

---

# **AequilibraE Documentation**

**Pedro Camargo**

**Aug 19, 2024**



## CONTENTS

<b>1</b>	<b>Examples</b>	<b>3</b>
<b>2</b>	<b>Modeling with AequilibraE</b>	<b>85</b>
<b>3</b>	<b>API Reference</b>	<b>177</b>
<b>A</b>	<b>Installation</b>	<b>251</b>
<b>B</b>	<b>IPF Performance</b>	<b>255</b>
<b>C</b>	<b>Traffic Assignment</b>	<b>261</b>
<b>D</b>	<b>Importing from Open Street Maps</b>	<b>281</b>
<b>E</b>	<b>Importing from files in GMNS format</b>	<b>283</b>
<b>F</b>	<b>Exporting AequilibraE model to GMNS format</b>	<b>285</b>
	<b>Python Module Index</b>	<b>287</b>
	<b>Index</b>	<b>289</b>





AequilibraE is the first comprehensive Python package for transportation modeling, and it aims to provide all the resources not easily available from other open-source packages in the Python (NumPy, really) ecosystem.

**Download documentation:** [HTML](#) | [PDF](#)

**Previous versions:** documentation for AequilibraE's versions before 0.9.0 are available [here](#).

**Useful links:** [Installation](#) | [validation](#) | [developing\\_aequilibrae](#) | [support](#) | [history\\_of\\_aequilibrae](#)

**Examples** A series of examples on how to use AequilibraE, from building a model from scratch to editing an existing, performing trip distribution or traffic assignment to analyzing results.

[Examples](#) **Modeling with AequilibraE** An in-depth guide to modeling with AequilibraE, including the concepts that guide its development and user-experience.

[Modeling with AequilibraE](#) **API References** Reference guide to AequilibraE's API.

[API Reference](#) Not a programmer? [Take me to the GUI!](#)

<https://www.aequilibrae.com/qgis/latest/>



## EXAMPLES

A series of different examples using AequilibraE's main features

## 1.1 Creating Models

## 1.2 Editing networks

## 1.3 Trip Distribution

## 1.4 Visualization

Examples in this session allows the user to plot some data visualization.

## 1.5 AequilibraE without a Model

## 1.6 Assignment Workflows

## 1.7 Other Applications

### 1.7.1 Creating Models

#### 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 Folium to visualize the network.

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

We create an empty project on an arbitrary folder

```
fldr = join(gettempdir(), uuid4().hex)
project = Project()
project.new(fldr)
```

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

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

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

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

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

We grab all the links data as a Pandas DataFrame so we can process it easier

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

We create a Folium layer

```
network_links = folium.FeatureGroup("links")
```

We do some Python magic to transform this dataset into the format required by Folium. We are only getting link\_id and link\_type into the map, but we could get other pieces of info as well.

```
for i, row in links.iterrows():
    points = row.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "
    ↪").split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
    # we need to take from x/y to lat/long
    points = [[x[1], x[0]] for x in eval(points)]

    line = folium.vector_layers.PolyLine(
        points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.link_type}",
    ↪color="blue", weight=10
    ).add_to(network_links)
```

We get the center of the region we are working with some SQL magic

```
curr = project.conn.cursor()
curr.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
long, lat = curr.fetchone()
```

```
map_osm = folium.Map(location=[lat, long], zoom_start=14)
network_links.add_to(map_osm)
folium.LayerControl().add_to(map_osm)
map_osm
```

```
project.close()
```

## Creating a zone system based on Hex Bins

In this example, we show how to create hex bin zones covering an arbitrary area.

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.

We also add centroid connectors to our network to make it a pretty complete example

```
# 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
```

We create an empty project on an arbitrary folder

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

# We can print the list of examples that ship with AequilibraE
print(list_examples())

# Let's use the Nauru example project for display
project = create_example(fldr, "nauru")
```

```
['sioux_falls', 'nauru', 'coquimbo']
```

We said we wanted 100 zones

```
zones = 100
```

## Hex Bins using Spatialite

Spatialite requires a few things to compute hex bins. One of them is the area you want to cover.

```
network = project.network
```

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

```
geo = network.convex_hull()
```

The second thing is the side of the hex bin, which we can compute from its area. The approximate area of the desired hex bin is

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

Since the area of the hexagon is  $3 * \text{sqrt}(3) * \text{side}^2 / 2$  is side is equal to  $\text{sqrt}(2 * \text{sqrt}(3) * A/9)$

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

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

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

```
curr = project.conn.cursor()
b = extent.bounds
curr.execute(
    "select st_asbinary(HexagonalGrid(GeomFromWKB(?), ?, 0, GeomFromWKB(?)))",
    [extent.wkb, zone_side, Point(b[2], b[3]).wkb],
)
grid = curr.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(grid):
    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

```
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)
```

(continues on next page)

(continued from previous page)

```
# This takes a few minutes to compute, so we will break after processing the first
↪ 10 zones
if zone_id >= 10:
    break
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪ network/connector_creation.py:85: UserWarning: One of the clusters is empty. Re-run
↪ kmeans with a different initialization.
centroids, allocation = kmeans2(whitened, num_connectors)
```

Let's add special generator zones We also add a centroid at the airport terminal

```
nodes = project.network.nodes
```

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. Distance here is in degrees, so 0.01 is equivalent to roughly 1.1km

```
airport.connect_mode(airport.geometry.buffer(0.01), mode_id="c", link_types="ytrusP",
↪ connectors=1)
```

```
project.close()
```

**Total running time of the script:** (0 minutes 4.438 seconds)

## 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.

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

import folium
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
```

Let's create an empty project on an arbitrary folder.

```
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "coquimbo")
```

As the Coquimbo example already has a complete GTFS model, we shall remove its public transport database for the sake of this example.

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

Let's import the GTFS feed.

```
dest_path = join(fldr, "gtfs_coquimbo.zip")
```

Now we create our Transit object and import the GTFS feed into our model. This will automatically create a new public transport database.

