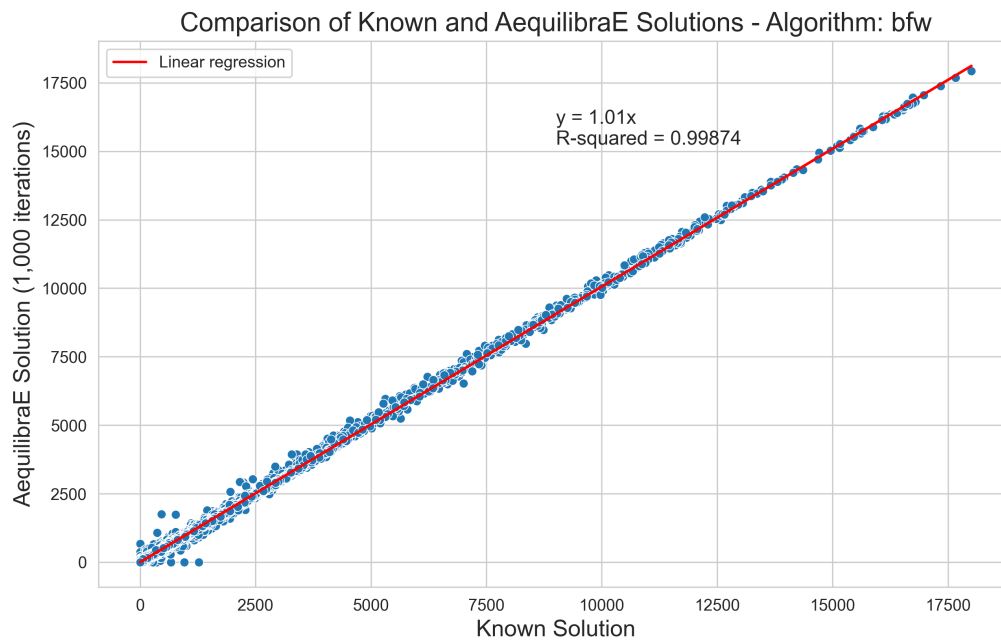
Conjugate Frank-Wolfe
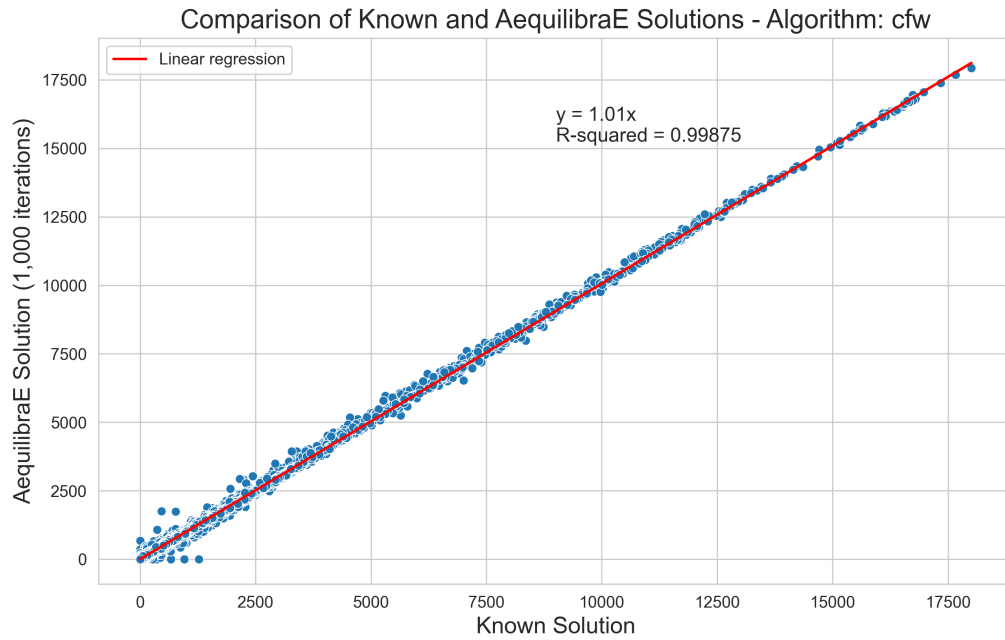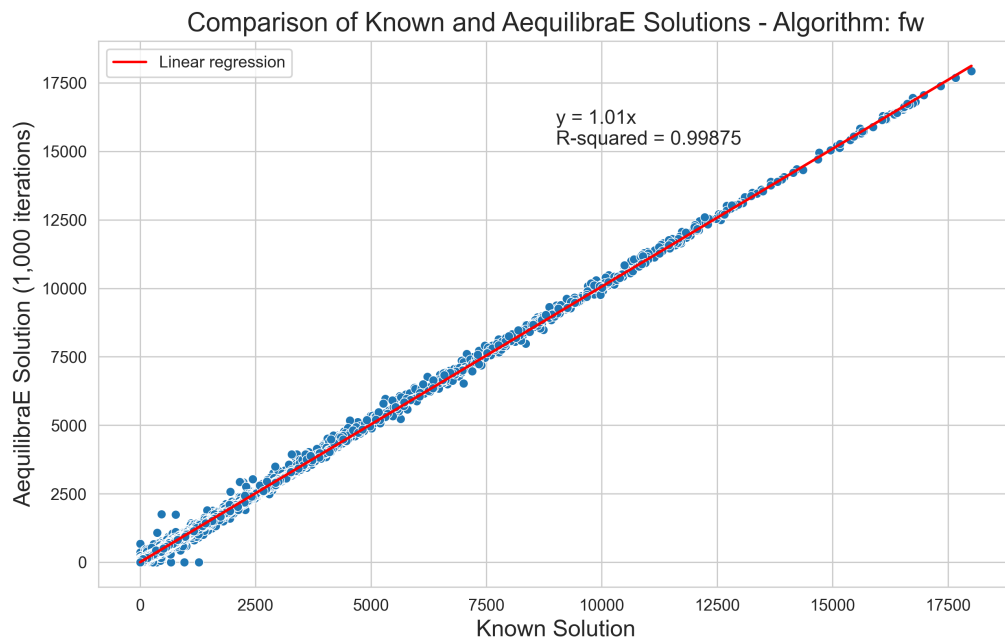
Frank-Wolfe

MSA
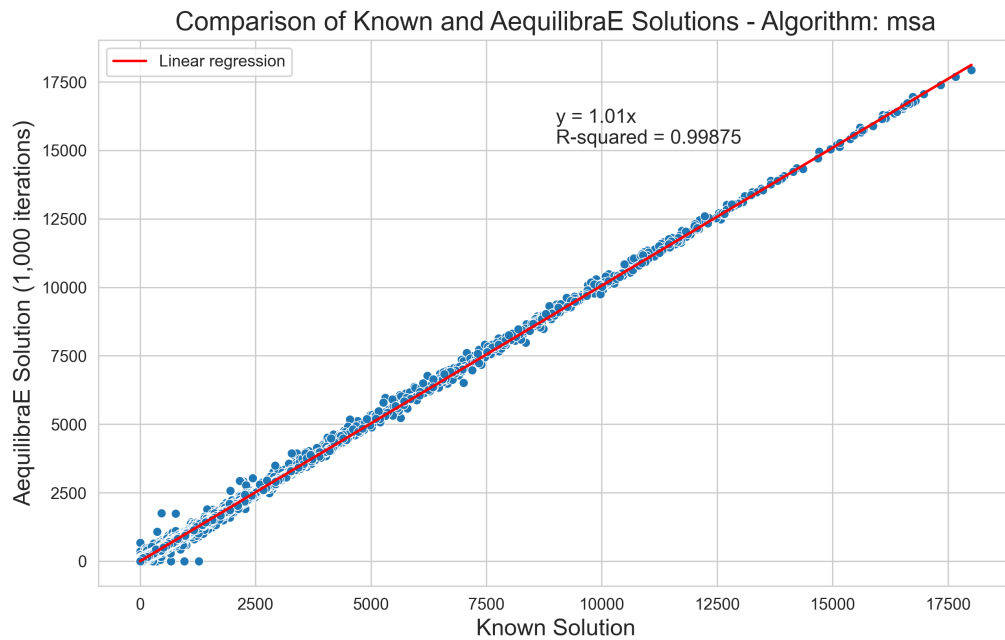
Barcelona

Network stats

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

biconjugate Frank-Wolfe

Conjugate Frank-Wolfe

Comparison of Known and AequilibraE Solutions - Algorithm: cfw)

Frank-Wolfe

Comparison of Known and AequilibraE Solutions - Algorithm: fw)

MSA
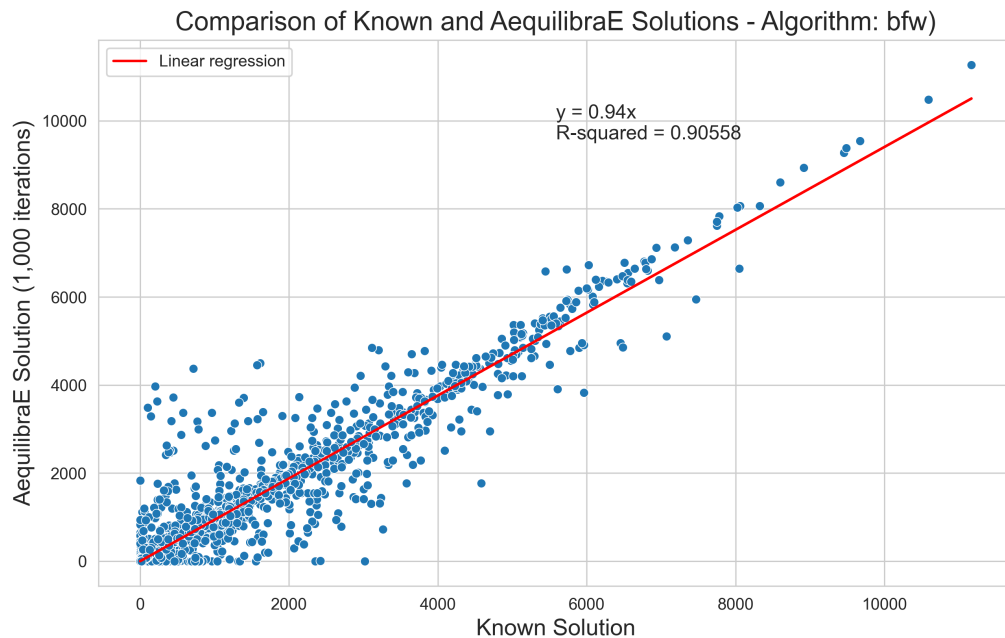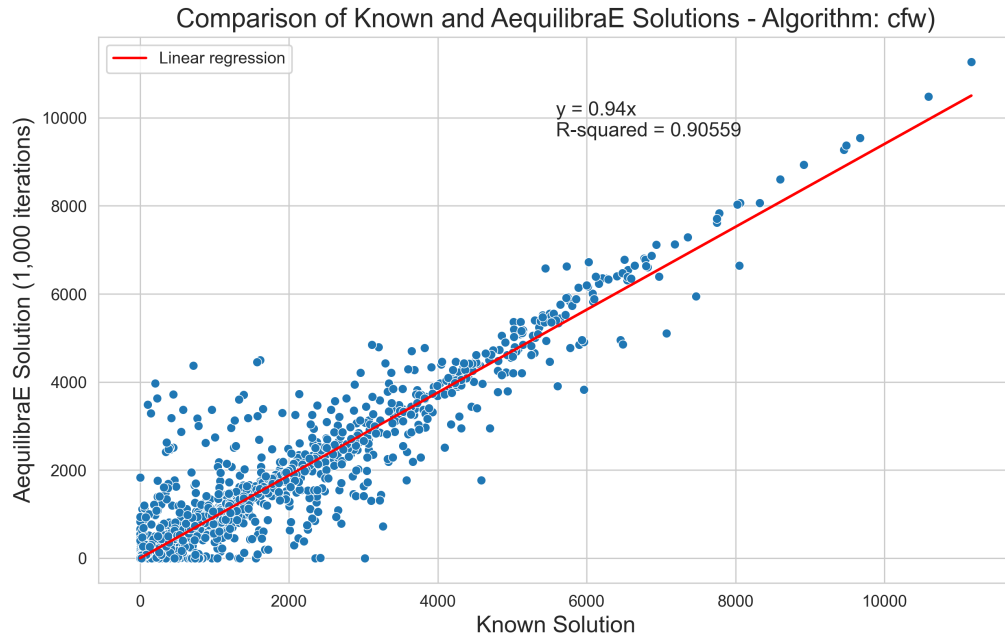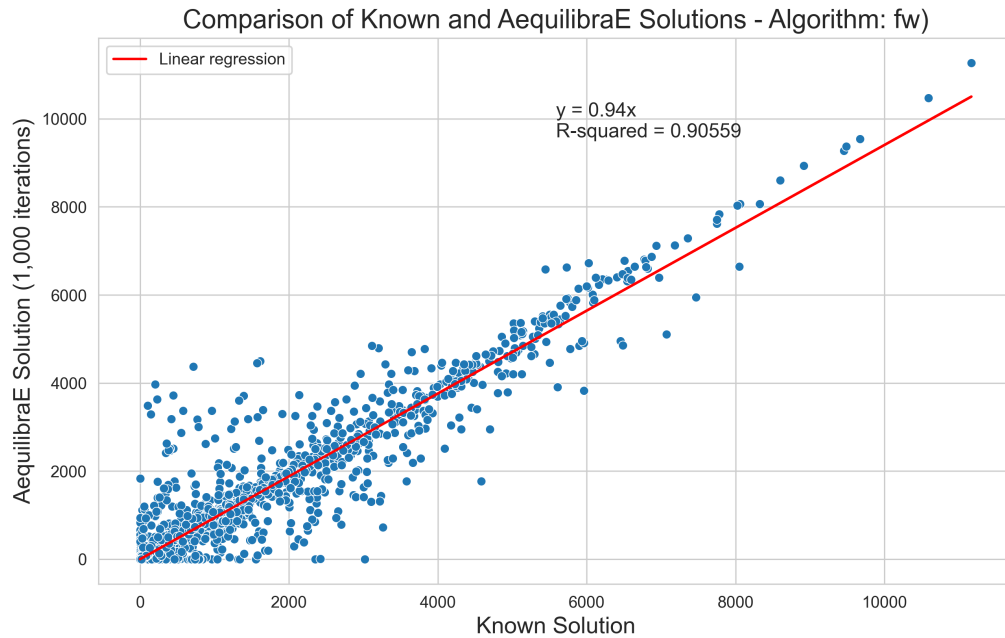
Comparison of Known and AequilibraE Solutions - Algorithm: msa)
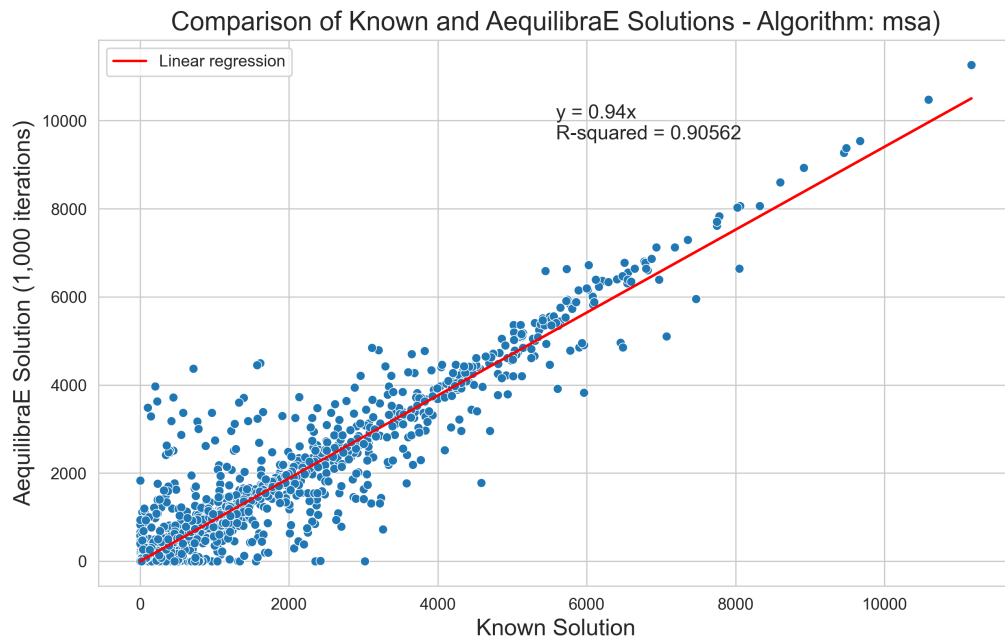
Winnipeg

Network stats

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

biconjugate Frank-Wolfe

Comparison of Known and AequilibraE Solutions - Algorithm: bfw)