```
data = Transit(project)

transit = data.new_gtfs_builder(agency="Lisanco", file_path=dest_path)
```

To load the data, we must choose one date. We're going to continue with 2016-04-13 but feel free to experiment with any other available dates. Transit class has a function allowing you to check dates for the GTFS feed. It should take approximately 2 minutes to load the data.

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

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

# 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")

links = 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)
```

```
gtfs_links = folium.FeatureGroup("links")
gtfs_stops = folium.FeatureGroup("stops")

layers = [gtfs_links, gtfs_stops]
```

```
pattern_colors = ["#146DB3", "#EB9719"]
```

```
for i, row in links.iterrows():
    points = row.geom.replace("MULTILINESTRING", "").replace("(", "").replace(")", "").
↳split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
```

(continues on next page)



(continued from previous page)

```

points = [[x[1], x[0]] for x in eval(points)]

_ = folium.vector_layers.PolyLine(
    points,
    popup=f"<b>pattern_id: {row.pattern_id}</b>",
    color=pattern_colors[i],
    weight=5,
).add_to(gtfs_links)

for i, row in stops.iterrows():
    point = (row.Y, row.X)

    _ = folium.vector_layers.CircleMarker(
        point,
        popup=f"<b>stop_id: {row.stop_id}</b>",
        color="black",
        radius=2,
        fill=True,
        fillColor="black",
        fillOpacity=1.0,
    ).add_to(gtfs_stops)

```

Let's create the map!

```

map_osm = folium.Map(location=[-29.93, -71.29], zoom_start=13)

# add all layers
for layer in layers:
    layer.add_to(map_osm)

# And add layer control before we display it
folium.LayerControl().add_to(map_osm)
map_osm

```

```
project.close()
```

**Total running time of the script:** (0 minutes 26.250 seconds)

## Importing network 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. Here's the repository: <https://github.com/zephyr-data-specs/GMNS>

```

# Imports
from uuid import uuid4
from os.path import join
from tempfile import gettempdir
from aequilibrae.project 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)

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
)
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↳network/gmns_builder.py:228: FutureWarning: Setting an item of incompatible dtype is
↳deprecated and will raise in a future error of pandas. Value '[-71.15427901 -71.
↳15136675 -71.15133733 -71.1551403 -71.15469925
-71.15315227 -71.15214063 -71.15219271 -71.15159845 -71.15143163
-71.15498211 -71.15514462 -71.15462884 -71.15464758 -71.15317057
-71.15288507 -71.15323051 -71.15342915 -71.15199735 -71.15216062]' has dtype
↳incompatible with int64, please explicitly cast to a compatible dtype first.
    self.node_df.loc[:, "x_coord"] = np.around(lons, decimals=10)
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↳network/gmns_builder.py:229: FutureWarning: Setting an item of incompatible dtype is
↳deprecated and will raise in a future error of pandas. Value '[42.41718801 42.41661235
↳42.41468574 42.41394787 42.41597338 42.41551617
42.4150759 42.41411134 42.41663501 42.41637699 42.41394184 42.41407387
```

(continues on next page)

(continued from previous page)

```

42.41604675 42.4158843 42.41569593 42.41552119 42.41531658 42.41543892
42.41515063 42.41494944]' has dtype incompatible with int64, please explicitly cast to
↳ a compatible dtype first.
    self.node_df.loc[:, "y_coord"] = np.around(lats, decimals=10)
Fields not imported from node table: wkt_coord. If you want them to be imported, please
↳ modify the parameters.yml file.

```

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](https://github.com/zephyr-data-specs/GMNS/blob/develop/examples/Arlington_Signals/README.md)

```

links = project.network.links.data
nodes = project.network.nodes.data

```

We create our Folium layers

```

network_links = folium.FeatureGroup("links")
network_nodes = folium.FeatureGroup("nodes")
layers = [network_links, network_nodes]

```

We do some Python magic to transform this dataset into the format required by Folium We are only getting link\_id and link\_type into the map, but we could get other pieces of info as well

```

for i, row in links.iterrows():
    points = row.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "
↳ ").split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
    # we need to take from x/y to lat/long
    points = [[x[1], x[0]] for x in eval(points)]

    _ = folium.vector_layers.PolyLine(
        points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}", color=
↳ "black", weight=2
    ).add_to(network_links)

```

And now we get the nodes

```

for i, row in nodes.iterrows():
    point = (row.geometry.y, row.geometry.x)

    _ = folium.vector_layers.CircleMarker(
        point,
        popup=f"<b>link_id: {row.node_id}</b>",
        tooltip=f"{row.modes}",
        color="red",
        radius=5,
        fill=True,
        fillColor="red",
        fillOpacity=1.0,
    ).add_to(network_nodes)

```

We get the center of the region

```
curr = project.conn.cursor()
curr.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
long, lat = curr.fetchone()
```

We create the map

```
map_gmns = folium.Map(location=[lat, long], zoom_start=17)

# Add all layers
for layer in layers:
    layer.add_to(map_gmns)

# And Add layer control before we display it
folium.LayerControl().add_to(map_gmns)
map_gmns
```

```
project.close()
```

**Total running time of the script:** (0 minutes 1.571 seconds)

### 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.

```
# 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
import folium

from aequilibrae import Project
```

We create an empty project on an arbitrary folder

```
fldr = join(gettempdir(), uuid4().hex)
project = Project()
project.new(fldr)
```

Now we obtain the link data for our example (in this case from a link layer we will download from the AequilibraE website). With data, we load it on Pandas

```
dest_path = join(fldr, "queluz.csv")
urllib.request.urlretrieve("https://aequilibrae.com/data/queluz.csv", dest_path)

df = pd.read_csv(dest_path)
```

Let's see if we have to add new link\_types to the model before we add links The links we have in the data are:

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

And the existing link types are

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

We could also get it directly from the project database

```
# existing_types = [x[0] for x in project.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 lose 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 Pandas DataFrame so we can process it easier

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

We create a Folium layer

```
network_links = folium.FeatureGroup("links")
```

We do some Python magic to transform this dataset into the format required by Folium. We are only getting *link\_id* and *link\_type* into the map, but we could get other pieces of info as well

```
for i, row in links.iterrows():
    points = row.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "
↪").split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
    # We need to take from x/y to lat/long
    points = [[x[1], x[0]] for x in eval(points)]

    line = folium.vector_layers.PolyLine(
        points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.link_type}",
↪color="blue", weight=10
    ).add_to(network_links)
```

We get the center of the region we are working with some SQL magic

```
curr = project.conn.cursor()
curr.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
long, lat = curr.fetchone()
```

```
map_osm = folium.Map(location=[lat, long], zoom_start=15)
network_links.add_to(map_osm)
folium.LayerControl().add_to(map_osm)
map_osm
```

```
project.close()
```

**Total running time of the script:** (0 minutes 2.640 seconds)

## 1.7.2 Editing networks

### Editing network geometry: Nodes

In this example, we show how to move a node in the network and look into what happens to the links.

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
from shapely.geometry import Point
import matplotlib.pyplot as plt
```

We create the example project inside our temp folder.

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

Let's move node one from the upper left corner of the image above, a bit to the left and to the bottom.

We also add the node we want to move.

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

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

If you want to show the path in Python.

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

We plot the entire network.