Conjugate Frank-Wolfe

Frank-Wolfe

MSA
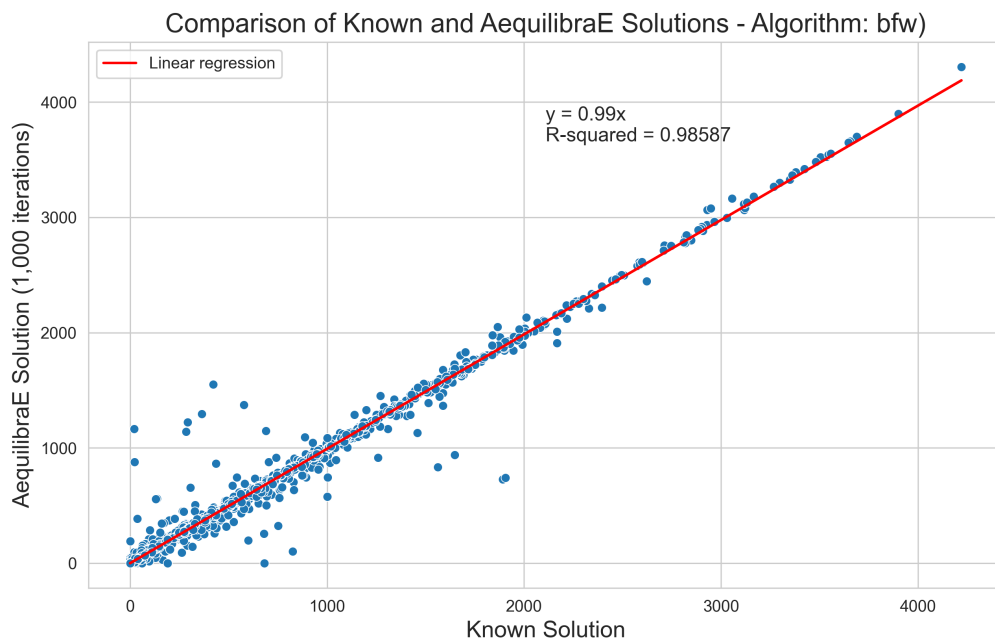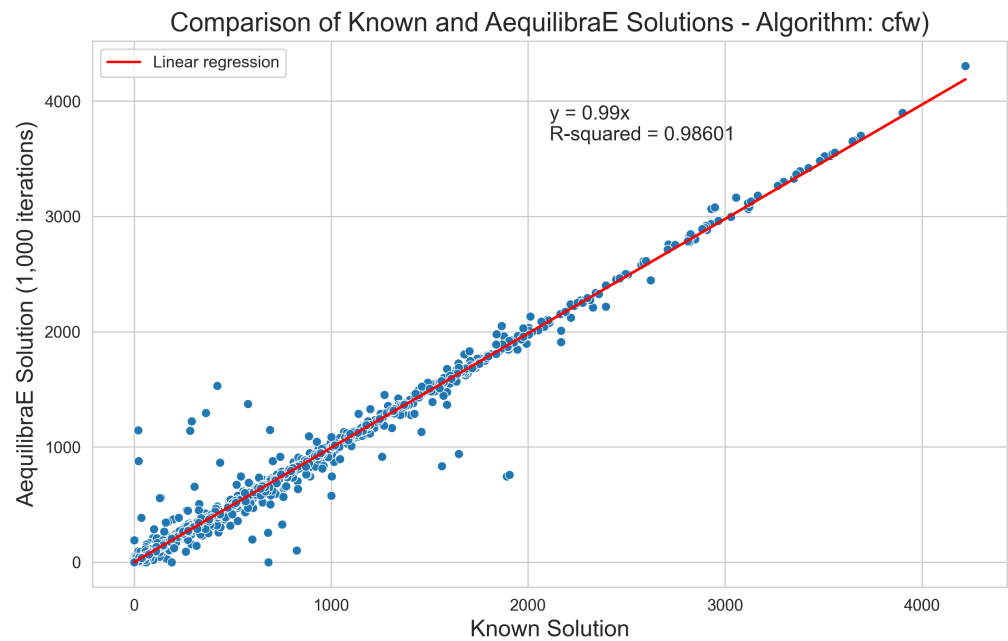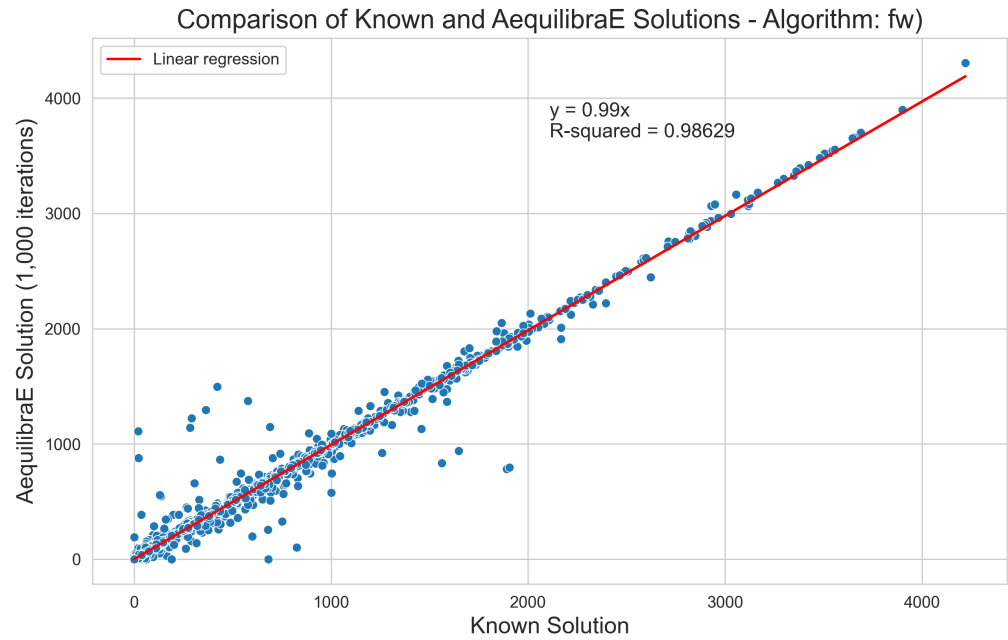
Comparison of Known and AequilibraE Solutions - Algorithm: msa)
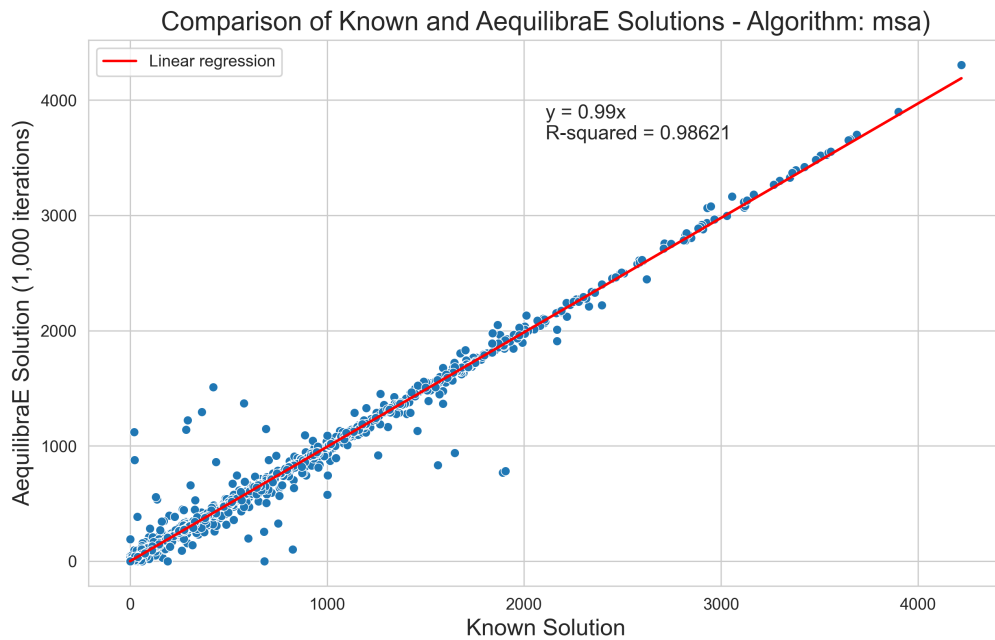
Anaheim

Network stats

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

biconjugate Frank-Wolfe

Comparison of Known and AequilibraE Solutions - Algorithm: bfw)

Conjugate Frank-Wolfe

Frank-Wolfe

MSA
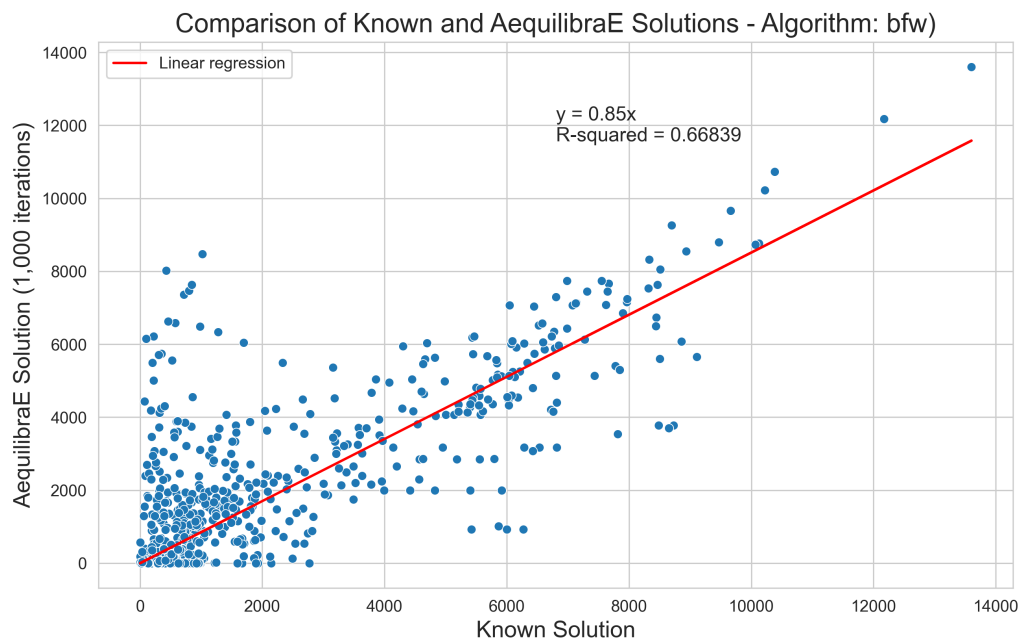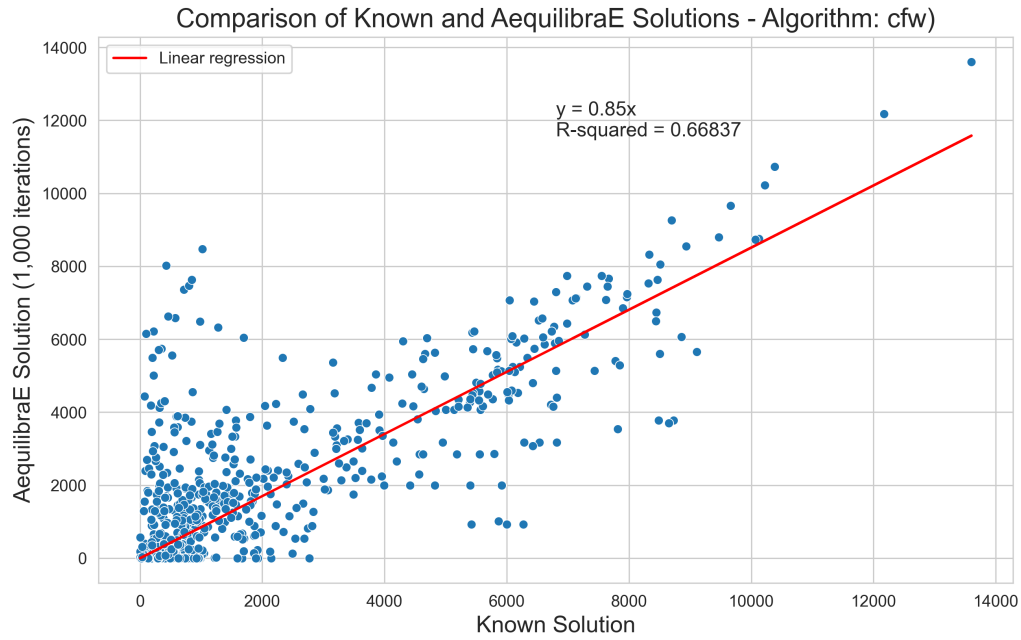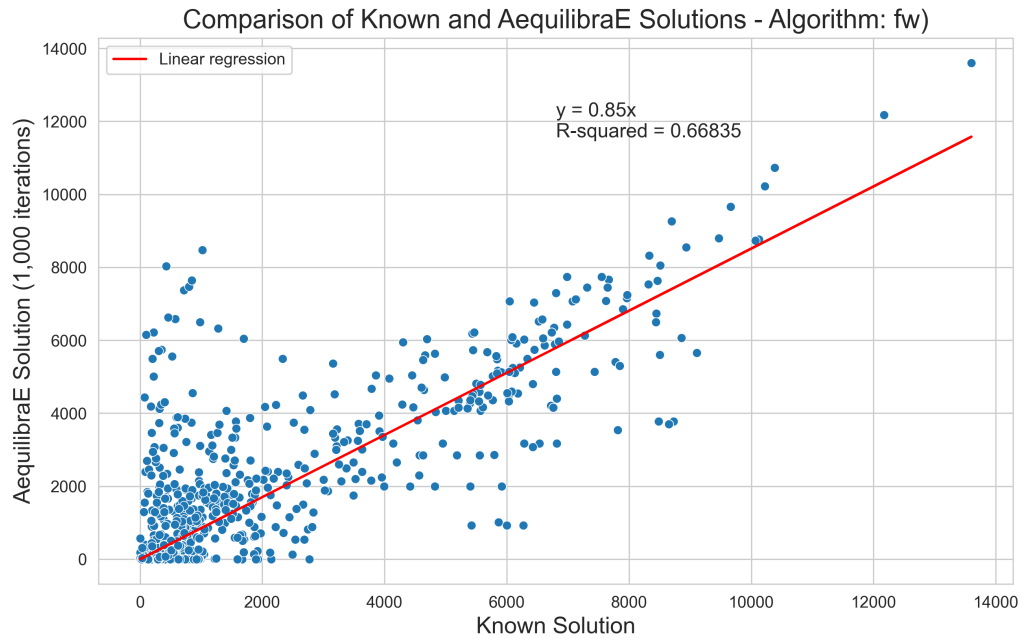
Comparison of Known and AequilibraE Solutions - Algorithm: msa)
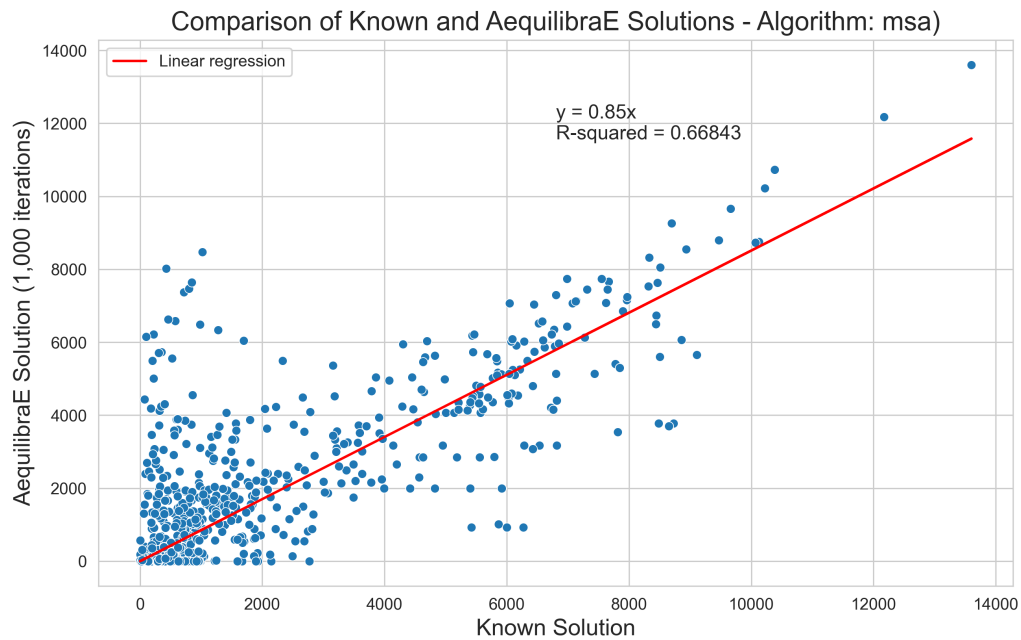
Sioux Falls

Network stats

- Links: 76

- Nodes: 24

- Zones: 24

biconjugate Frank-Wolfe

Comparison of Known and AequilibraE Solutions - Algorithm: bfw)
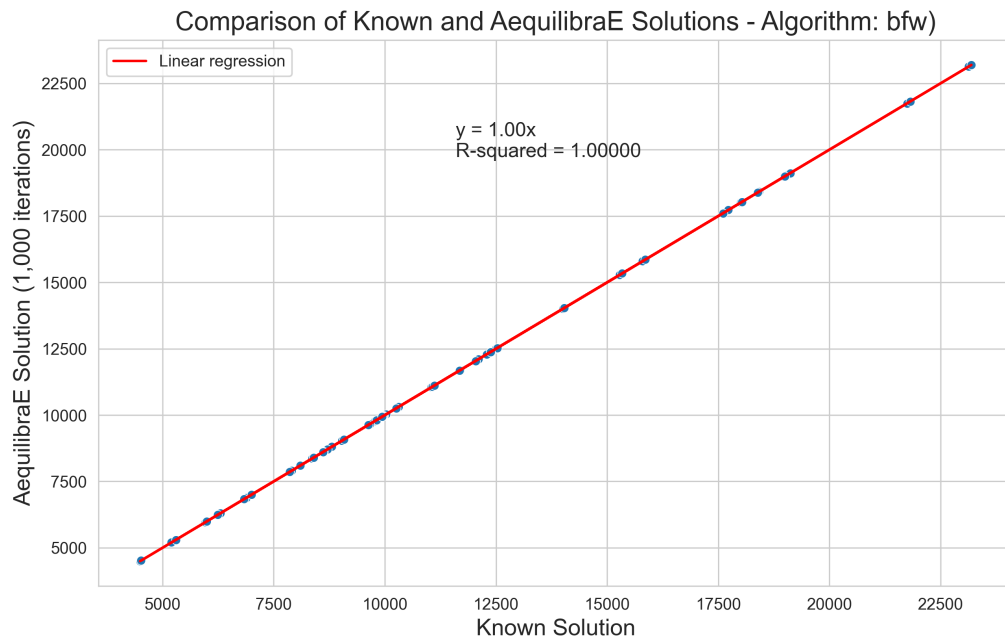
Conjugate Frank-Wolfe
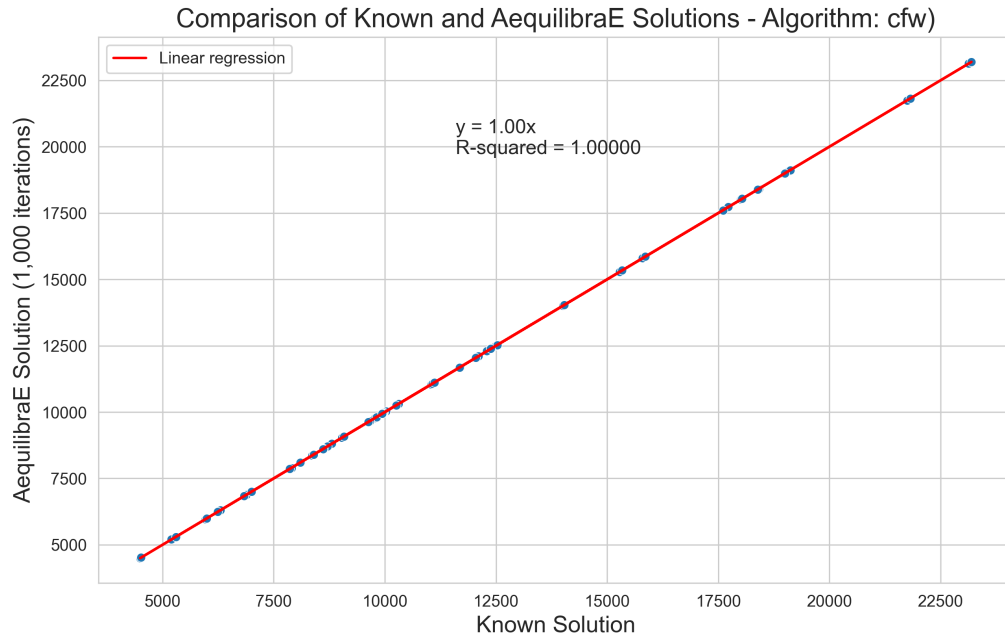
Frank-Wolfe
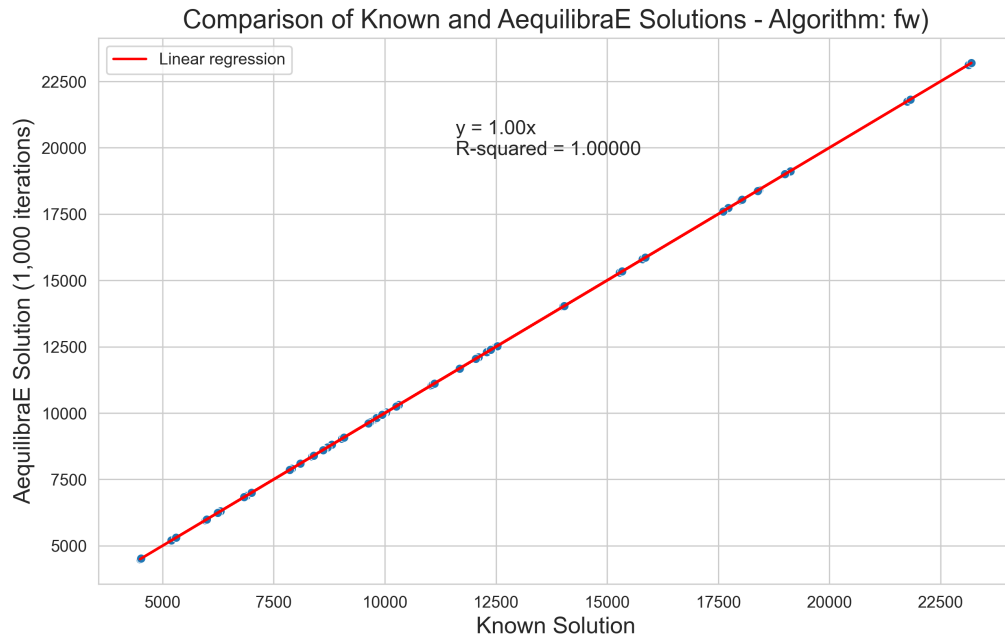
MSA
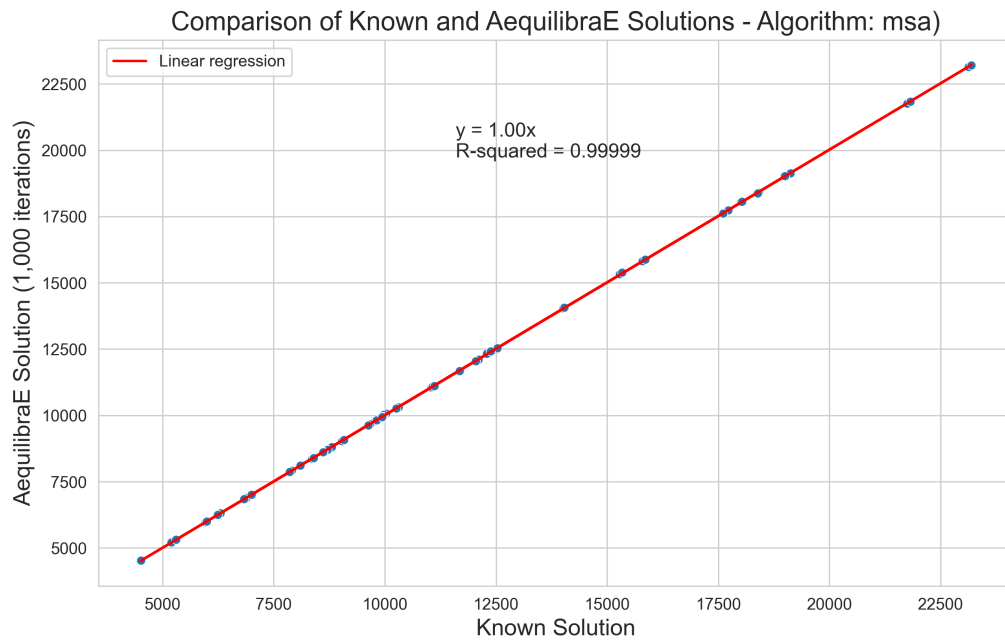
7.3.1 Convergence Study

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
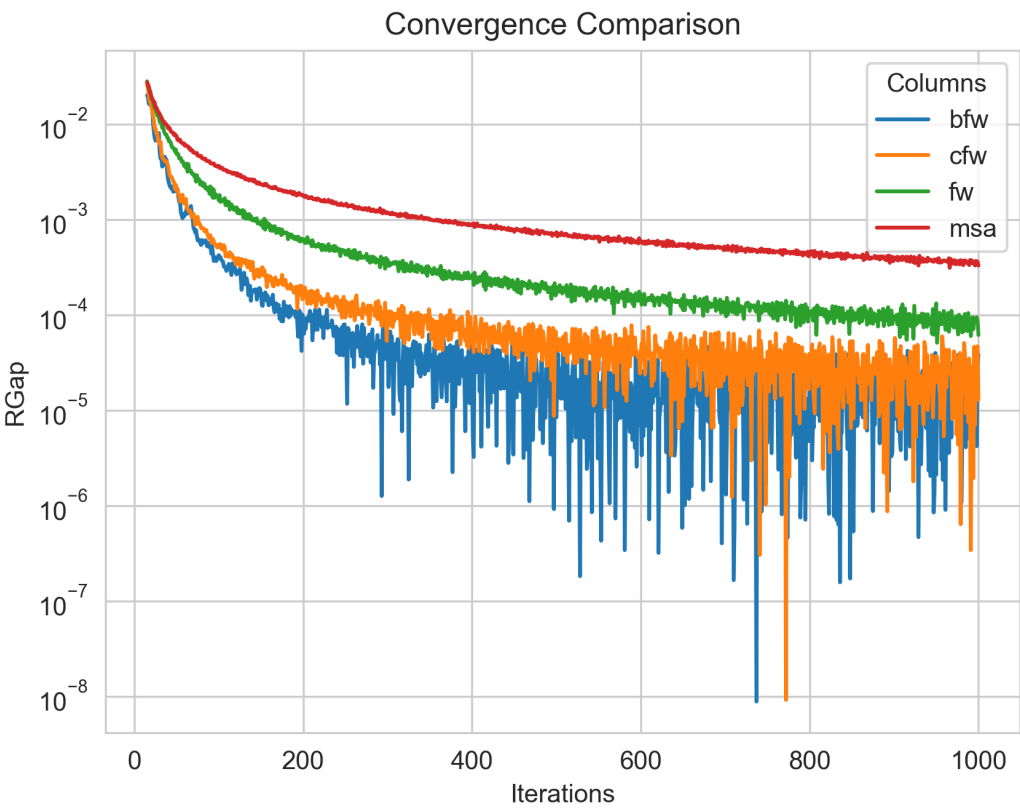
Barcelona
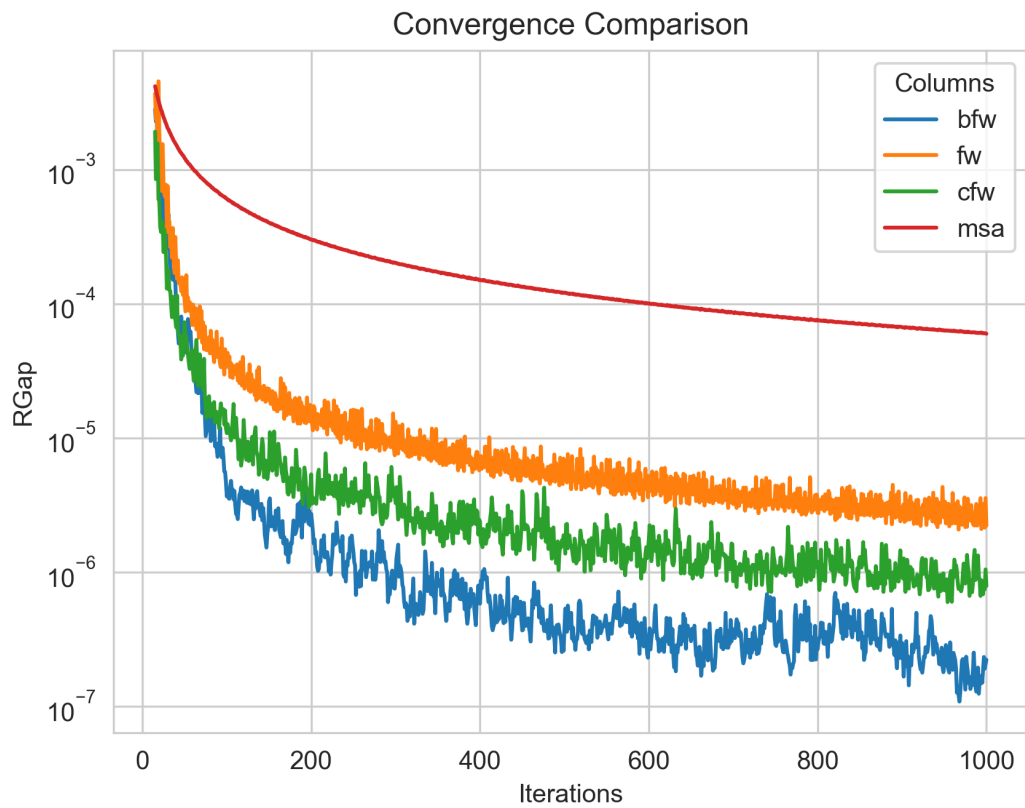
Winnipeg

Anaheim

Sioux-Falls

Convergence Comparison
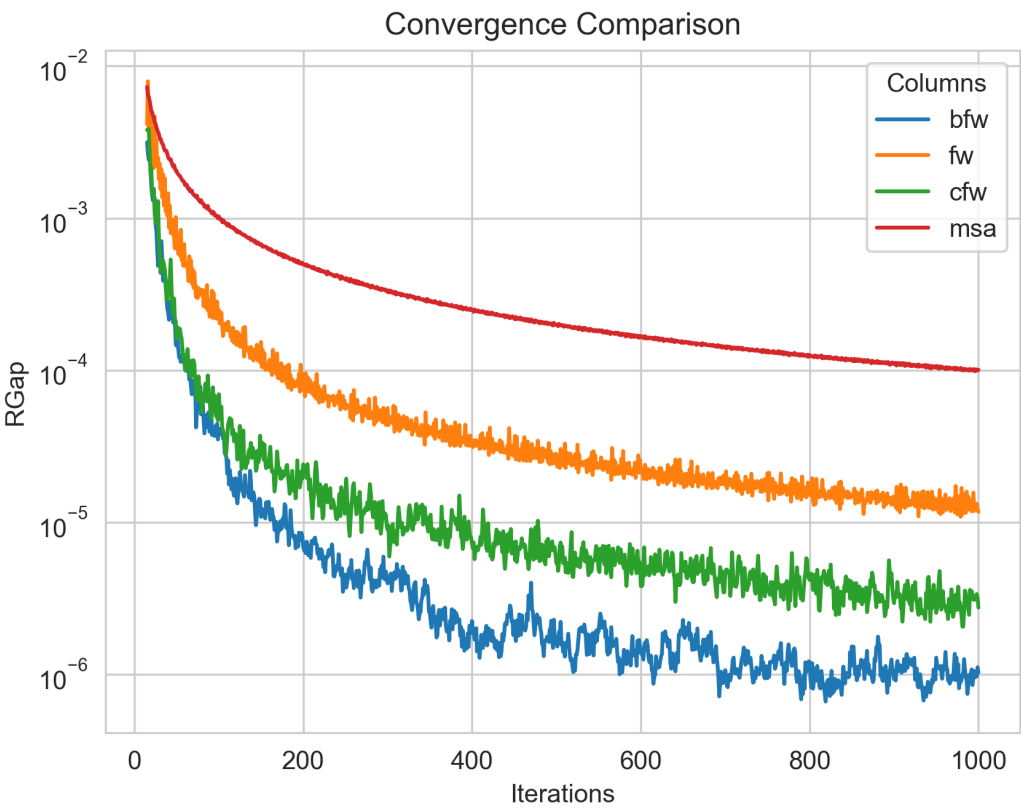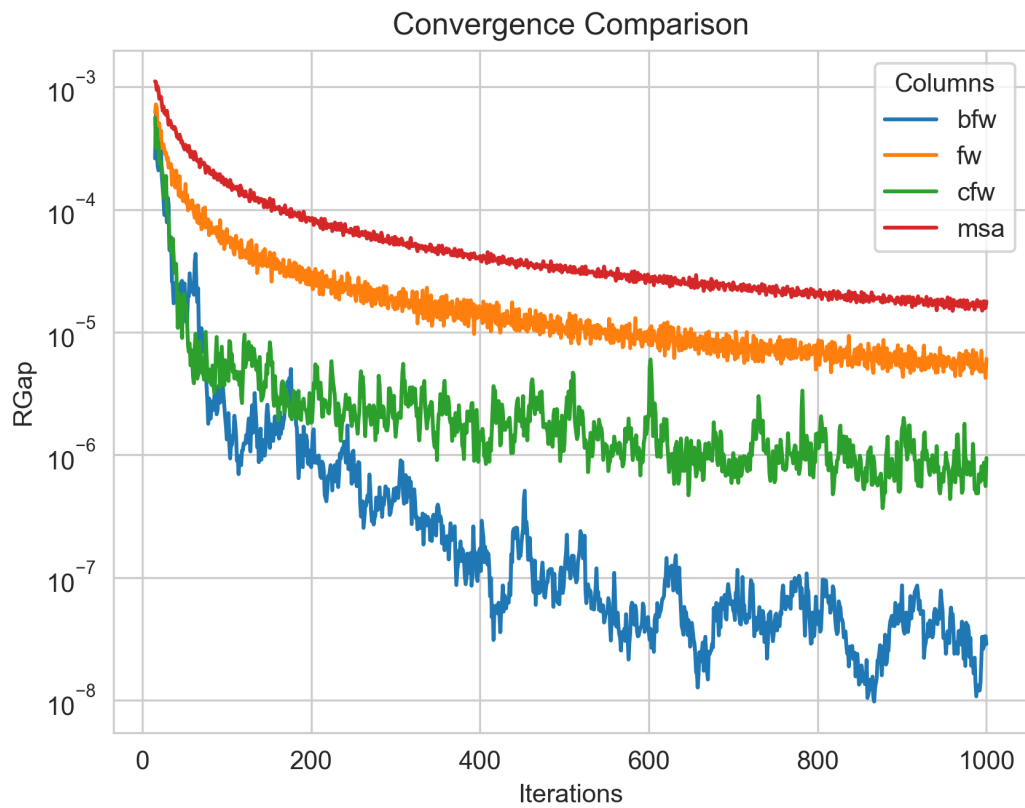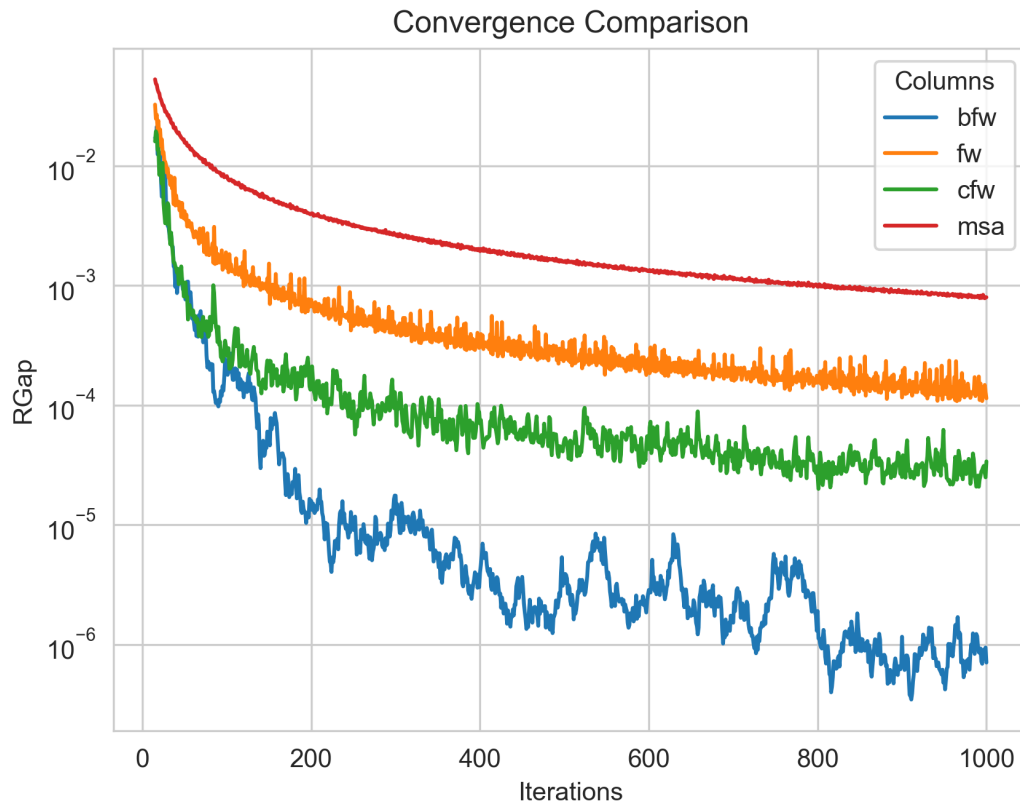
Not surprisingly, one can see that Frank-Wolfe far outperforms the Method of Successive Averages for a number of iterations larger than 25 in the case of Chicago, and is capable of reaching 1.0e-04 just after 800 iterations, while MSA is still at 3.5e-4 even after 1,000 iterations for that same case.