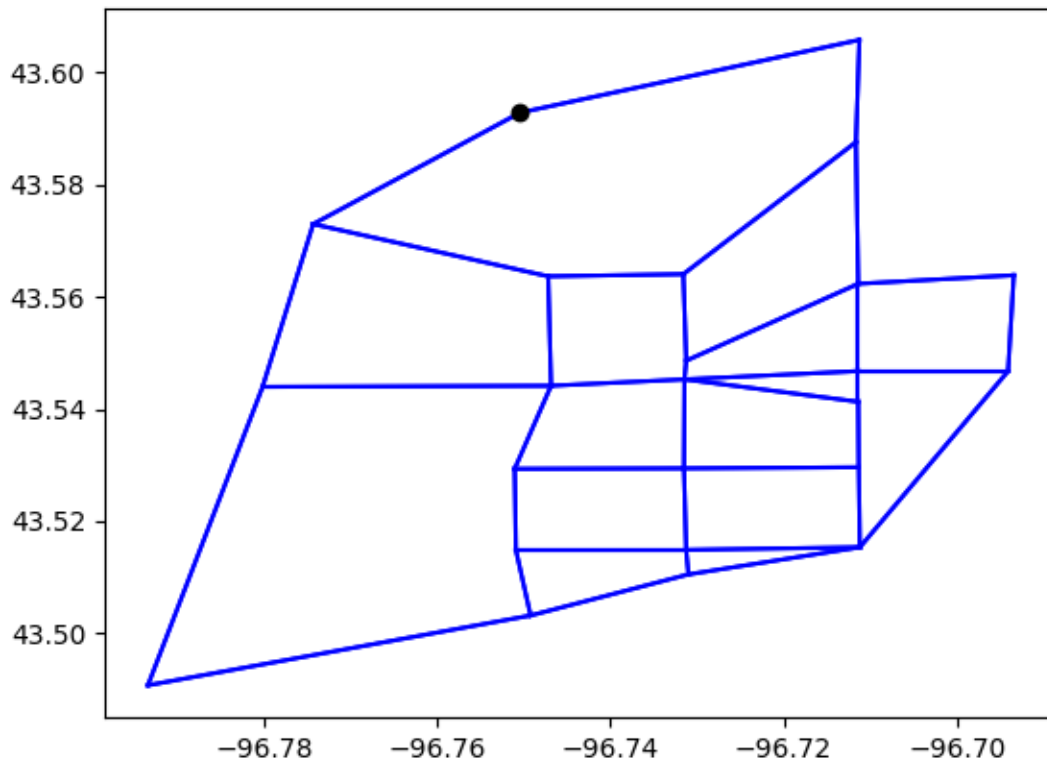
```
links.refresh()
curr = project.conn.cursor()
curr.execute("Select link_id from links;")

for lid in curr.fetchall():
    geo = links.get(lid[0]).geometry
```

(continues on next page)

(continued from previous page)

```
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()
```

**Total running time of the script:** (0 minutes 0.531 seconds)

### Editing network geometry: Links

In this example, we move a link extremity from one point to another and see what happens to the network.

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
from shapely.geometry import LineString, Point
import matplotlib.pyplot as plt
```



We create the example project inside our temp folder

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

project = create_example(fldr)
```

```
all_nodes = project.network.nodes
links = project.network.links
```

Let's move node one from the upper left corner of the image above, a bit to the left and to the bottom

```
# We edit the link that goes from node 1 to node 2
link = links.get(1)
node = all_nodes.get(1)
new_extremity = Point(node.geometry.x + 0.02, node.geometry.y - 0.02)
link.geometry = LineString([node.geometry, new_extremity])

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

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.

We plot the entire network.

```
curr = project.conn.cursor()
curr.execute("Select link_id from links;")

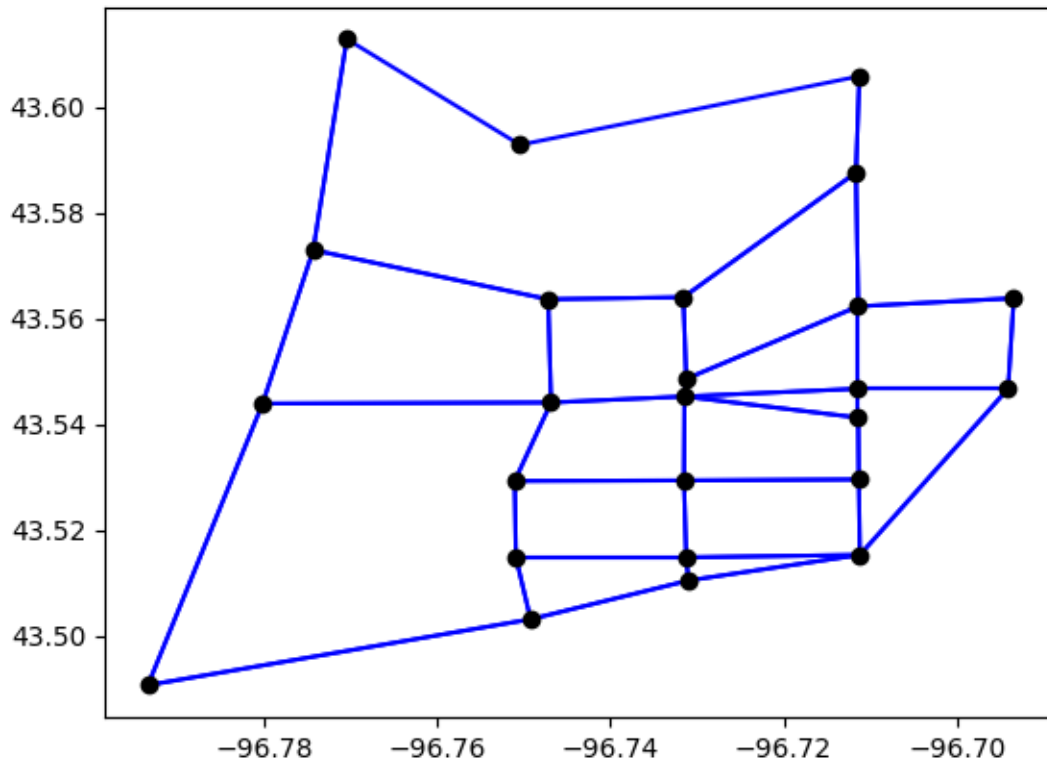
for lid in curr.fetchall():
    geo = links.get(lid[0]).geometry
    plt.plot(*geo.xy, color="blue")

all_nodes = project.network.nodes
curr = project.conn.cursor()
curr.execute("Select node_id from nodes;")

for nid in curr.fetchall():
    geo = all_nodes.get(nid[0]).geometry
    plt.plot(*geo.xy, "o", color="black")

plt.show()

# Now look at the network and how it used to be.
```



```
project.close()
```

**Total running time of the script:** (0 minutes 0.617 seconds)

### 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.

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
from shapely.ops import substring
import matplotlib.pyplot as plt
```

We create the example project inside our temp folder

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

project = create_example(fldr)
```

We will split link 37 right in the middle. Let's get the link and check its length.

```
links = project.network.links
all_nodes = project.network.nodes

link = links.get(37)
print(link.distance)
```

```
6010.108655014215
```

The idea is basically to copy a link and allocate the appropriate geometries to split the geometry we use Shapely's substring.

```
new_link = links.copy_link(37)

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

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

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

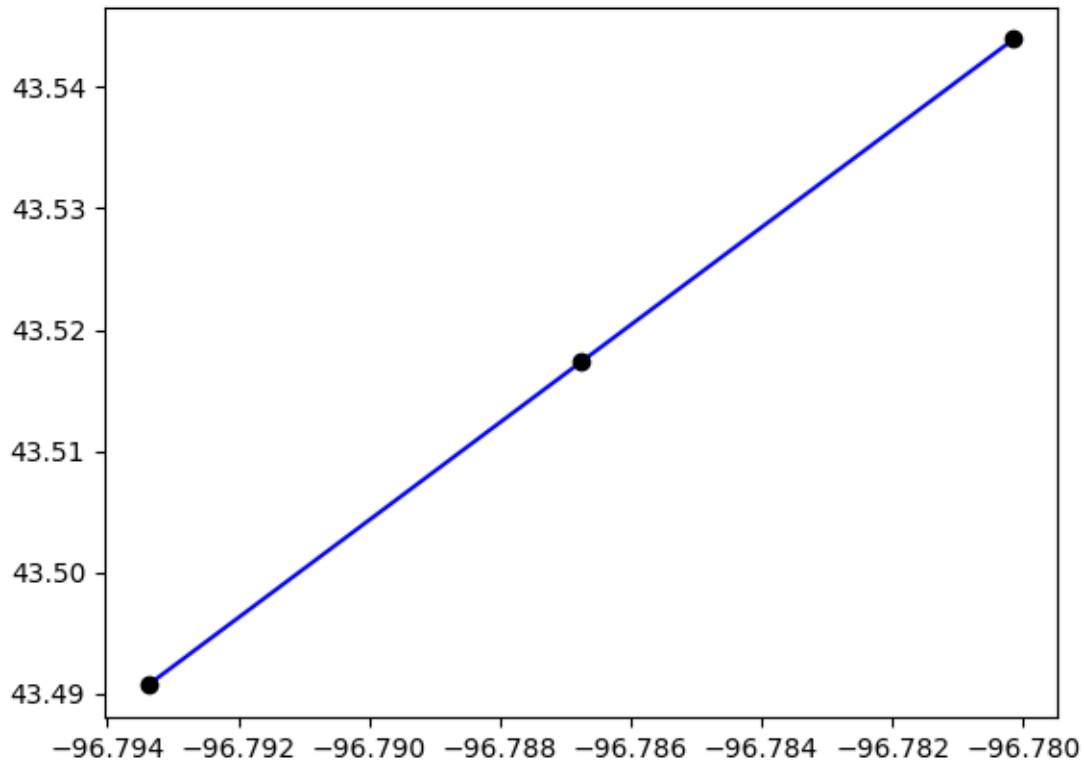
```
links.refresh()
link = links.get(37)
new_link = links.get(new_link.link_id)
print(link.distance, new_link.distance)
```

```
3005.040184141035 3005.0684894898027
```

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

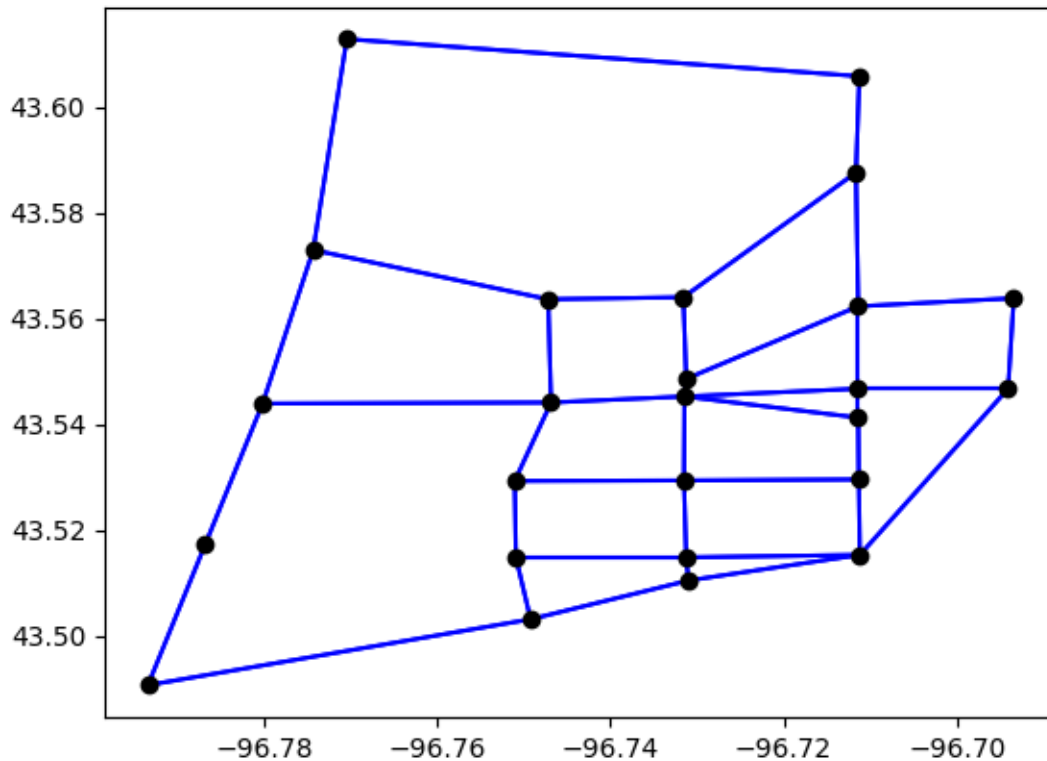
```
plt.clf()
curr = project.conn.cursor()
curr.execute("Select link_id from links;")

for lid in curr.fetchall():
    geo = links.get(lid[0]).geometry
    plt.plot(*geo.xy, color="blue")

all_nodes = project.network.nodes
curr = project.conn.cursor()
curr.execute("Select node_id from nodes;")

for nid in curr.fetchall():
    geo = all_nodes.get(nid[0]).geometry
    plt.plot(*geo.xy, "o", color="black")

plt.show()
```



```
project.close()
```

**Total running time of the script:** (0 minutes 0.731 seconds)

### 1.7.3 Trip Distribution

#### 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 compliment to the application in *Trip Distribution*.

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).

```
# Imports
import numpy as np

from aequilibrae.distribution import Ipf
from os.path import join
from tempfile import gettempdir
from aequilibrae.matrix import AequilibraeMatrix, AequilibraeData
```

```
folder = gettempdir()
```

```
matrix = np.array([[5, 50, 100, 200], [50, 5, 100, 300],  
                  [50, 100, 5, 100], [100, 200, 250, 20]], dtype="float64")  
future_prod = np.array([400, 460, 400, 702], dtype="float64")  
future_attr = np.array([260, 400, 500, 802], dtype="float64")  
  
num_zones = matrix.shape[0]
```