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.

7.3.2 Computational performance

All tests were run on a workstation equipped AMD Threadripper 3970X with 32 cores (64 threads) @ 3.7 GHz (memory use is trivial for these instances).

On this machine, AequilibraE performed 1,000 iterations of biconjugate Frank-Wolfe assignment on the Chicago Network in a little over 4 minutes, or a little less than 0.43s per iteration.

Compared with AequilibraE previous versions, we can notice a reasonable decrease in processing time.

> **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

7.3.3 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.

```python
# Imports
from pathlib import Path
from time import perf_counter

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

from aequilibrae.matrix import AequilibraeMatrix
from aequilibrae.paths import Graph
from aequilibrae.paths import TrafficAssignment
from aequilibrae.paths.traffic_class import TrafficClass


# Helper functions
def build_matrix(folder: Path, model_stub: str) -> AequilibraeMatrix:
    omx_name = folder / f"{model_stub}_trips.omx"
    if omx_name.exists():
        mat = AequilibraeMatrix()
        mat.load(omx_name)
        mat.computational_view()
        return mat

    matfile = str(folder / f"{model_stub}_trips.tntp")
    # Creating the matrix
    f = open(matfile, 'r')
    all_rows = f.read()
    blocks = all_rows.split('Origin')[1:]
    matrix = {}
    for k in range(len(blocks)):
        orig = blocks[k].split('\n')
        dests = orig[1:]
        orig = int(orig[0])

        d = [eval('{' + a.replace(';', ',').replace(' ', '') + '}') for a in dests]
        destinations = {}
        for i in d:
            destinations = {**destinations, **i}
        matrix[orig] = destinations
    zones = max(matrix.keys())
    index = np.arange(zones) + 1
    mat_data = np.zeros((zones, zones))
    for i in range(zones):
        for j in range(zones):
```

(continues on next page)

```python
            mat_data[i, j] = matrix[i + 1].get(j + 1, 0)

    # Let's save our matrix in AequilibraE Matrix format
    mat = AequilibraeMatrix()
    mat.create_empty(zones=zones, matrix_names=['matrix'], memory_only=True)
    mat.matrix['matrix'][:, :] = mat_data[:, :]
    mat.index[:] = index[:]
    mat.computational_view(["matrix"])
    mat.export(str(omx_name))
    return mat


# Now let's parse the network
def build_graph(folder: Path, model_stub: str, centroids: np.array) -> Graph:
    net = pd.read_csv(folder / f"{model_stub}_net.tntp", skiprows=7, sep='\t')
    cols = ['init_node', 'term_node', 'free_flow_time', 'capacity', "b", "power"]
    if 'toll' in net.columns:
        cols.append('toll')
    network = net[cols]
    network.columns = ['a_node', 'b_node', 'free_flow_time', 'capacity', "b", "power",
→ "toll"]
    network = network.assign(direction=1)
    network["link_id"] = network.index + 1
    network.free_flow_time = network.free_flow_time.astype(np.float64)

    # If you want to create an AequilibraE matrix for computation, then it follows
    g = Graph()
    g.cost = net['free_flow_time'].values
    g.capacity = net['capacity'].values
    g.free_flow_time = net['free_flow_time'].values

    g.network = network
    g.network.loc[(g.network.power < 1), "power"] = 1
    g.network.loc[(g.network.free_flow_time == 0), "free_flow_time"] = 0.01
    g.prepare_graph(centroids)
    g.set_graph("free_flow_time")
    g.set_skimming(["free_flow_time"])
    g.set_blocked_centroid_flows(False)
    return g

def known_results(folder: Path, model_stub: str) -> pd.DataFrame:
    df = pd.read_csv(folder / f"{model_stub}_flow.tntp", sep='\t')
    df.columns = ["a_node", "b_node", "TNTP Solution", "cost"]
    return df


# Let's run the assignment
def assign(g: Graph, mat: AequilibraeMatrix, algorithm: str):
    assigclass = TrafficClass("car", g, mat)
    if "toll" in g.network.columns:
        assigclass.set_fixed_cost("toll")

    assig = TrafficAssignment()
    assig.set_classes([assigclass])
```

```python
    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.max_iter = 1000
    assig.rgap_target = 1e-10 # Nearly guarantees that convergence won't be reached
    assig.set_algorithm(algorithm)
    assig.execute()
    return assig


# We compare the results
def validate(assig: TrafficAssignment, known_flows: pd.DataFrame, algorithm: str,
→folder: Path, model_name):
    modeled = g.network[["link_id", "a_node", "b_node"]].merge(assig.results().matrix_
→ab.reset_index(),
                                                               on="link_id").rename(
        columns={"matrix_ab": "AequilibraE Solution"})
    merged = known_flows.merge(modeled, on=["a_node", "b_node"])

    # Scatter plot
    plt.figure(figsize=(10, 6))
    sns.scatterplot(data=merged, x="TNTP Solution", y="AequilibraE Solution", s=30)

    # Linear regression
    X = merged["TNTP Solution"].values.reshape(-1, 1)
    y = merged["AequilibraE Solution"].values
    reg = LinearRegression(fit_intercept=False).fit(X, y)
    y_pred = reg.predict(X)
    r_squared = r2_score(y, y_pred)

    # Plot regression line
    plt.plot(merged["TNTP Solution"], y_pred, color='red', label='Linear regression')

    # Customize the plot
    plt.title(f'Comparison of Known and AequilibraE Solutions - Algorithm: {algorithm}
→', fontsize=16)
    plt.xlabel('Known Solution', fontsize=14)
    plt.ylabel('AequilibraE Solution (1,000 iterations)', fontsize=14)

    # Display the equation and R-squared on the plot
    equation_text = f'y = {reg.coef_[0]:.2f}x\nR-squared = {r_squared:.5f}'
    plt.text(x=merged["TNTP Solution"].max() * 0.5, y=merged["AequilibraE Solution"].
→max() * 0.85, s=equation_text,
             fontsize=12)

    plt.legend()
    plt.savefig(folder / f"{model_name}_{algorithm}-1000_iter.png", dpi=300)
    plt.close()


def assign_and_validate(g: Graph, mat: AequilibraeMatrix, folder: Path, model_stub:
→str):
    known_flows = known_results(folder, model_stub)
```

```python
    # We run the traffic assignment
    conv = None
    for algorithm in ["bfw", "cfw", "fw", "msa"]:
        t = -perf_counter()
        assig = assign(g, mat, algorithm)
        t += perf_counter()
        print(f"{model_stub},{algorithm},{t:0.4f}")

        res = assig.report()[["iteration", "rgap"]].rename(columns={"rgap": algorithm}
↪)

        validate(assig, known_flows, algorithm, folder, model_stub)

        conv = res if conv is None else conv.merge(res, on="iteration")
    df = conv.replace(np.inf, 1).set_index("iteration")
    convergence_chart(df, data_folder, model_stub)
    df.to_csv(folder / f"{model_stub}_convergence.csv")

def convergence_chart(df: pd.DataFrame, folder: Path, model_name):
    import matplotlib.pyplot as plt

    plt.cla()
    df = df.loc[15:, :]
    for column in df.columns:
        plt.plot(df.index, df[column], label=column)
    # Customize the plot
    plt.title('Convergence Comparison')
    plt.xlabel('Iterations')
    plt.ylabel('RGap')
    plt.yscale("log")
    plt.legend(title='Columns')
    plt.savefig(folder / f"convergence_comparison_{model_name}.png", dpi=300)

models = {"chicago": [Path(r'D:\src\TransportationNetworks\chicago-regional'),
↪"ChicagoRegional"],
    "sioux_falls": [Path(r'D:\src\TransportationNetworks\SiouxFalls'), "SiouxFalls
↪"],
      "anaheim": [Path(r'D:\src\TransportationNetworks\Anaheim'), "Anaheim"],
      "winnipeg": [Path(r'D:\src\TransportationNetworks\Winnipeg'), "Winnipeg"],
      "barcelona": [Path(r'D:\src\TransportationNetworks\Barcelona'), "Barcelona"],
    }

convergence = {}
for model_name, (data_folder, model_stub) in models.items():
    print(model_name)
    mat = build_matrix(data_folder, model_stub)
    g = build_graph(data_folder, model_stub, mat.index)
    assign_and_validate(g, mat, data_folder, model_stub)
```

7.4 Examples

7.4.1 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.TrafficClass()*
> - *aequilibrae.paths.TrafficAssignment()*

```python
# 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
```

```python
# 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

```python
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
```

(continues on next page)

```
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
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.TrafficClass()*
> - *aequilibrae.paths.TrafficAssignment()*
> - *aequilibrae.matrix.AequilibraeMatrix()*

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

from aequilibrae.matrix import AequilibraeMatrix
from aequilibrae.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.TrafficClass()*
> - *aequilibrae.paths.TrafficAssignment()*
> - *aequilibrae.distribution.Ipf()*
> - *aequilibrae.distribution.GravityCalibration()*
> - *aequilibrae.distribution.GravityApplication()*
> - *aequilibrae.distribution.SyntheticGravityModel()*

```python
# 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
```

```python
# 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.

```python
from aequilibrae.paths import TrafficAssignment, TrafficClass
```

```python
# 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")

# 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.

```python
from math import log10, floor
import matplotlib.pyplot as plt
```

```python
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

```python
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_aem")
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

```python
for function in ["power", "expo"]:
    gc = GravityCalibration(matrix=demand, impedance=imped, function=function, nan_as_
↪zero=True)
    gc.calibrate()
    model = gc.model
    # We save the model
    model.save(join(fldr, f"{function}_model.mod"))

    _ = 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 recored

```
ipf.save_to_project(name="demand_ipfd", file_name="demand_ipfd.aem")
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

```python
for function in ["power", "expo"]:
    model = SyntheticGravityModel()
    model.load(join(fldr, f"{function}_model.mod"))

    outmatrix = join(proj_matrices.fldr, f"demand_{function}_model.aem")
    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

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

Traffic assignment with Select Link Analysis

We'll perform traffic assignment for the future year.

```python
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`.

```python
demand = proj_matrices.get_matrix("demand_ipfd")
demand.names
```

Let's prepare our data for computation

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

The future year assignment is quite similar to the one we did for the base-year.

```python
# 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)

# 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

```python
assig.save_results("future_year_assignment")
```

And save the skims

```python
assig.save_skims("future_year_assignment_skims", which_ones="all", format="omx")
```

Run convergence study

```python
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

```python
project.close()
```

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).
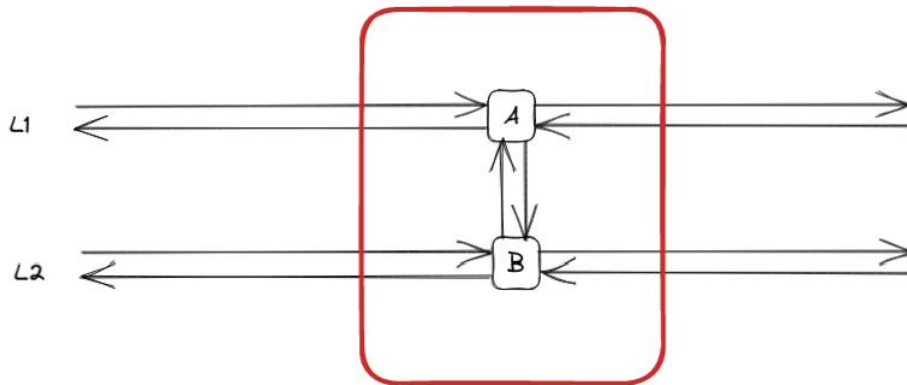
- **Transit lines**: a transit line is a set of services that may use different routes, decomposed into segments.

- **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').
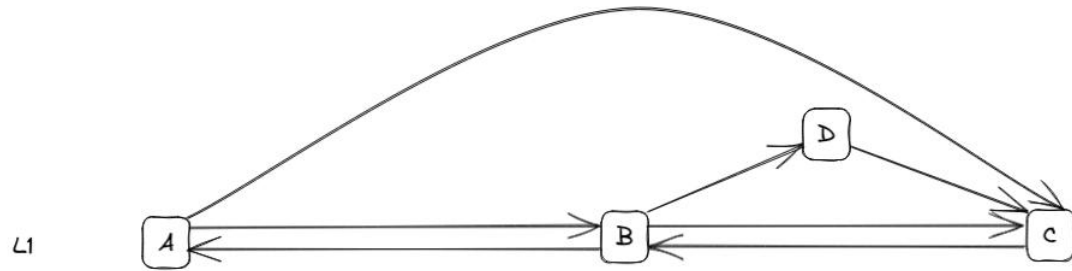
Lines can also be decomposed into multiple sub-lines, each representing distinct routes. For the given example, we may have several sub-lines under the same commercial line (L1).

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.
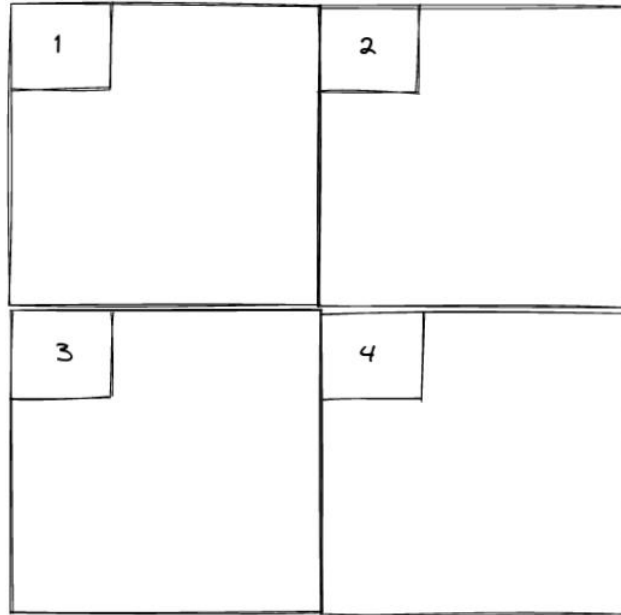
- **Connectors**: connectors are special network nodes that facilitate the connection between supply and demand.
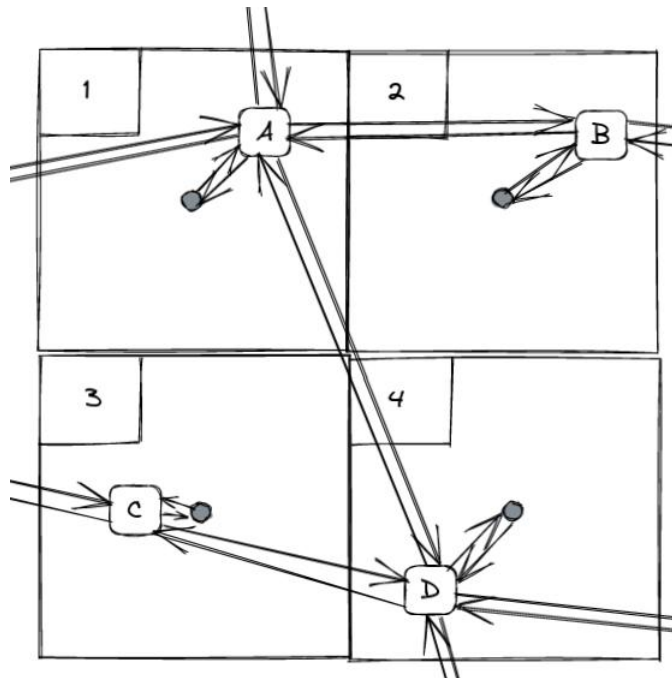
8.1.2 The assignment graph

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.
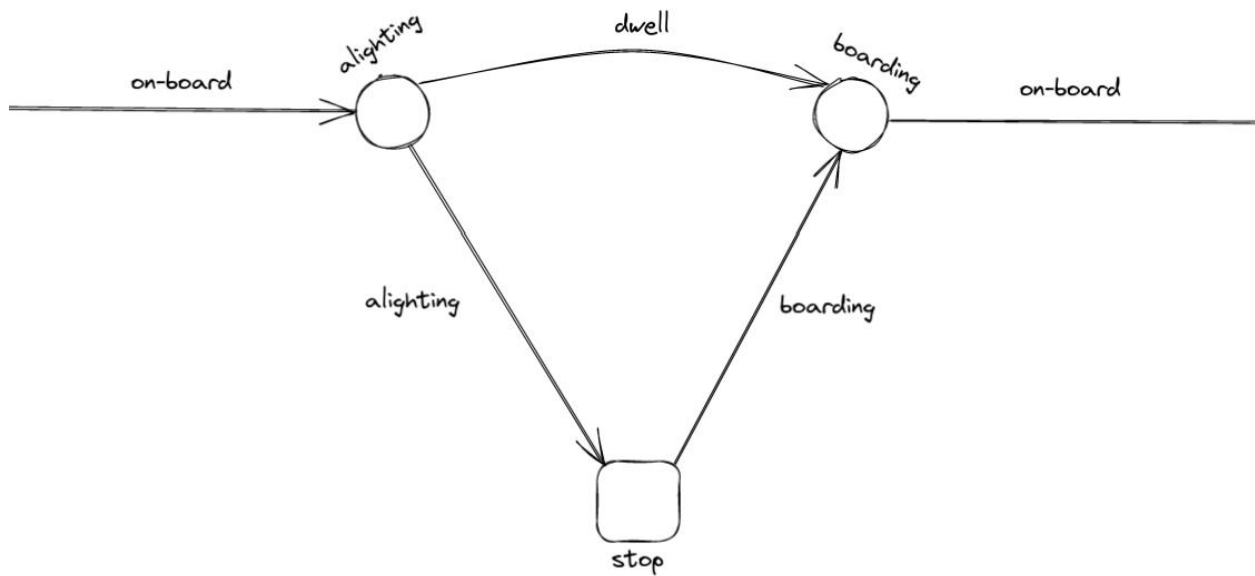
Transfer links enable to compute the passenger flow count between line couples at the same stop. These links can be extended between all lines of a station if an increase in the number of links is viable.
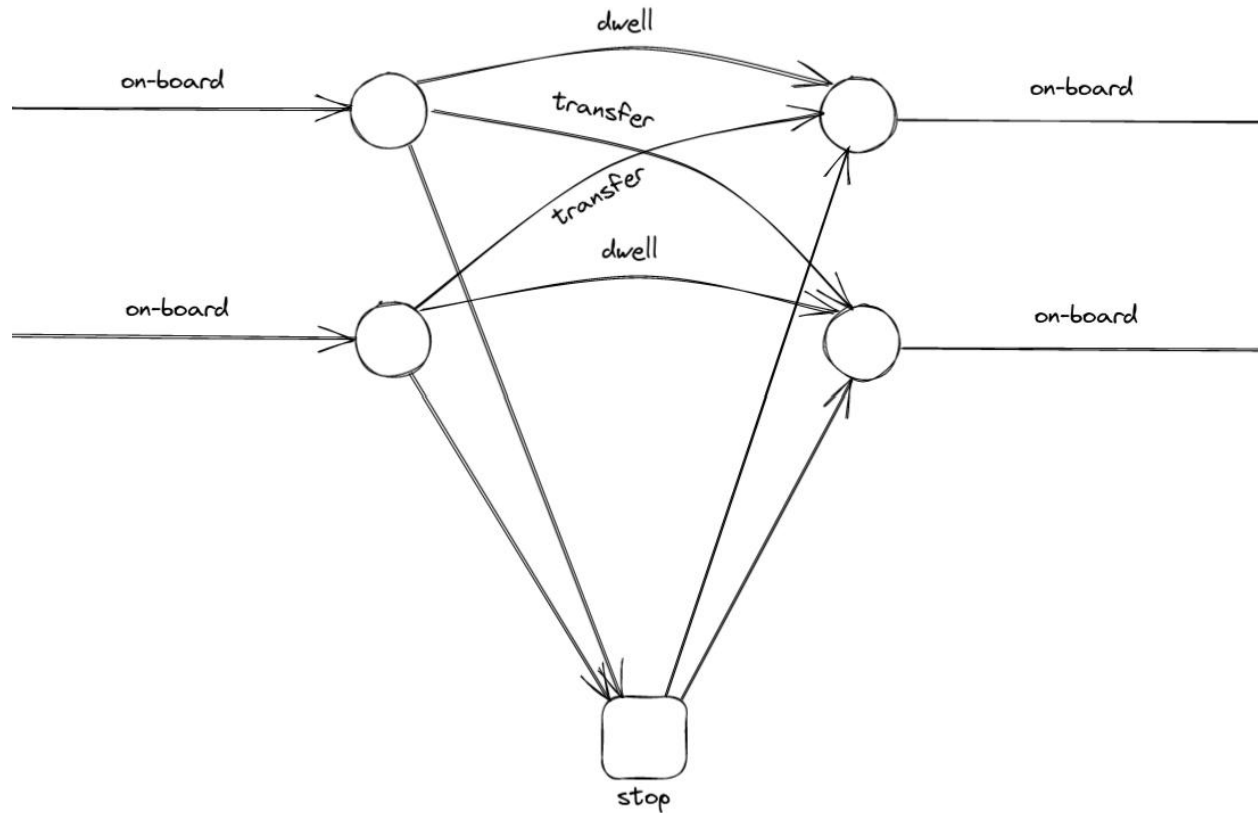
Walking links connect *stop* nodes within a station, while *connector* links connect the zone centroids (OD nodes) to *stop* nodes. Connectors that connect OD to *stop* nodes allow passengers to access the network, while connectors in the opposite direction allow them to egress. Walking nodes/links may also be used to connect stops from distant stations.
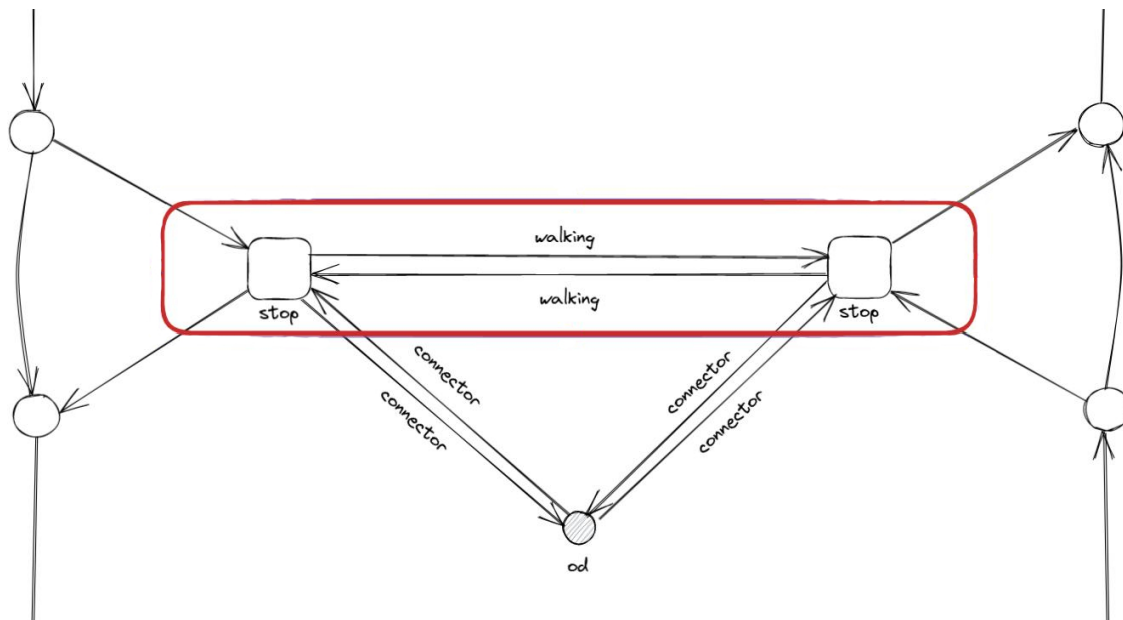
The table below summarizes link characteristics and attributes based on link types:

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 161, 1].

The following figure presents the travel times for each line.
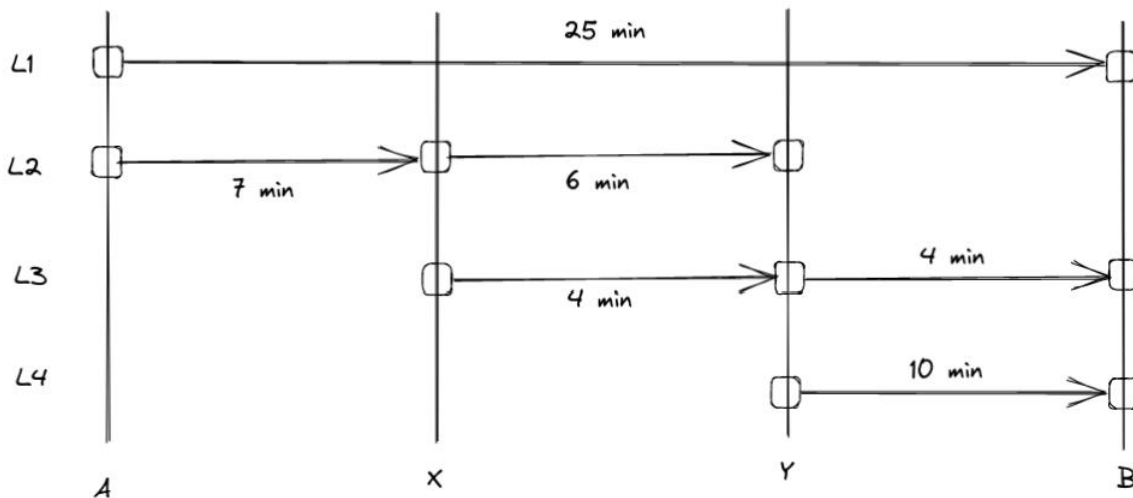
We have the following four distinct line characteristics:

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:
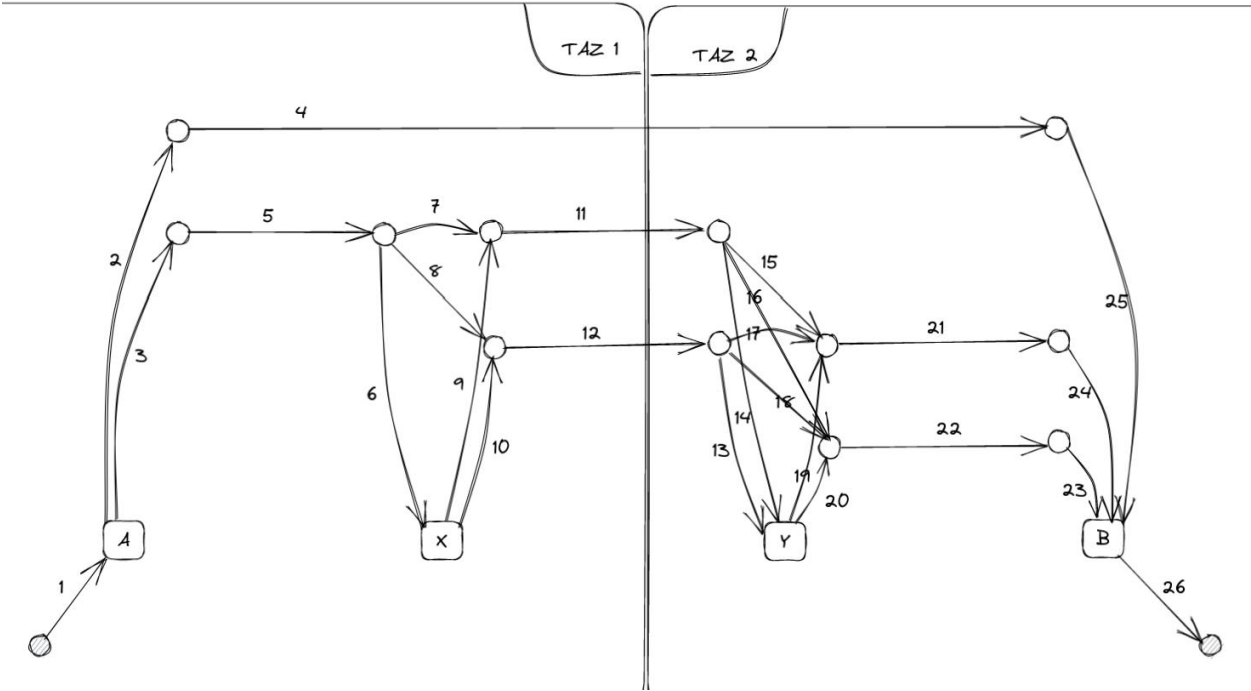
Here is a table listing all links:

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 170, 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:

$$f_a = \begin{cases} 1/ (\alpha \ d_a) & \text{if } \alpha \ d_a \neq 0 \\ \infty & \text{otherwise} \end{cases}$$

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

```python
# 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.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
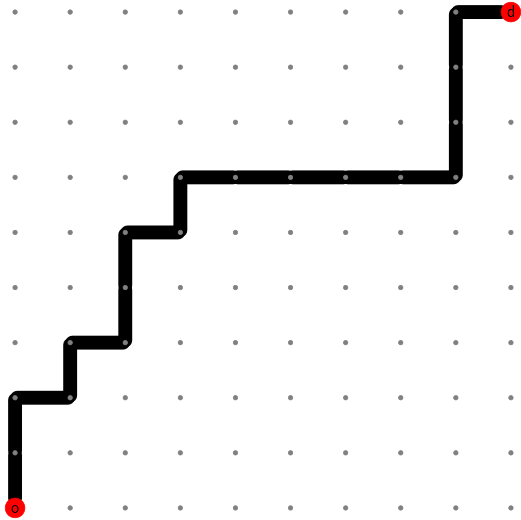
Shortest path - Bell's network
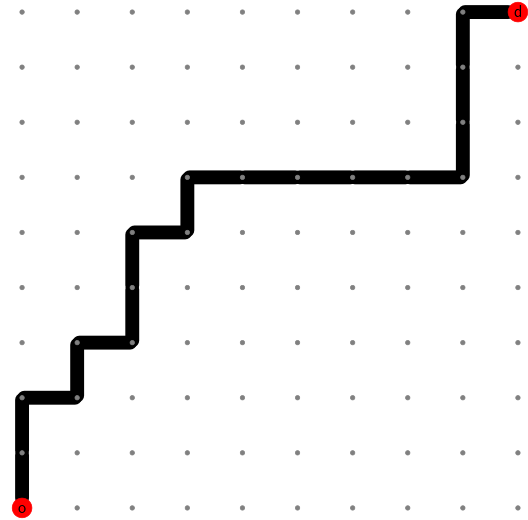
Shortest hyperpath - Bell's network α=0.5
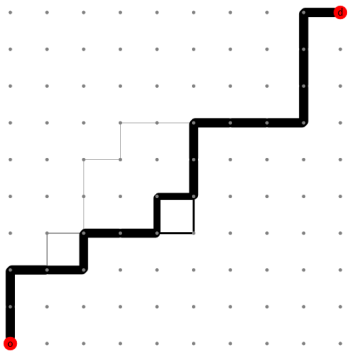
Shortest hyperpath - Bell's network α=1.0
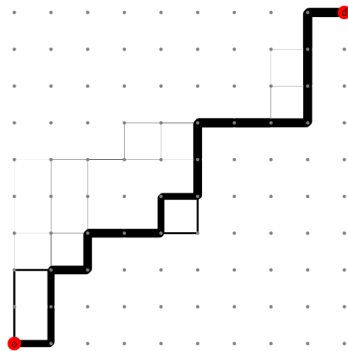
Shortest hyperpath - Bell's network α=100.0
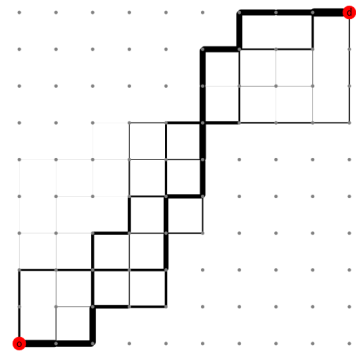
```python
FIGURE_SIZE = (6, 6)  # figure size


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
```

```python
    # 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,
```

```
        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)
```

```
>>> 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(
```

```
...        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.GTFSRouteSystemBuilder()*

```python
# 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.project.database_connection import database_connection
from aequilibrae.transit import Transit
from aequilibrae.utils.create_example import create_example
```

```python
# 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.

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

Let's import the GTFS feed.

```python
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.

```python
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.

```python
transit.load_date("2016-04-13")

# Now we execute the map matching to find the real paths.
# Depending on the GTFS size, this process can be really time-consuming.

# transit.set_allow_map_match(True)
# transit.map_match()
```

```
# Finally, we save our GTFS into our model.
transit.save_to_disk()
```

Now we will plot one of the route's patterns we just imported

```
conn = database_connection("transit")

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.TransitGraphBuilder()*

- *aequilibrae.paths.TransitClass()*

- *aequilibrae.paths.TransitAssignment()*

- *aequilibrae.matrix.AequilibraeMatrix()*

Imports for example construction

```python
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.project.database_connection import database_connection
from aequilibrae.transit import Transit
from aequilibrae.transit.transit_graph_builder import TransitGraphBuilder
from aequilibrae.utils.create_example import create_example
```

```python
# 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.

```python
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()`.

```python
graph = data.create_graph(
    with_outer_stop_transfers=False,
    with_walking_edges=False,
    blocking_centroid_flows=False,
    connector_method="overlapping_regions"
)
```

Connector project matching

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

(continues on next page)

```python
# Reading back into AequilibraE
pt_con = database_connection("transit")
graph_db = TransitGraphBuilder.from_db(pt_con, project.network.periods.default_period.
 ↪period_id)
graph_db.vertices.drop(columns="geometry")

# To perform an assignment we need to convert the graph builder into a graph.
transit_graph = graph_db.to_transit_graph()
```

```python
# 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`.

```python
# 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.

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

Wrapping up

```python
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.TransitGraphBuilder()*
> - *aequilibrae.paths.TransitClass()*
> - *aequilibrae.paths.TransitAssignment()*
> - *aequilibrae.matrix.AequilibraeMatrix()*

```python
# 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.

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

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

Let's create our `Transit` object.

```python
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()`.