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

```
args = {  
    "entries": mtx.index.shape[0],  
    "field_names": ["productions", "attractions"],  
    "data_types": [np.float64, np.float64],  
    "file_path": join(folder, "vectors.aem"),  
}
```

```
vectors = AequilibraeData()  
vectors.create_empty(**args)  
  
vectors.productions[:] = future_prod[:]  
vectors.attractions[:] = future_attr[:]  
  
vectors.index[:] = mtx.index[:]
```

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

```
fratar.output.matrix_view
```

```
array([[ 5.19523253, 43.60117896, 97.1907419 , 254.02026689],  
       [44.70942645,  3.75225496, 83.64095928, 327.90930057],  
       [76.67583276, 128.70094592,  7.17213428, 187.45277887],  
       [133.41950826, 223.94562016, 311.99616454, 32.61765367]])
```

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

```
#####   IPF computation   #####

Target convergence criteria: 0.0001
Maximum iterations: 5000

Rows:4
Columns: 4
Total of seed matrix:           1,635.0000
Total of target vectors:        1,962.0000

Iteration,   Convergence
7   ,   0.0000902085

Running time: 0.002s
```

## Reference

ORTÚZAR, J.D., WILLUMSEN, L.G. (2011) *Modelling Transport* (4th ed.). Wiley-Blackwell.

**Total running time of the script:** (0 minutes 0.038 seconds)

## Network skimming

In this example, we show how to perform network skimming for Coquimbo, a city in La Serena Metropolitan Area in Chile.

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

We create the example project inside our temp folder

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

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

```
import logging
import sys
```

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

## Network Skimming

```
from aequilibrae.paths import NetworkSkimming
import numpy as np
```

Let's build all graphs

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
    df = pd.read_sql(sql, conn).fillna(value=np.nan)
2024-08-19 06:15:29,631;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:29,716;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:29,821;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:29,919;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

We grab the graph for cars

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

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

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

# And will skim distance while we are at it, other fields like `free_flow_time` or_
↪`travel_time`
# can be added here as well
graph.set_skimming(["distance"])

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

```
2024-08-19 06:15:30,001;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

And run the skimming

```
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, :]
```

```
array([[ 0.          ],
       [4166.92919206],
       [5532.32681478]],

      [[3733.4499255 ],
       [ 0.          ],
       [3311.30654014]],

      [[5446.26074416],
       [3596.12274848],
       [ 0.          ]]])
```

Or access each matrix, lets just look at the first 3x3

```
skims.distance[:3, :3]
```

```
array([[ 0.          , 4166.92919206, 5532.32681478],
       [3733.4499255 ,  0.          , 3311.30654014],
       [5446.26074416, 3596.12274848,  0.          ]])
```

We can save it to the project if we want

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

```
2024-08-19 06:15:30,211;WARNING ; Matrix Record has been saved to the database
```

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()
```

**Total running time of the script:** (0 minutes 0.990 seconds)

## Path computation

In this example, we show how to perform path computation for Coquimbo, a city in La Serena Metropolitan Area in Chile.

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

We create the example project inside our temp folder

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

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

```
import logging
import sys
```

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

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

## Path Computation

```
from aequilibrae.paths import PathResults
```

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.
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
2024-08-19 06:15:30,746;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:30,831;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:30,933;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:15:31,034;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

We grab the graph for cars

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

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

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

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

# And we will allow paths to be computed going through other centroids/centroid,
↳ connectors.
# We recommend you to `be extremely careful` with this setting.
graph.set_blocked_centroid_flows(False)
```

Let's instantiate a path results object and prepare it to work with the graph

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

# compute a path from node 32343 to 22041, thats from near the airport to Fort Lambert,
# a popular location due to its views of the Coquimbo bay.
res.compute_path(32343, 22041)

# We can get the sequence of nodes we traverse
res.path_nodes
```

```
array([32343, 79778, 68225, 32487, 63937, 63192, 46510, 32380, 32373,
       55817, 55816, 11982, 46516, 75015, 79704, 79785, 78576, 68242,
       79144, 78635, 79784, 78748, 79082, 65861, 78343, 21311, 20312,
       21308, 78834, 79862, 79450, 63873, 79458, 78986, 78884, 79152,
       78645, 78549, 68503, 13380, 13383, 79199, 79745, 79457, 80001,
       78217, 78093, 80013, 25130, 80012, 40633, 11010, 11009, 40846,
       21827, 80056, 80055, 79481, 79486, 79485, 75142, 11448, 11446,
       11445, 67684, 60645, 11447, 11422, 11420, 11421, 13723, 10851,
       79462, 26681, 13718, 12079, 79460, 23707, 29778, 75451, 75445,
       45342, 39399, 13626, 13627, 45379, 21384, 63812, 40005, 12207,
       44243, 44241, 23405, 60002, 27114, 79431, 15148, 15146, 60000,
       75486, 55963, 55958, 59043, 59050, 59988, 39402, 59017, 59019,
       79398, 75520, 75516, 75512, 75509, 75505, 75511, 63544, 63543,
       75510, 75515, 75476, 63539, 30138, 11695, 61061, 30148, 44192,
       75556, 79364, 75534, 75552, 75548, 75321, 75532, 14802, 14823,
       71435, 65497, 64708, 64709, 64712, 64713, 40374, 40375, 77308,
       65518, 75566, 68526, 75573, 41306, 41308, 75619, 75617, 14899,
       14875, 38674, 75595, 65067, 65068, 79508, 29452, 44797, 29447,
       10065, 44798, 30552, 44783, 44808, 75612, 73617, 79653, 79651,
       73620, 73923, 79820, 14864, 69009, 22040, 22041])
```

We can get the link sequence we traverse

```
res.path
```

```
array([34709, 34710, 34711, 34712, 34713, 34714, 34715, 34716, 34717,
       34718, 34719, 34720, 34721, 34722, 3321, 3322, 3323, 3324,
       3325, 3326, 3327, 3328, 3329, 3330, 3331, 3332, 2970,
       2971, 2969, 19995, 1434, 1435, 1436, 19326, 19327, 19328,
       19329, 19330, 33674, 33675, 33676, 33677, 26525, 20765, 20746,
       20747, 20748, 20749, 20750, 20751, 20752, 496, 497, 498,
       499, 500, 501, 10380, 15408, 553, 552, 633, 634,
       635, 630, 631, 632, 623, 624, 625, 626, 471,
       5363, 34169, 34170, 34171, 34785, 6466, 6465, 29938, 29939,
       29940, 29941, 1446, 1447, 1448, 1449, 1450, 939, 940,
       941, 9840, 9841, 26314, 26313, 26312, 26311, 26310, 26309,
       26308, 26307, 26306, 26305, 26304, 26303, 26302, 26301, 26300,
       34079, 34147, 29962, 26422, 26421, 26420, 765, 764, 763,
       762, 761, 760, 736, 10973, 10974, 10975, 725, 10972,
       727, 728, 26424, 733, 734, 29899, 20970, 20969, 20968,
       20967, 20966, 20965, 20964, 20963, 20962, 9584, 9583, 20981,
       21398, 20982, 20983, 20984, 20985, 10030, 10031, 10032, 10033,
       10034, 10035, 10036, 64, 65, 21260, 21261, 21262, 21263,
       21264, 21265, 21266, 33, 11145, 11146, 71, 72, 34529,
       34530, 34531, 28691, 28692, 28693, 3574])
```