```python
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)

```
# 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.project.database_connection import database_connection
from aequilibrae.transit.transit_graph_builder import TransitGraphBuilder
```

```
pt_con = database_connection("transit")


graph_db = TransitGraphBuilder.from_db(pt_con, 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.

```python
import numpy as np
from aequilibrae.matrix import AequilibraeMatrix
```

```python
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`

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

Create the assignment class

```python
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

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.
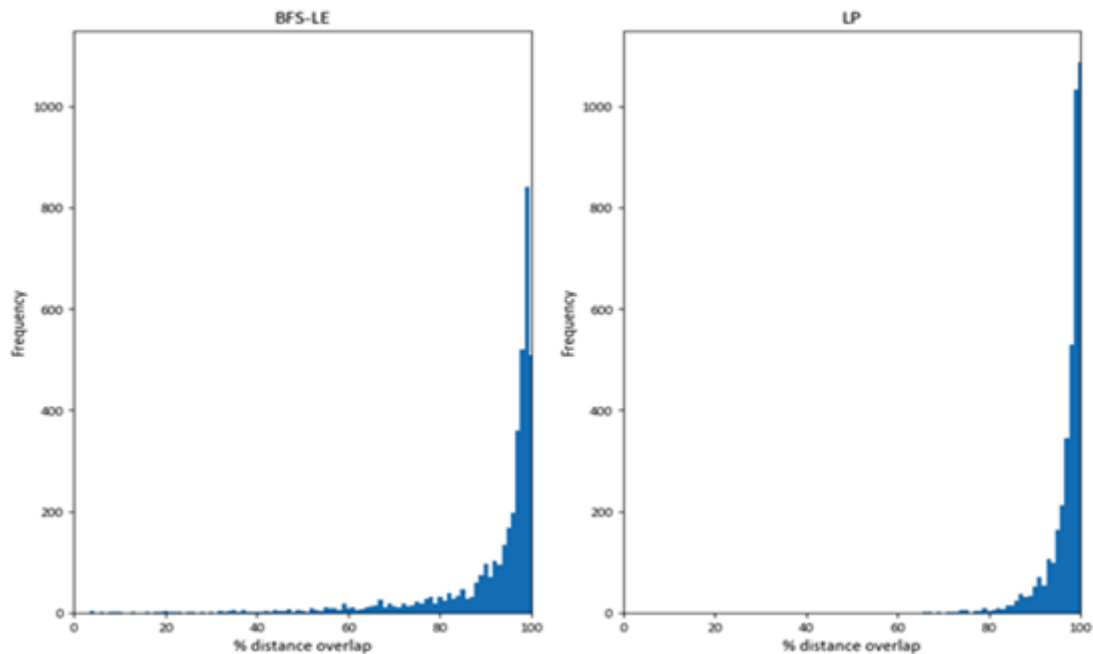
[1] Rieser-Schüssler, N., Balmer, M., and Axhausen, K.W. (2012). Route choice sets for very high-resolution data. Transportmetrica A: Transport Science, 9(9), 825–845. https://doi.org/10.1080/18128602.2012.671383

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.RouteChoice()*

```python
# 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
```

```python
# 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

```python
od_pairs_of_interest = [(71645, 79385), (77011, 74089)]
nodes_of_interest = (71645, 74089, 77011, 79385)
```

Let's build all graphs

```python
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 preform 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

```python
# 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(plot_routes[0])]
route_2 = links[links.link_id.isin(plot_routes[1])]
route_3 = links[links.link_id.isin(plot_routes[2])]
route_4 = links[links.link_id.isin(plot_routes[3])]
route_5 = links[links.link_id.isin(plot_routes[4])]

nodes = project.network.nodes.data
nodes = nodes[nodes["node_id"].isin([77011, 74089])]
```

```python
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
```

```python
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.RouteChoice()*
>
>   • *aequilibrae.matrix.AequilibraeMatrix()*

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

(continues on next page)

```python
from os.path import join

from aequilibrae.utils.create_example import create_example
```

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

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

```python
import logging
import sys
```

```python
# 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

```python
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 $beta$.

```python
# Distance factor
theta = 0.00011

# PSL parameter
beta = 1.1
```

Let's select a set of nodes of interest

```python
nodes_of_interest = (71645, 74089, 77011, 79385)
```

Let's build all graphs

```python
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

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

We grab the graph for cars

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

```python
# 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.

```python
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.

```python
import folium
```

```python
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,
            }
        },
```

```
    )
```

Route Choice class

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

```python
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.

```python
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 preformed.

```python
rc.add_demand(mat)
```

It is highly recommended to set either `max_routes` or `max_depth` to prevent runaway results.

```python
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.

```python
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.

```python
res = rc.get_results()
res.head()
```

```python
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.

```python
rc.prepare()
```

Now we can perform a batch computation with an assignment

```python
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.RouteChoice()*
>
> - *aequilibrae.paths.SubAreaAnalysis()*
>
> - *aequilibrae.matrix.AequilibraeMatrix()*

```python
# 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
```

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

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

```python
import logging
import sys
```

```python
# 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 $beta$.

```python
theta = 0.011  # Distance factor

beta = 1.1  # PSL parameter
```

Let's build all graphs

```python
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

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

We also see what graphs are available

```python
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.

```python
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.

```python
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

```python
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

```python
poly = zones.union_all()
poly
```

And prepare the sub-area to plot.

```python
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

```python
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={
```

```
        "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 preformed.

```
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.

```python
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.

```python
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

OTHER APPLICATIONS

In this section, we bring some of AequilibraE's applications that do not match a specific subject.

10.1 Examples

10.1.1 Other applcations

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()*

```python
# 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
```

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

project = create_example(fldr)
```

Get the Delaunay Lines generation class

```python
da = DelaunayAnalysis(project)
```

(continues on next page)

```
# 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

```
conn = sqlite3.connect(join(fldr, "results_database.sqlite"))
results = pd.read_sql("Select * from delaunay_test", conn).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*

```python
# 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

```python
print(list_examples())
```

```python
# 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

```python
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.

```python
network = project.network
```

So we use the convenient network method `convex_hull()` (it may take some time for very large networks)

```python
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

```python
zone_area = geo.area / zones
```

Since the area of the hexagon is $\frac{3\sqrt{3}}{2} * side^2$ the side is equal to $\sqrt{\frac{2\sqrt{3}*area}{9}}$

```python
zone_side = sqrt(2 * sqrt(3) * zone_area / 9)
```

Now we can run an SQL query to compute the hexagonal grid. There are many ways to create hex bins (including with a GUI on QGIS), but we find that using SpatiaLite is a pretty neat solution, for which we will use the entire network bounding box to make sure we cover everything.

```
extent = network.extent()
```

```
b = extent.bounds
sql = "select st_asbinary(HexagonalGrid(GeomFromWKB(?), ?, 0, GeomFromWKB(?)))"
with project.db_connection as conn:
    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()
```

API REFERENCE

11.1 Project

Project()	AequilibraE project class

11.1.1 aequilibrae.project.Project

class aequilibrae.project.**Project**

AequilibraE project class

Listing 1: Create Project

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

Listing 2: Open Project

```
>>> existing_project = Project()
>>> existing_project.open(project_path)
```

__init__()

Methods

__init__()	
activate()	
check_file_indices()	Makes results_database.sqlite and the matrices folder compatible with project database
close()	Safely closes the project
deactivate()	
from_path(project_folder)	
log()	Returns a log object
new(project_path)	Creates a new project
open(project_path)	Loads project from disk
upgrade()	Find and apply all applicable migrations.

Attributes

db_connection	
parameters	
project_parameters	
run	Load and return the AequilibraE run module with the default arguments from `parameters.yml` partially applied.
zoning	

activate()

check_file_indices() → None

    Makes results_database.sqlite and the matrices folder compatible with project database

close() → None

    Safely closes the project

deactivate()

classmethod from_path(*project_folder*)

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()

    Find and apply all applicable migrations.

    All database upgrades are applied within a single transaction.

    If skipping a specific migration is required, use the `aequilbrae.project.tools.MigrationManager` object directly. Consult it's documentation page for details. Take care when skipping migrations.

property db_connection

property parameters: dict

property project_parameters: *Parameters*

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 zoning

11.1.2 Project Components

About(project)	Provides an interface for querying and editing the **about** table of an AequilibraE project
FieldEditor(project, table_name)	Allows user to edit the project data tables
Log(project_base_path)	API entry point to the log file contents
Matrices(project)	Gateway into the matrices available/recorded in the model
Network(project)	Network class.
Zoning(network)	Access to the API resources to manipulate the 'zones' table in the project

aequilibrae.project.About

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()
```

  __init__(*project*)

Methods

__init__(project)	
add_info_field(info_field)	Adds new information field to the model
create()	Creates the 'about' table for project files that did not previously contain it
list_fields()	Returns a list of all characteristics the about table holds
write_back()	Saves the information parameters back to the project database

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()
```

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()
```

aequilibrae.project.FieldEditor

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
```

>Field descriptions are kept in the table *attributes_documentation*

>**__init__**(*project*, *table_name: str*) → None

Methods

__init__(project, table_name)	
add(field_name, description[, data_type])	Adds new field to the data table
all_fields()	Returns the list of fields available in the database
remove(field_name)	
save()	Saves any field descriptions which my have been changed to the database and update layer statistics.

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.Log

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()
```

__init__(*project_base_path: Path*)

Methods

__init__(project_base_path)	
clear()	Clears the log file.
contents()	Returns contents of log file

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.project.Matrices

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

>   Gateway into the matrices available/recorded in the model

>   **__init__**(*project*)

Methods

__init__(project)	
check_exists(name)	Checks whether a matrix with a given name exists
clear_database()	Removes records from the matrices database that do not exist in disk
delete_record(matrix_name)	Deletes a Matrix Record from the model and attempts to remove from disk
get_matrix(matrix_name)	Returns an AequilibraE matrix available in the project
get_record(matrix_name)	Returns a model Matrix Record for manipulation in memory
list()	List of all matrices available
new_record(name, file_name[, matrix])	Creates a new record for a matrix in disk, but does not save it
reload()	Discards all memory matrices in memory and loads recreate them
update_database()	Adds records to the matrices database for matrix files found on disk

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.Network

class aequilibrae.project.**Network**(*project*)

> Network class. Member of an AequilibraE Project

> **__init__**(*project*) → None

Methods

__init__(project)	
build_graphs([fields, modes, limit_to_area])	Builds graphs for all modes currently available in the model
convex_hull()	Queries the model for the convex hull of the entire network
count_centroids()	Returns the number of centroids in the model
count_links()	Returns the number of links in the model
count_nodes()	Returns the number of nodes in the model
create_from_gmns(link_file_path, node_file_path)	Creates AequilibraE model from links and nodes in GMNS format.
create_from_osm([model_area, place_name, ...])	Downloads the network from OpenStreetMap (OSM)
export_to_gmns(path)	Exports AequilibraE network to csv files in GMNS format.
extent()	Queries the extent of the network included in the model
list_modes()	Returns a list of all the modes in this model
set_time_field(time_field)	Set the time field for all graphs built in the model
skimmable_fields()	Returns a list of all fields that can be skimmed

Attributes

link_types
protected_fields
req_link_flds
req_node_flds
signal

build_graphs (*fields: list | None = None*, *modes: list | None = None*, *limit_to_area: Polygon | None = 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'])
```

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 = None*, *geometry_path: str | None = 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>

aequilibrae.project.Zoning

class aequilibrae.project.**Zoning**(*network*)

> Access to the API resources to manipulate the 'zones' table in the project

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

>>> 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')
```

__init__(*network*)

Methods

__init__(network)	
add_centroids([robust])	Adds automatic centroids to the network file.
all_zones()	Returns a dictionary with all Zone objects available in the model, using `zone_id` as key
connect_mode(mode_id[, link_types, ...])	Adds centroid connectors for the desired mode to the network file
coverage()	Returns a single polygon for the entire zoning coverage
create_zoning_layer()	Creates the 'zones' table for project files that did not previously contain it
extent()	Queries the extent of the layer included in the model
get(zone_id)	Get a zone from the model by its `zone_id`
get_closest_zone(geometry)	Returns the zone in which the given geometry is located.
new(zone_id)	Creates a new zone
refresh_geo_index()	
save()	

Attributes

data	Returns all zones data as a Pandas DataFrame
fields	Returns a FieldEditor class instance to edit the zones table fields and their metadata

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*)

    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`.

`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

11.1.3 Project Objects

Zone(dataset, zoning)	Single zone object that can be queried and manipulated in memory

aequilibrae.project.Zone

class `aequilibrae.project.`**Zone**(*dataset: dict*, *zoning*)

> Single zone object that can be queried and manipulated in memory

> **__init__**(*dataset: dict*, *zoning*)

Methods

__init__(dataset, zoning)	
add_centroid(point[, robust])	Adds a centroid to the network file
connect_mode(mode_id[, link_types, ...])	Adds centroid connectors for the desired mode to the network file
delete()	Removes the zone from the database
disconnect_mode(mode_id)	Removes centroid connectors for the desired mode from the network file
save()	Saves/Updates the zone data to the database

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

11.1.4 Project Tools

network_simplifier. *NetworkSimplifier*([project])	

aequilibrae.project.tools.network_simplifier.NetworkSimplifier

class aequilibrae.project.tools.network_simplifier.**NetworkSimplifier**(*project=None*)

> **__init__**(*project=None*) → None

Methods

__init__([project])	
collapse_links_into_nodes(links)	Collapses links into nodes, adjusting the network in the neighborhood.
rebuild_network()	Rebuilds the network elements that would have to be rebuilt after massive network simplification
simplify(graph[, max_speed_ratio])	Simplifies the network by merging links that are shorter than a given threshold

Attributes

signal

`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>`

11.2 Network Data

Modes(net)	Access to the API resources to manipulate the modes table in the network
LinkTypes(net)	Access to the API resources to manipulate the link_types table in the network.
Links(net)	Access to the API resources to manipulate the links table in the network
Nodes(net)	Access to the API resources to manipulate the nodes table in the network
Periods(net)	Access to the API resources to manipulate the periods table in the network

11.2.1 aequilibrae.project.network.Modes

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')

# 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()
```

__init__(*net*)

Methods

__init__(net)	
add(mode)	We add a mode to the project
all_modes()	Returns a dictionary with all mode objects available in the model.
delete(mode_id)	Removes the mode with *mode_id* from the project
get(mode_id)	Get a mode from the network by its *mode_id*
get_by_name(mode)	Get a mode from the network by its *mode_name*
new(mode_id)	Returns a new mode with *mode_id* that can be added to the model later

Attributes

fields	Returns a FieldEditor class instance to edit the Modes table fields and their metadata

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

11.2.2 aequilibrae.project.network.LinkTypes

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()
```

__init__(*net*)

Methods

__init__(net)	
all_types()	Returns a dictionary with all LinkType objects available in the model.
delete(link_type_id)	Removes the link_type with *link_type_id* from the project
get(link_type_id)	Get a link_type from the network by its *link_type_id*
get_by_name(link_type)	Get a link_type from the network by its *link_type* (i.e. name).
new(link_type_id)	
save()	

Attributes

fields	Returns a FieldEditor class instance to edit the Link_Types table fields and their metadata

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

11.2.3 aequilibrae.project.network.Links

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()
```

__init__(*net*)

Methods

__init__(net)	
copy_link(link_id)	Creates a copy of a link with a new id
delete(link_id)	Removes the link with link_id from the project
extent()	Queries the extent of the layer included in the model
get(link_id)	Get a link from the network by its *link_id*
new()	Creates a new link
refresh()	Refreshes all the links in memory
refresh_fields()	After adding a field one needs to refresh all the fields recognized by the software
save()	

Attributes

data	Returns all links data as a Pandas DataFrame
fields	Returns a FieldEditor class instance to edit the zones table fields and their metadata
sql	Query sql for retrieving links

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

11.2.4 aequilibrae.project.network.Nodes

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()
```

__init__(*net*)

Methods

__init__(net)	
extent()	Queries the extent of the layer included in the model
get(node_id)	Get a node from the network by its `node_id`
new_centroid(node_id)	Creates a new centroid with a given ID
refresh()	Refreshes all the nodes in memory
refresh_fields()	After adding a field one needs to refresh all the fields recognized by the software
save()	

Attributes

data	Returns all nodes data as a Pandas DataFrame
fields	Returns a FieldEditor class instance to edit the zones table fields and their metadata
lonlat	Returns all nodes lon/lat coords as a Pandas DataFrame
sql	Query sql for retrieving nodes

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

11.2.5 aequilibrae.project.network.Periods

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()
```

__init__(*net*)

Methods

__init__(net)	
extent()	Queries the extent of the layer included in the model
get(period_id)	Get a period from the network by its **period_id**
new_period(period_id, start, end[, description])	Creates a new period with a given ID
refresh()	Refreshes all the periods in memory
refresh_fields()	After adding a field one needs to refresh all the fields recognized by the software
save()	

Attributes

data	Returns all periods data as a Pandas DataFrame
default_period	
fields	Returns a FieldEditor class instance to edit the zones table fields and their metadata
sql	Query sql for retrieving periods

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 = 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

11.3 Network Items

Mode(mode_id, project)	A mode object represents a single record in the *modes* table
LinkType(data_set, project)	A link_type object represents a single record in the *link_types* table
Link(dataset, project)	A Link object represents a single record in the *links* table
Node(dataset, project)	A Node object represents a single record in the *nodes* table
Period(dataset, project)	A Period object represents a single record in the *periods* table

11.3.1 aequilibrae.project.network.Mode

class aequilibrae.project.network.**Mode**(*mode_id: str*, *project*)

A mode object represents a single record in the *modes* table

__init__(*mode_id: str*, *project*) → None

Methods

__init__(mode_id, project)	
save()	

save()

11.3.2 aequilibrae.project.network.LinkType

class aequilibrae.project.network.**LinkType**(*data_set: dict*, *project*)

A link_type object represents a single record in the *link_types* table

__init__(*data_set: dict*, *project*) → None

Methods

__init__(data_set, project)
delete()
save()

delete()

save()

11.3.3 aequilibrae.project.network.Link

class aequilibrae.project.network.**Link**(*dataset*, *project*)

    A Link object represents a single record in the *links* table

```python
>>> 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()
```

    __init__(*dataset*, *project*)

Methods

__init__(dataset, project)	
add_mode(mode)	Adds a new mode to this link
data_fields()	lists all data fields for the link, as available in the database
delete()	Deletes link from database
drop_mode(mode)	Removes a mode from this link
save([conn])	Saves link to database
set_modes(modes)	Sets the modes acceptable for this link

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

11.3.4 aequilibrae.project.network.Node

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()
```

__init__(*dataset*, *project*)

Methods

__init__(dataset, project)	
connect_mode(mode_id[, link_types, ...])	Adds centroid connectors for the desired mode to the network file
data_fields()	lists all data fields for the node, as available in the database
renumber(new_id)	Renumbers the node in the network
save()	Saves node to database

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

11.3.5 aequilibrae.project.network.Period

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()
```

__init__(*dataset*, *project*)

Methods

__init__(dataset, project)	
data_fields()	Lists all data fields for the period, as available in the database
renumber(new_id)	Renumbers the period in the network
save()	Saves period to database

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

11.4 Distribution

Ipf([project])	Iterative proportional fitting procedure
GravityCalibration([project])	Calibrate a traditional gravity model
GravityApplication([project])	Applies a synthetic gravity model.
SyntheticGravityModel()	Simple class object to represent synthetic gravity models

11.4.1 aequilibrae.distribution.Ipf

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",
...             "nan_as_zero": False}

>>> fratar = Ipf(**ipf_args)
>>> fratar.fit()

# We can get back to our OMX matrix in the end
>>> fratar.output.export(os.path.join(my_folder_path, "to_omx_output.omx"))
```

__init__(*project=None*, ***kwargs*)

Instantiates the IPF problem

Arguments

matrix (`AequilibraeMatrix`): Seed Matrix

vectors (`pd.DataFrame`): Dataframe with the vectors to be used for the IPF

row_field (`str`): Field name that contains the data for the row totals

column_field (`str`): Field name that contains the data for the column totals

parameters (`str`, *Optional*): Convergence parameters. Defaults to those in the parameter file

nan_as_zero (`bool`, *Optional*): If Nan values should be treated as zero. Defaults to `True`

Results

output (`AequilibraeMatrix`): Result Matrix

report (`list`): Iteration and convergence report

error (`str`): Error description

Methods

__init__([project])	Instantiates the IPF problem
fit()	Runs the IPF instance problem to adjust the matrix
save_to_project(name, file_name[, project])	Saves the matrix output to the project file

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

11.4.2 aequilibrae.distribution.GravityCalibration

class aequilibrae.distribution.**GravityCalibration**(*project=None*, ***kwargs*)

Calibrate a traditional gravity model

Available deterrence function forms are: 'EXPO', 'POWER' or 'GAMMA'.

```python
>>> 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(os.path.join(project_path, 'dist_expo_model.mod'))
```

__init__(*project=None*, ***kwargs*)

> Instantiates the Gravity calibration problem
>
> > **Arguments**
> >
> > > **matrix** (`AequilibraeMatrix`): Seed/base trip matrix
> > >
> > > **impedance** (`AequilibraeMatrix`): Impedance matrix to be used
> > >
> > > **function** (`str`): Function name to be calibrated. "EXPO" or "POWER"
> > >
> > > **project** (`Project`, *Optional*): The Project to connect to. By default, uses the currently active project
> > >
> > > **parameters** (`str`, *Optional*): Convergence parameters. Defaults to those in the parameter file
> > >
> > > **nan_as_zero** (`bool`, *Optional*): If Nan values should be treated as zero. Defaults to `True`
> >
> > **Results**
> >
> > > **model** (*SyntheticGravityModel*): Calibrated model
> > >
> > > **report** (`list`): Iteration and convergence report
> > >
> > > **error** (`str`): Error description

Methods

__init__([project])	Instantiates the Gravity calibration problem
calibrate()	Calibrate the model

calibrate()

> Calibrate the model
>
> Resulting model is in *output* class member

11.4.3 aequilibrae.distribution.GravityApplication

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"
```

```python
# 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.loc[:, "attractions"] *= vectors["productions"].sum() / vectors[
→"attractions"].sum()

# Create the problem object
>>> args = {"impedance": matrix,
...         "vectors": vectors,
...         "row_field": "productions",
...         "model": model,
...         "column_field": "attractions",
...         "output": os.path.join(project_path, 'matrices/gravity_matrix.aem'),
...         "nan_as_zero":True
...         }
>>> gravity = GravityApplication(**args)

# Solve and save the outputs
>>> gravity.apply()
>>> gravity.output.export(os.path.join(project_path, 'matrices/gravity_omx.omx'))
```

__init__(*project=None*, ***kwargs*)

    Instantiates the IPF problem

        Arguments

            model (*SyntheticGravityModel*): Synthetic gravity model to apply

            impedance (`AequilibraeMatrix`): Impedance matrix to be used

            vectors (`pd.DataFrame`): Dataframe with data for row and column totals

            row_field (`str`): Field name that contains the data for the row totals

            column_field (`str`): Field name that contains the data for the column totals

            project (`Project`, *Optional*): The Project to connect to. By default, uses the currently active

project

core_name (`str`, *Optional*): Name for the output matrix core. Defaults to "gravity"

parameters (`str`, *Optional*): Convergence parameters. Defaults to those in the parameter file

nan_as_zero (`bool`, *Optional*): If NaN values should be treated as zero. Defaults to `True`

Results
output (`AequilibraeMatrix`): Result Matrix

report (`list`): Iteration and convergence report

error (`str`): Error description

Methods

__init__([project])	Instantiates the IPF problem
apply()	Runs the Gravity Application instance as instantiated
save_to_project(name, file_name[, project])	Saves the matrix output to the project file

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

11.4.4 aequilibrae.distribution.SyntheticGravityModel

class aequilibrae.distribution.**SyntheticGravityModel**

Simple class object to represent synthetic gravity models

__init__()

Methods

__init__()	
load(file_name)	Loads model from disk.
save(file_name)	Saves model to disk in yaml format.

load(*file_name*)

Loads model from disk. Extension is *.mod

save(*file_name*)

Saves model to disk in yaml format. Extension is *.mod

11.5 Matrix

AequilibraeMatrix()	Matrix class

11.5.1 aequilibrae.matrix.AequilibraeMatrix

class aequilibrae.matrix.**AequilibraeMatrix**

Matrix class

__init__()

Creates a memory instance for a matrix, that can be used to load an existing matrix or to create an empty one

Methods

__init__()	Creates a memory instance for a matrix, that can be used to load an existing matrix or to create an empty one
close()	Removes matrix from memory and flushes all data to disk, or closes the OMX file if that is the case
columns()	Returns column vector for the matrix in the computational view
computational_view([core_list])	Creates a memory view for a list of matrices that is compatible with Cython memory buffers
copy([output_name, cores, names, compress, ...])	Copies a list of cores (or all cores) from one matrix file to another one
create_empty([file_name, zones, ...])	Creates an empty matrix in the AequilibraE format
create_from_omx(omx_path[, file_path, ...])	Creates an AequilibraeMatrix from an original OpenMatrix
create_from_trip_list(path_to_file, ...)	Creates an AequilibraeMatrix from a trip list csv file The output is saved in the same folder as the trip list file
export(output_name[, cores])	Exports the matrix to other formats, rather than AEM.
get_matrix(core[, copy])	Returns the data for a matrix core
is_omx()	Returns `True` if matrix data source is OMX, `False` otherwise
load(file_path)	Loads matrix from disk.
nan_to_num()	Converts all `NaN` values in all cores in the computational view to zeros
random_name()	Returns a random name for a matrix with root in the temp directory of the user
rows()	Returns row vector for the matrix in the computational view
save([names, file_name])	Saves matrix data back to file.
setDescription(matrix_description)	Sets description for the matrix
setName(matrix_name)	Sets the name for the matrix itself.
set_index(index_to_set)	Sets the standard index to be the one the user wants to have be the one being used in all operations during run time.

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(os.path.join(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 ])
```

computational_view(*core_list: List[str] | None = 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(file_name=os.path.join(my_folder_path, 'my_matrix.aem'),
...                  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 = None, cores: List[str] | None = None, names: List[str] | None = None, compress: bool | None = 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`

```python
>>> 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(file_name=os.path.join(my_folder_path, 'my_matrix.aem'),
...                  zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.copy(os.path.join(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(os.path.join(my_folder_path, 'copy_of_my_matrix.aem'))
>>> mat2.cores
2
```

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

> Creates an empty matrix in the AequilibraE format
>
> > **Arguments**
> >
> > **file_name** (`str`): 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`

```python
>>> 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(file_name=os.path.join(my_folder_path, 'my_matrix.aem'),
...                  zones=zones_in_the_model,
...                  matrix_names=names_list,
...                  memory_only=False)
>>> mat.num_indices
1
>>> mat.zones
3317
```

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

    Creates an AequilibraeMatrix from an original OpenMatrix

    Arguments

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

        file_path (`str`, *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: str*, *cores: List[str] | None = 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 (`str`): 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(file_name=os.path.join(my_folder_path, 'my_matrix.aem'),
...                  zones=zones_in_the_model,
...                  matrix_names=names_list)

>>> mat.export(os.path.join(my_folder_path, 'my_new_path.aem'), ['Car trips',
↪'bike trips'])

>>> mat2 = AequilibraeMatrix()
>>> mat2.load(os.path.join(my_folder_path, 'my_new_path.aem'))
>>> mat2.cores
2
```

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: str*)

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(os.path.join(project_path, 'matrices/skims.omx'))
>>> mat.computational_view()
>>> mat.names
['distance_blended', 'time_final']
```

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))
```

(continues on next page)

```
>>> nan_matrix[:] = np.nan

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

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name=os.path.join(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.]])
```

static random_name() → str

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

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

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

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(os.path.join(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])
```

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=os.path.join(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(os.path.join(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=os.path.join(my_folder_path, 'my_matrix.aem'),
...                   zones=zones_in_the_model,
...                   memory_only=False)
>>> mat.setName('This is my example')
>>> mat.save()
>>> mat.close()

>>> mat = AequilibraeMatrix()
>>> mat.load(os.path.join(my_folder_path, 'my_matrix.aem'))
>>> mat.name.decode('utf-8')
'This is my example'
```

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(file_name=os.path.join(my_folder_path, 'my_matrix.aem'),
...                  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'
```

11.6 Paths

11.6.1 Skimming

NetworkSkimming(graph[, origins, project])	

aequilibrae.paths.NetworkSkimming

class aequilibrae.paths.**NetworkSkimming**(*graph*, *origins=None*, *project=None*)

    __init__(*graph*, *origins=None*, *project=None*)

Methods

__init__(graph[, origins, project])	
doWork()	
execute()	Runs the skimming process as specified in the graph
save_to_project(name[, format, project])	Saves skim results to the project folder and creates record in the database
set_cores(cores)	Sets number of cores (threads) to be used in computation

Attributes

signal

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>

```
>>> from aequilibrae.paths.network_skimming import NetworkSkimming
```

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

```
>>> network = project.network
>>> network.build_graphs()
```

```
>>> graph = network.graphs['c']
>>> graph.set_graph(cost_field="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(os.path.join(project_path, 'matrices/skimming_result.omx'))

Or specify the AequilibraE's matrix file format >>> skm.save_to_project(os.path.join(project_path, 'matrices/skimming_result.aem'), 'aem')

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

11.6.2 Graph

Graph(*args, **kwargs)

TransitGraph([config, od_node_mapping])

aequilibrae.paths.Graph

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

> **__init__**(**args*, ***kwargs*)

> #### Methods

> | | |
> |---|---|
> | *__init__*(*args, **kwargs) | |
> | *available_skims*() | Returns graph fields that are available to be set as skims |
> | *compute_path*(origin, destination[, ...]) | Returns the results from path computation result holder. |
> | *compute_skims*([cores]) | Returns the results from network skimming result holder. |
> | *create_compressed_link_network_mapping*() | Create three arrays providing a mapping of compressed ID to link ID. |
> | *default_types*(tp) | Returns the default integer and float types used for computation |
> | *exclude_links*(links) | Excludes a list of links from a graph by setting their B node equal to their A node |
> | *load_from_disk*(filename) | Loads graph from disk |
> | *prepare_graph*([centroids, remove_dead_ends]) | Prepares the graph for a computation for a certain set of centroids |
> | *save_compressed_correspondence*(path, ...) | Save graph and nodes_to_indices to disk |
> | *save_to_disk*(filename) | Saves graph to disk |
> | *set_blocked_centroid_flows*(block_centroid_flc | Chooses whether we want to block paths to go through centroids or not. |
> | *set_graph*(cost_field) | Sets the field to be used for path computation |
> | *set_skimming*(skim_fields) | Sets the list of skims to be computed |

> **available_skims**() → List[str]
>
> > Returns graph fields that are available to be set as skims
> >
> > > **Returns**
> > > > **list** (str): 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()
```

 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`): Array with centroid IDs. Mandatory type Int64, unique and positive
> > > **remove_dead_ends** (`bool`): Whether or not to remove dead ends from the graph. (*Optional*, default is "True").

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

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

aequilibrae.paths.TransitGraph

class aequilibrae.paths.**TransitGraph** (*config: dict | None = None*, *od_node_mapping: DataFrame | None = None*, **args*, ***kwargs*)

> **__init__** (*config: dict | None = None*, *od_node_mapping: DataFrame | None = None*, **args*, ***kwargs*)

Methods

__init__([config, od_node_mapping])	
available_skims()	Returns graph fields that are available to be set as skims
compute_path(origin, destination[, ...])	Returns the results from path computation result holder.
compute_skims([cores])	Returns the results from network skimming result holder.
create_compressed_link_network_mapping()	Create three arrays providing a mapping of compressed ID to link ID.
default_types(tp)	Returns the default integer and float types used for computation
exclude_links(links)	Excludes a list of links from a graph by setting their B node equal to their A node
load_from_disk(filename)	Loads graph from disk
prepare_graph([centroids, remove_dead_ends])	Prepares the graph for a computation for a certain set of centroids
save_compressed_correspondence(path, ...)	Save graph and nodes_to_indices to disk
save_to_disk(filename)	Saves graph to disk
set_blocked_centroid_flows(block_centroid_flc	Chooses whether we want to block paths to go through centroids or not.
set_graph(cost_field)	Sets the field to be used for path computation
set_skimming(skim_fields)	Sets the list of skims to be computed

Attributes

config

available_skims() → List[str]

Returns graph fields that are available to be set as skims

> **Returns**
> **list** (`str`): 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()
```

> **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`): Array with centroid IDs. Mandatory type Int64, unique and positive

> **remove_dead_ends** (`bool`): Whether or not to remove dead ends from the graph. (*Optional*, default is "True").

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

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

11.6.3 Traffic assignment

TrafficClass(name, graph, matrix)	Traffic class for equilibrium traffic assignment
TransitClass(name, graph, matrix)	
VDF()	Volume-Delay function
TrafficAssignment([project])	Traffic assignment class.
TransitAssignment(*args[, project])	
AssignmentResults()	Assignment result holder for a single *TrafficClass* with multiple user classes
TransitAssignmentResults()	Assignment result holder for a single `Transit`
SkimResults()	Network skimming result holder.
PathResults()	Path computation result holder

aequilibrae.paths.TrafficClass

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(True)

>>> proj_matrices = project.matrices

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

>>> tc = TrafficClass("car", graph, demand)
>>> tc.set_pce(1.3)
```

__init__(*name: str*, *graph:* Graph, *matrix:* AequilibraeMatrix) → None

   Instantiates the class

   Arguments

   name (`str`): UNIQUE class name.

   graph (`Graph`): Class/mode-specific graph

   matrix (`AequilibraeMatrix`): Class/mode-specific matrix. Supports multiple user classes

Methods

__init__(name, graph, matrix)	Instantiates the class
set_fixed_cost(field_name[, multiplier])	Sets value of time
set_pce(pce)	Sets Passenger Car equivalent
set_select_links(links)	Set the selected links.
set_vot(value_of_time)	Sets value of time
skim_congested([skim_fields])	Skims the congested network.