We can get the mileposts for our sequence of nodes

```
res.milepost
```

```
array([ 0.          , 161.94834565, 252.51996291, 390.08508135,
       549.82648658, 561.81026027, 581.36871507, 593.21987605,
       630.08836889, 1030.57121686, 1052.34444478, 1112.07906484,
       1165.81004929, 1267.22602763, 1624.43103234, 1924.50863633,
       1972.01917098, 2021.6997169 , 2062.34315111, 2109.67655695,
       2155.98823775, 2196.91106309, 2221.69061004, 2249.01761535,
       2298.72337036, 2363.21417492, 2375.9615406 , 2392.22488207,
       2426.81462733, 2675.27978499, 3632.15818275, 3699.27505758,
       3823.65932479, 3956.65040737, 4017.46560312, 4072.01402297,
       4129.64308736, 4163.12532905, 4187.96101397, 4224.21499938,
       4307.91646806, 4323.79479478, 4458.86701963, 4569.20833913,
       4692.25740782, 4930.34266637, 4997.45500599, 5068.23739204,
       5120.12897437, 5187.77254139, 5207.89952503, 5327.95612724,
       5368.62533167, 5378.38940327, 5385.33334293, 5426.19568418,
       5468.3231786 , 5523.97170089, 5548.40671886, 5559.07102378,
       5592.3855075 , 5757.7535581 , 5923.39441593, 5950.82694781,
       5956.69374937, 5982.88386915, 6047.00805103, 6254.07203583,
       6284.94026694, 6297.98296824, 7047.68761904, 7322.48973364,
       7410.01354224, 7554.81816989, 7654.30645223, 8000.95440184,
       8009.6579379 , 8048.90437039, 8056.1819337 , 8230.32269321,
       8476.75016293, 8620.89727688, 9010.45521832, 9274.61196368,
       9280.12960244, 9383.74516275, 9496.88846171, 9711.90807592,
       10093.92389013, 10097.37343293, 10098.79806269, 10100.74846208,
       10158.97660612, 10308.02183308, 11038.25297634, 11043.82678472,
       11184.07536528, 11241.90536698, 11251.4005022 , 11507.50220541,
```

(continues on next page)

(continued from previous page)

```

11734.64237287, 11891.6028426 , 11914.05783947, 11931.91664218,
11955.2823231 , 12032.22401165, 12107.39910016, 12119.5125075 ,
12136.71574687, 12210.28469894, 12389.41094897, 12417.43348286,
12718.35052704, 12730.00818724, 12843.38553893, 12911.9429086 ,
12921.34664177, 13071.40008271, 13080.33320947, 13161.6119503 ,
13335.34223859, 13388.53619865, 13412.08272452, 13574.14596808,
13738.18613296, 13798.71647839, 14036.00947087, 14102.56840025,
14197.10918933, 14292.09203161, 14315.76699626, 14434.43917168,
14484.88708304, 14491.84542125, 14851.61908868, 14868.34527227,
14877.57764797, 15028.49457611, 15037.54620347, 15204.94094821,
15215.00009437, 15351.10413143, 15623.06737615, 15672.61684115,
15722.54738786, 15807.97421917, 15901.71819152, 15937.27964123,
16179.52060712, 16191.20903769, 16203.56934613, 16308.11327422,
16479.15664668, 16536.53372128, 16548.12481843, 16559.65540426,
16680.70266884, 16720.69837006, 16796.68692111, 16841.49843813,
16877.97092451, 16889.16327603, 16920.08515864, 16976.52291632,
17076.47607269, 17094.22058834, 17134.27959626, 17183.85951634,
17472.72121764, 17554.85677394, 17659.06248146, 17923.55576674,
17933.3892542 , 17942.66958063, 18164.10699022, 18421.89973826,
18597.64939792, 18599.05175306])

```

Additionally we could also provide `early_exit=True` or `a_star=True` to `compute_path` to adjust its path finding behaviour. Providing `early_exit=True` will allow the path finding to quit once it's discovered the destination, this means it will perform better for ODs that are topographically close. However, exiting early may cause subsequent calls to `update_trace` to recompute the tree in cases where it usually wouldn't. `a_star=True` has precedence of `early_exit=True`.

```
res.compute_path(32343, 22041, early_exit=True)
```

If you'd prefer to find a potentially non-optimal path to the destination faster provide `a_star=True` to use A\* with a heuristic. With this method `update_trace` will always recompute the path.

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

By default a equirectangular heuristic is used. We can view the available heuristics via

```
res.get_heuristics()
```

```
['haversine', 'equirectangular']
```

If you'd like the more accurate, but slower, but more accurate haversine heuristic you can set it using

```
res.set_heuristic("haversine")
```

or

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

If we want to compute the path for a different destination and the same origin, we can just do this. It is way faster when you have large networks. Here we'll adjust our path to the University of La Serena. Our previous early exit and A\* settings will persist with calls to `update_trace`. If you'd like to adjust them for subsequent path re-computations set the `res.early_exit` and `res.a_star` attributes.

```
res.a_star = False
res.update_trace(73131)
```

```
res.path_nodes
```

```
array([32343, 79778, 68225, 32487, 63937, 63192, 46510, 32380, 32373,
       55817, 55816, 11982, 46516, 75015, 79704, 79785, 78576, 68242,
       79144, 78635, 79784, 78748, 79082, 65861, 78343, 21311, 20312,
       21308, 78834, 79862, 79450, 63873, 79458, 78986, 78884, 79152,
       78645, 78549, 68503, 13380, 13383, 79199, 79745, 79457, 80001,
       78217, 78093, 80013, 25130, 80012, 40633, 11010, 11009, 40846,
       21827, 80056, 80055, 79481, 79486, 79485, 75142, 11448, 11446,
       11445, 67684, 60645, 11447, 11422, 11420, 11421, 13723, 10851,
       79462, 26681, 13718, 12079, 79460, 23707, 29778, 75451, 75445,
       45342, 39399, 13626, 13627, 45379, 21384, 63812, 40005, 12207,
       44243, 44241, 23405, 60002, 27114, 79431, 15148, 15146, 60000,
       75486, 55963, 55958, 59043, 59050, 59988, 39402, 59017, 59019,
       79398, 75520, 75516, 75512, 75509, 75505, 75511, 63544, 63543,
       75510, 75515, 75476, 63539, 30138, 11695, 61061, 30148, 44192,
       75556, 79364, 75534, 75552, 75548, 75321, 75532, 14802, 14823,
       71435, 65497, 64708, 64709, 64712, 64713, 40374, 40375, 77308,
       65518, 75566, 68526, 79517, 51754, 77189, 65059, 10093, 65058,
       30491, 66966, 66863, 30492, 77190, 77191, 79366, 77417, 79368,
       77406, 77421, 77425, 77393, 77398, 53993, 77394, 70959, 77395,
       27752, 65293, 73131])
```

If you want to show the path in Python.