Attributes

info

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

aequilibrae.paths.TransitClass

class aequilibrae.paths.**TransitClass**(*name: str*, *graph:* TransitGraph, *matrix:* AequilibraeMatrix)

> **__init__**(*name: str*, *graph:* TransitGraph, *matrix:* AequilibraeMatrix)
> > Instantiates the class
> >
> > > **Arguments**
> > > > **name** (`str`): UNIQUE class name.
> > > >
> > > > **graph** (*Graph*): Class/mode-specific graph
> > > >
> > > > **matrix** (`AequilibraeMatrix`): Class/mode-specific matrix. Supports multiple user classes

Methods

__init__(name, graph, matrix)	Instantiates the class
set_demand_matrix_core(core)	Set the matrix core to use for demand.

Attributes

info

set_demand_matrix_core(*core: str*)
> Set the matrix core to use for demand.
>
> > **Arguments**
> > > **core** (`str`):

property info: dict

aequilibrae.paths.VDF

class aequilibrae.paths.**VDF**

    Volume-Delay function

```
>>> from aequilibrae.paths import VDF

>>> vdf = VDF()
>>> vdf.functions_available()
['bpr', 'bpr2', 'conical', 'inrets']
```

    __init__()

Methods

__init__()	
functions_available()	returns a list of all functions available

    functions_available() → list

        returns a list of all functions available

aequilibrae.paths.TrafficAssignment

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()

# 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
```

__init__(*project=None*) → None

Methods

__init__([project])	
add_class(traffic_class)	Adds a traffic class to the assignment
add_preload(preload[, name])	Given a dataframe of 'link_id', 'direction' and 'preload', merge into current preloads dataframe.
algorithms_available()	Returns all algorithms available for use
execute([log_specification])	Processes assignment
get_skim_results()	Prepares the assignment skim results for all classes
info()	Returns information for the traffic assignment procedure
log_specification()	
report()	Returns the assignment convergence report
results()	Prepares the assignment results as a Pandas DataFrame
save_results(table_name[, keep_zero_flows, ...])	Saves the assignment results to results_database.sqlite
save_select_link_flows(table_name[, project])	Saves the select link link flows for all classes into the results database.
save_select_link_matrices(matrix_name[, project])	Saves the Select Link matrices for each TrafficClass in the current TrafficAssignment class into OMX format.
save_select_link_results(name)	Saves both the Select Link matrices and flow results at the same time, using the same name.
save_skims(matrix_name[, which_ones, ...])	Saves the skims (if any) to the skim folder and registers in the matrix list
select_link_flows()	Returns a dataframe of the select link flows for each class
set_algorithm(algorithm)	Chooses the assignment algorithm.
set_capacity_field(capacity_field)	Sets the graph field that contains link capacity for the assignment period -> e.g. 'capacity1h'.
set_classes(classes)	Sets Traffic classes to be assigned
set_cores(cores)	Allows one to set the number of cores to be used AFTER traffic classes have been added
set_path_file_format(file_format)	Specify path saving format.
set_save_path_files(save_it)	Turn path saving on or off.
set_time_field(time_field)	Sets the graph field that contains free flow travel time -> e.g. 'fftime'.
set_vdf(vdf_function)	Sets the Volume-delay function to be used
set_vdf_parameters(par)	Sets the parameters for the Volume-delay function.
skim_congested([skim_fields, return_matrices])	Skims the congested network.

Attributes

all_algorithms	
bpr_parameters	

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*) → 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'}
>
> > **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']

aequilibrae.paths.TransitAssignment

class aequilibrae.paths.**TransitAssignment**(**args*, *project=None*, ***kwargs*)

    __init__(**args*, *project=None*, ***kwargs*)

Methods

__init__(*args[, project])	
add_class(transport_class)	Adds a Transport class to the assignment
algorithms_available()	Returns all algorithms available for use
execute([log_specification])	Processes assignment
get_skim_results()	Prepares the assignment skim results for all classes
info()	Returns information for the transit assignment procedure
log_specification()	
report()	Returns the assignment convergence report
results()	Prepares the assignment results as a Pandas DataFrame
save_results(table_name[, keep_zero_flows, ...])	Saves the assignment results to results_database.sqlite
set_algorithm(algorithm)	Chooses the assignment algorithm.
set_classes(classes)	Sets Transport classes to be assigned
set_cores(cores)	Allows one to set the number of cores to be used AFTER transit classes have been added
set_frequency_field(frequency_field)	Sets the graph field that contains the frequency -> e.g. 'freq'.
set_skimming_fields([skimming_fields])	Sets the skimming fields for the transit assignment.
set_time_field(time_field)	Sets the graph field that contains free flow travel time -> e.g. 'trav_time'.

Attributes

all_algorithms

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*) → 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

all_algorithms = ['optimal-strategies', 'os']

aequilibrae.paths.AssignmentResults

class aequilibrae.paths.**AssignmentResults**

> Assignment result holder for a single [`TrafficClass`](#) with multiple user classes
>
> **__init__**()

Methods

[*__init__*]()	
[*get_graph_to_network_mapping*]()	
[*get_load_results*]()	Translates the assignment results from the graph format into the network format
[*get_sl_results*]()	
[*prepare*](graph, matrix)	Prepares the object with dimensions corresponding to the assignment matrix and graph objects
[*reset*]()	Resets object to prepared and pre-computation state
[*set_cores*](cores)	Sets number of cores (threads) to be used in computation
[*total_flows*]()	Totals all link flows for this class into a single link load

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

aequilibrae.paths.TransitAssignmentResults

class aequilibrae.paths.**TransitAssignmentResults**

> Assignment result holder for a single Transit
>
> **__init__**()

Methods

__init__()	
get_load_results()	Translates the assignment results from the graph format into the network format
prepare(graph, matrix)	Prepares the object with dimensions corresponding to the assignment matrix and graph objects
reset()	Resets object to prepared and pre-computation state
set_cores(cores)	Sets number of cores (threads) to be used in computation

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.SkimResults

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(os.path.join(project_path, "skim_matrices.aem"))
> ```
>
> **__init__**()

Methods

__init__()	
prepare(graph)	Prepares the object with dimensions corresponding to the graph objects

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.PathResults

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
>>> res.reset()
```

__init__() → None

Methods

__init__()	
compute_path(origin, destination[, ...])	Computes the path between two nodes in the network.
get_heuristics()	Return the availiable heuristics.
prepare(graph)	Prepares the object with dimensions corresponding to the graph object
reset()	Resets object to prepared and pre-computation state
set_heuristic(heuristic)	Set the heuristics to be used in A*.
update_trace(destination)	Updates the path's nodes, links, skims and mileposts

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 availiable 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

11.6.4 Route choice

HyperpathGenerating(edges[, tail, head, ...])	A class for hyperpath generation.
OptimalStrategies(assig_spec)	
RouteChoice(graph[, project])	
SubAreaAnalysis(graph, subarea, demand[, ...])	

aequilibrae.paths.HyperpathGenerating

class aequilibrae.paths.**HyperpathGenerating**(*edges*, *tail='tail'*, *head='head'*, *trav_time='trav_time'*, *freq='freq'*, *check_edges=False*, *skim_cols=None*, ***, *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`): The column name for the tail of the edge (*Optional*, default is "tail").
>>
>> **head** (`str`): The column name for the head of the edge (*Optional*, default is "head").
>>
>> **trav_time** (`str`): The column name for the travel time of the edge (*Optional*, default is "trav_time").
>>
>> **freq** (`str`): The column name for the frequency of the edge (*Optional*, default is "freq").
>>
>> **check_edges** (`bool`): If True, check the validity of the edges (*Optional*, default is False).

__init__(*edges*, *tail='tail'*, *head='head'*, *trav_time='trav_time'*, *freq='freq'*, *check_edges=False*, *skim_cols=None*, ***, *o_vert_ids=array([], dtype=int64)*, *d_vert_ids=array([], dtype=int64)*, *nodes_to_indices*)

Methods

__init__(edges[, tail, head, trav_time, ...])	
assign(origin_column, destination_column, ...)	Assigns demand to the edges of the graph.
check_skim_cols(skim_cols)	
compute_skim_cols(skim_cols, edges, trav_time)	
info()	
run(origin, destination, volume)	
save_results(table_name[, keep_zero_flows, ...])	Saves the assignment results to results_database.sqlite

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`): The column for the origin vertices (*Optional*, default is
    "orig_vert_idx").

destination_column (`np.ndarray`): The column or the destination vertices (*Optional*,
default is
    "dest_vert_idx").

demand_column (`np.ndarray`): The column for the demand values (*Optional*, default is
"demand").

check_demand (`bool`): If True, check the validity of the demand data (*Optional*, default
is
    `False`).

threads (`int`):The number of threads to use for computation (*Optional*, 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

aequilibrae.paths.OptimalStrategies

class aequilibrae.paths.**OptimalStrategies**(*assig_spec*)

__init__(*assig_spec*)

Methods

[*__init__*](assig_spec)	
[*execute*]()	

execute()

aequilibrae.paths.RouteChoice

class aequilibrae.paths.**RouteChoice**(*graph:* Graph, *project=None*)

    __init__(*graph:* Graph, *project=None*)

Methods

__init__(graph[, project])	
add_demand(demand[, fill])	Add demand DataFrame or matrix for the assignment.
execute([perform_assignment])	Generate route choice sets between the previously supplied nodes, potentially performing an assignment.
execute_from_pandas(df[, recompute_psl])	Perform an assignment using route sets from a Pandas DataFrame.
execute_from_path_files(path_files[, ...])	Perform an assignment from an existing set of path-files.
execute_single(origin, destination[, demand])	Generate route choice sets between origin and destination, potentially performing an assignment.
get_load_results()	Translates the link loading results from the graph format into the network format.
get_results()	Returns the results of the route choice procedure
get_select_link_loading_results()	Get the select link loading results.
get_select_link_od_matrix_results()	Get the select link OD matrix results as a sparse matrix.
info()	Returns information for the transit assignment procedure
log_specification()	
prepare([nodes])	Prepare OD pairs for batch computation.
save_link_flows(table_name[, project])	Saves the link link flows for all classes into the results database.
save_path_files([where])	Save path-files to the directory specific.
save_select_link_flows(table_name[, project])	Saves the select link link flows for all classes into the results database.
set_choice_set_generation([algorithm])	Chooses the assignment algorithm and set its parameters.
set_cores(cores)	Allows one to set the number of cores to be used
set_save_routes([where])	Set save path for route choice results.
set_select_links(links[, link_loading])	Set the selected links.

Attributes

all_algorithms
default_parameters
demand_index_names

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``s or ``float64``s.
>
> **fill** (`float`): Value to fill any ``NaN``s 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 direc-
>>> tion.

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*)

> 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.

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 = 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 runaway 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*) → None

    Set save path for route choice results. Provide `None` to disable.

    Arguments

        save_it (`bool`): Boolean to indicate whether routes should be saved

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']

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']
```

aequilibrae.paths.SubAreaAnalysis

class aequilibrae.paths.**SubAreaAnalysis**(*graph:* Graph, *subarea: GeoDataFrame*, *demand: DataFrame |* AequilibraeMatrix, *project=None*)

__init__(*graph:* Graph, *subarea: GeoDataFrame*, *demand: DataFrame |* AequilibraeMatrix, *project=None*)

Construct a sub-area matrix from a provided sub-area GeoDataFrame using route choice.

This class aims to provide a semi-automated method for constructing the sub-area matrix. The user should provide the Graph object, demand matrix, and a GeoDataFrame whose geometry union represents the desired sub-area. Perform a route choice assignment, then call the `post_process` method to obtain a sub-area matrix.

Check how to run sub-area analysis *here*.

> **Arguments**
>> **graph** (`Graph`): AequilibraE graph object to use
>>
>> **subarea** (`gpd.GeoDataFrame`): A GeoPandas GeoDataFrame whose geometry union represents the sub-area.
>>
>> **demand** (`Union[pandas.DataFrame, AequilibraeMatrix]`): The demand matrix to provide to the route choice assignment.

Methods

__init__(graph, subarea, demand[, project])	Construct a sub-area matrix from a provided sub-area GeoDataFrame using route choice.
post_process([demand_cols, keep_original_ods])	Apply the necessary post processing to the route choice assignment select link results.

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** (*Optional*: `[list[str]]`): **If provided, only construct the**
>> sub-area matrix for these demand matrices.
>>
>> **keep_original_ods** (*Optional*: `bool`): **If provided, the original origin and destination**
>> **IDs for**
>> the demand will be kept. This will create a significantly larger demand matrix but is more flexible.
>
> **Returns**
>> **sub_area_demand** (`pd.DataFrame`): A DataFrame representing the sub-area demand matrix.

11.7 Transit

Transit(project)	
TransitGraphBuilder(public_transport_conn[, ...])	Graph builder for the transit assignment Spiess & Florian algorithm.
lib_gtfs.GTFSRouteSystemBuilder(network, ...)	

11.7.1 aequilibrae.transit.Transit

class aequilibrae.transit.**Transit**(*project*)

    __init__(*project*)

        Arguments
            project (Project, *Optional*): The Project to connect to. By default, uses the currently active project

Methods

__init__(project)	
build_pt_preload(start, end[, inclusion_cond])	Builds a preload vector for the transit network over the specified time period
create_graph(**kwargs)	Create a transit graph from an existing GTFS import.
create_transit_database()	Creates the public transport database
get_table(table_name)	
load([period_ids])	Load the previously saved transit graphs from the 'public_transport.sqlite' database.
new_gtfs_builder(agency, file_path[, day, ...])	Returns a GTFSRouteSystemBuilder object compatible with the project
remove_graphs(period_ids[, unload])	Remove the previously saved transit graphs from the 'public_transport.sqlite' database.
save_graphs([period_ids, force])	Save the previously build transit graphs to the 'public_transport.sqlite' database.

Attributes

default_capacities
default_pces
graphs
transit
pt_con

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)
```

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 = 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 = 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>

11.7.2 aequilibrae.transit.TransitGraphBuilder

class aequilibrae.transit.**TransitGraphBuilder** (*public_transport_conn: Connection*, *period_id: int = 1*, *time_margin: int = 0*, *projected_crs: str = 'EPSG:3857'*, *num_threads: int = -1*, *seed: int | None = None*, *geometry_noise: bool | None = None*, *noise_coef: float | None = 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.

__init__ (*public_transport_conn: Connection, period_id: int = 1, time_margin: int = 0, projected_crs: str = 'EPSG:3857', num_threads: int = -1, seed: int | None = None, geometry_noise: bool | None = None, noise_coef: float | None = 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*)

Methods

__init__(public_transport_conn[, period_id, ...])	
add_zones(zones[, from_crs])	Add zones as ODs.
*convert_demand_matrix_from_zone_to_node_*	Convert a sparse demand matrix from `zone_id`'s to the corresponding `node_id`'s.
create_additional_db_fields([conn])	Create the additional required entries in the tables.
create_graph()	Create the SF transit graph (vertices and edges).
create_line_geometry([method, graph])	Create the LineString for each edge.
create_od_node_mapping()	Build a dataframe mapping the centroid node ids with to transport assignment zone ids.
from_db(public_transport_conn, period_id, ...)	Create a SF graph instance from an existing database save.
remove(pt_conn, period_id)	
remove_config(conn, period_id)	Remove a transit graph configuration from the project database specified by it's 'period_id'.
remove_edges(pt_conn, period_id)	Remove a transit graph's edges from the public transport database specified by it's 'period_id'.
remove_vertices(pt_conn, period_id)	Remove a transit graph's vertices from the public transport database specified by it's 'period_id'.
save([robust, pt_conn, project_conn])	Save the current graph to the public transport database.
save_config([conn])	
save_edges([recreate_line_geometry, conn])	Save the contents of self.edges to the public transport database.
save_vertices([robust, conn])	Write the vertices DataFrame to the public transport database.
to_transit_graph()	Create an AequilibraE `TransitGraph` object from an SF graph builder.

Attributes

config

---

add_zones (*zones*, *from_crs: str | None = 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 = 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.

classmethod from_db (*public_transport_conn*, *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
    public_transport_conn (`sqlite3.Connection`): Connection to the 'public_transport.sqlite' database.

classmethod remove(*pt_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.

save(*robust=True*, *pt_conn: Connection | None = None*, *project_conn: Connection | None = 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 = None*)

save_edges(*recreate_line_geometry=False*, *conn: Connection | None = 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 = 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

11.7.3 aequilibrae.transit.lib_gtfs.GTFSRouteSystemBuilder

class aequilibrae.transit.lib_gtfs.**GTFSRouteSystemBuilder**(*network*, *agency_identifier*, *file_path*,
day='', *description=''*, *capacities=None*,
pces=None)

>   **__init__**(*network*, *agency_identifier*, *file_path*, *day=''*, *description=''*, *capacities=None*, *pces=None*)
>
>   Instantiates a transit class for the network
>
>   >   **Arguments**
>   >   **local network** (`Network`): Supply model to which this GTFS will be imported
>   >
>   >   **agency_identifier** (`str`): ID 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. 'CTA19 fixed by John after coffee')

Methods

__init__(network, agency_identifier, file_path)	Instantiates a transit class for the network
builds_link_graphs_with_broken_stops()	Build the graph for links for a certain mode while splitting the closest links at stops' projection
dates_available()	Returns a list of all dates available for this feed.
doWork()	Alias for execute_import
execute_import()	
load_date(service_date)	Loads the transit services available for *service_date*
map_match([route_types])	Performs map-matching for all routes of one or more types.
save_to_disk()	Saves all transit elements built in memory to disk
set_agency_identifier(agency_id)	Adds agency ID to this GTFS for use on import.
set_allow_map_match([allow])	Changes behavior for finding transit-link shapes.
set_capacities(capacities)	Sets default capacities for modes/vehicles.
set_date(service_date)	Sets the date for import without doing any of data processing, which is left for the importer
set_description(description)	Adds description to be added to the imported layers metadata
set_feed(feed_path)	Sets GTFS feed source to be used.
set_maximum_speeds(max_speeds)	Sets the maximum speeds to be enforced at segments.
set_pces(pces)	Sets default passenger car equivalent (PCE) factor for each GTFS mode.

Attributes

signal	Container for GTFS feeds providing data retrieval for the importer

builds_link_graphs_with_broken_stops()

>   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

11.8 Utils

create_delaunay_network. *DelaunayAnalysis*(project)	
create_example	

11.8.1 aequilibrae.utils.create_delaunay_network.DelaunayAnalysis

class aequilibrae.utils.create_delaunay_network.**DelaunayAnalysis**(*project*)

> **__init__**(*project*)
> > Start a Delaunay analysis

> > > **Arguments**
> > > **project** (`Project`): The Project to connect to

Methods

__init__(project)	Start a Delaunay analysis
assign_matrix(matrix, result_name)	
create_network([source, overwrite])	Creates a delaunay network based on the existing model

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`

11.8.2 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]

11.9 Parameters

Parameters([project])	Global parameters module.

11.9.1 aequilibrae.Parameters

class aequilibrae.**Parameters**(*project=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 import Parameters

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

>>> p = Parameters(project)

>>> 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()
```

__init__(*project=None*)

> Loads parameters from file. The place is always the same. The root of the package

Methods

__init__([project])	Loads parameters from file.
restore_default()	Restores parameters to generic default
write_back()	Writes the parameters back to file

Attributes

file_default

restore_default()

> Restores parameters to generic default

write_back()

> Writes the parameters back to file

file_default: str =
'/home/runner/work/aequilibrae/aequilibrae/aequilibrae/parameters.yml'

PYTHON MODULE INDEX

a

aequilibrae.utils.create_example, 291

F

G

H

I