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

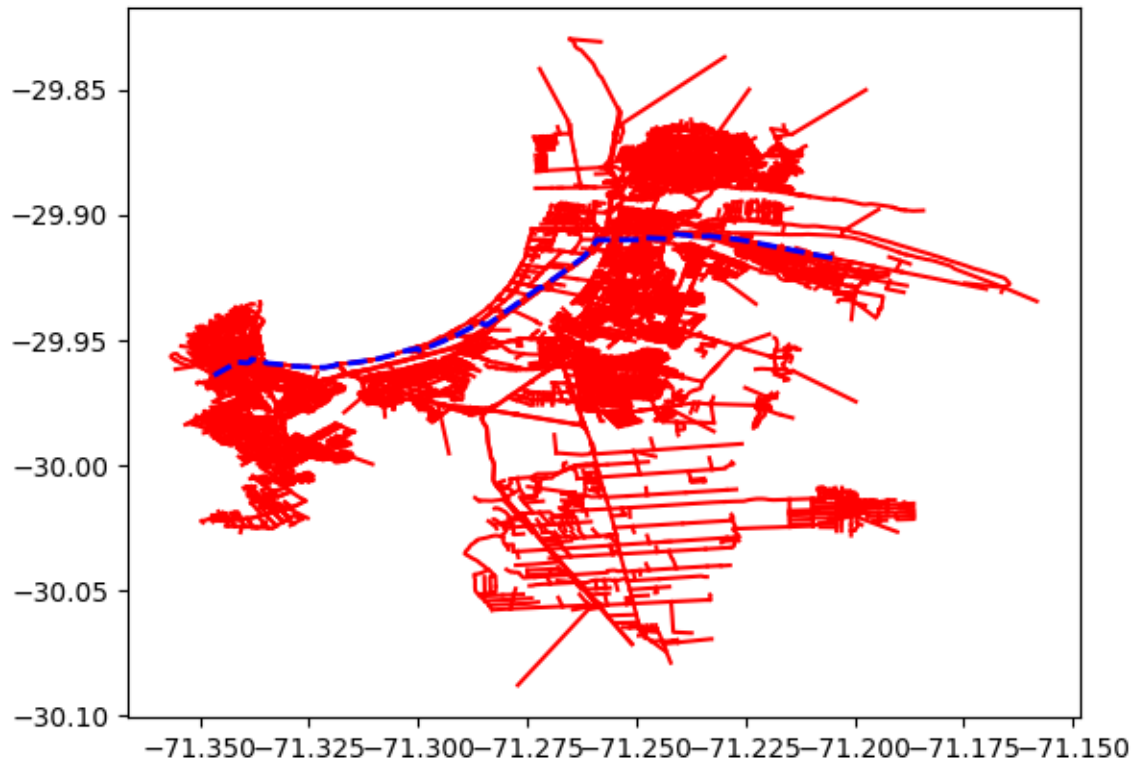
```
import matplotlib.pyplot as plt
from shapely.ops import linemerge
```

```
links = project.network.links

# We plot the entire network
curr = project.conn.cursor()
curr.execute("Select link_id from links;")

for lid in curr.fetchall():
    geo = links.get(lid[0]).geometry
    plt.plot(*geo.xy, color="red")

path_geometry = linemerge(links.get(lid).geometry for lid in res.path)
plt.plot(*path_geometry.xy, color="blue", linestyle="dashed", linewidth=2)
plt.show()
```



```
project.close()
```

**Total running time of the script:** (1 minutes 24.087 seconds)

### Trip Distribution

In this example, we calibrate a Synthetic Gravity Model that same model plus IPF (Fratrar/Furness).

```
# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from aequilibrae.utils.create_example import create_example
import pandas as pd
import numpy as np
```

We create the example project inside our temp folder

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

We get the demand matrix directly from the project record so let's inspect what we have in the project

```
proj_matrices = project.matrices
print(proj_matrices.list())
```

```

      name      file_name  ...      description status
0  demand_omx  demand.omx  ...  Original data imported to OMX format
1  demand_mc   demand.mc.omx  ...      None
2      skims     skims.omx  ...      Example skim
3  demand_aem   demand.aem  ...  Original data imported to AEM format

[4 rows x 8 columns]
```

We get the demand matrix

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

And the impedance

```
impedance = proj_matrices.get_matrix("skims")
impedance.computational_view(["time_final"])
```

Let's have a function to plot the Trip Length Frequency Distribution

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

```
def plot_tlfd(demand, skim, name):
    plt.clf()
    b = floor(log10(skim.shape[0]) * 10)
    n, bins, patches = plt.hist(
        np.nan_to_num(skim.flatten(), 0),
        bins=b,
        weights=np.nan_to_num(demand.flatten()),
        density=False,
        facecolor="g",
        alpha=0.75,
    )

    plt.xlabel("Trip length")
    plt.ylabel("Probability")
    plt.title("Trip-length frequency distribution")
    plt.savefig(name, format="png")
    return plt
```

```
from aequilibrae.distribution import GravityCalibration
```

```
for function in ["power", "expo"]:
    gc = GravityCalibration(matrix=demand, impedance=impedance, function=function, nan_
    ↪as_zero=True)
    gc.calibrate()
    model = gc.model
    # We save the model
```

(continues on next page)



(continued from previous page)

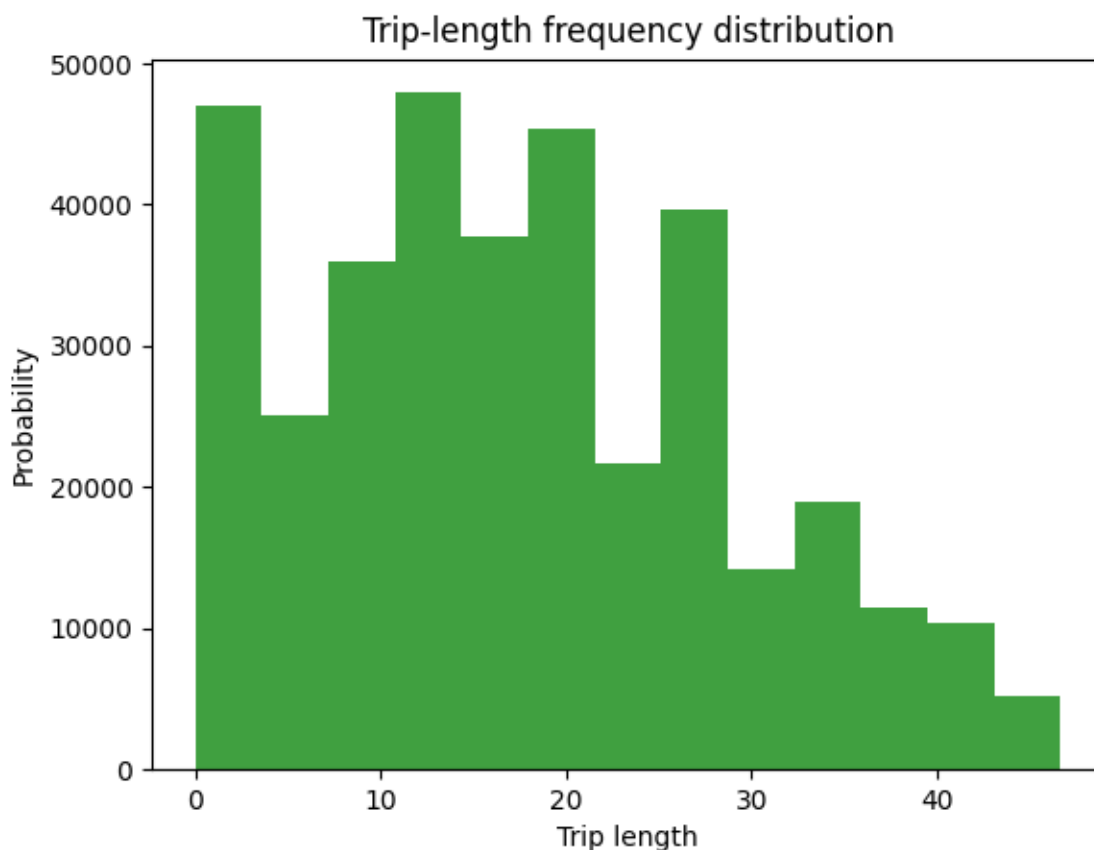
```

model.save(join(fldr, f"{function}_model.mod"))

# We can save an image for the resulting model
_ = plot_tlfd(gc.result_matrix.matrix_view, impedance.matrix_view, join(fldr, f"
↪{function}_tlfd.png"))

# 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")

```



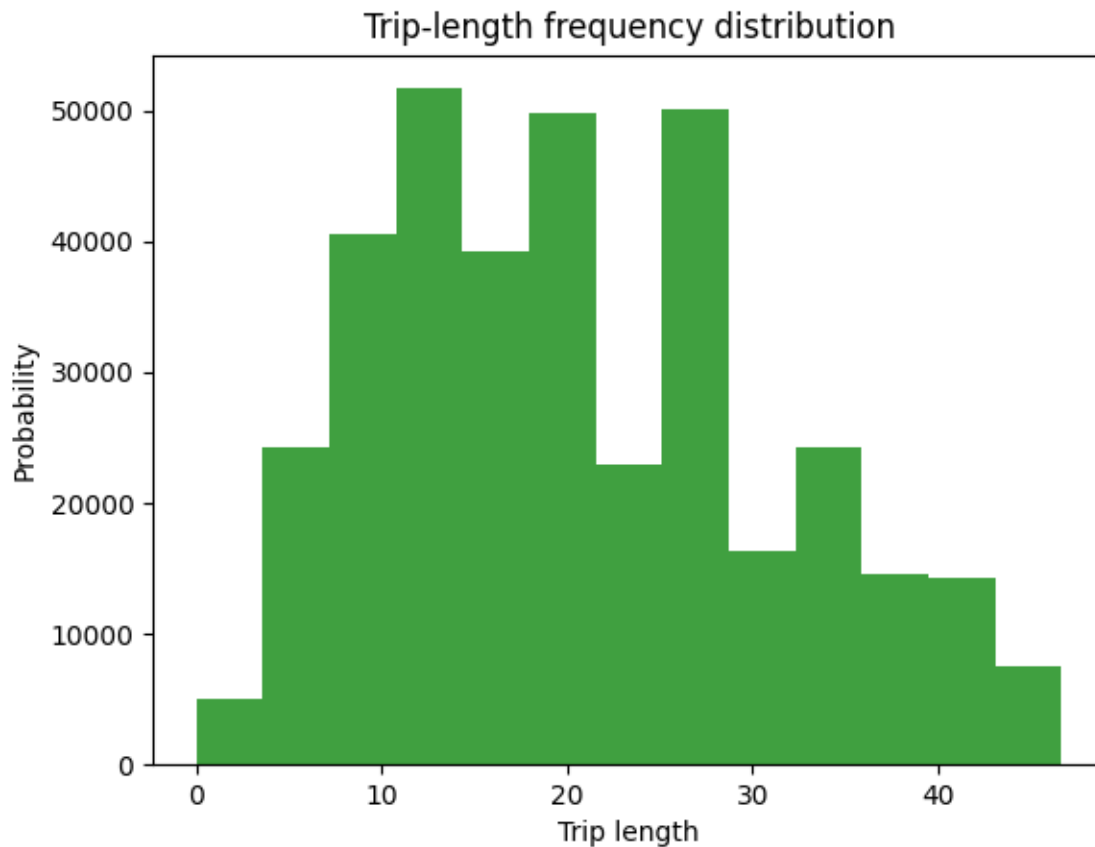
```

/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/
↪distribution/gravity_application.py:321: RuntimeWarning: divide by zero encountered in _
↪power
    self.output.matrix_view[i, :] = (np.power(self.impedance.matrix_view[i, :], -self.
↪model.alpha) * p * a)[
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/
↪distribution/gravity_application.py:335: RuntimeWarning: invalid value encountered in _
↪multiply
    self.output.matrix_view[:, :] = self.output.matrix_view[:, :] * non_inf

```

We save a trip length frequency distribution for the demand itself

```
plt = plot_tlfld(demand.matrix_view, impedance.matrix_view, join(fldr, "demand_tfld.png"))
plt.show()
```



## Forecast

We create a set of *'future'* vectors by applying some models and apply the model for both deterrence functions

```
from aequilibrae.distribution import Ipfd, GravityApplication, SyntheticGravityModel
from aequilibrae.matrix import AequilibraeData
import numpy as np
```

```
zonal_data = pd.read_sql("Select zone_id, population, employment from zones order by ↵
↵zone_id", project.conn)
```

We compute the vectors from our matrix

```
args = {
    "file_path": join(fldr, "synthetic_future_vector.aed"),
    "entries": demand.zones,
    "field_names": ["origins", "destinations"],
    "data_types": [np.float64, np.float64],
    "memory_mode": True,
```

(continues on next page)

(continued from previous page)

```

}

vectors = AequilibraeData()
vectors.create_empty(**args)

vectors.index[:] = zonal_data.zone_id[:]

# We apply a trivial regression-based model and balance the vectors
vectors.origins[:] = zonal_data.population[:] * 2.32
vectors.destinations[:] = zonal_data.employment[:] * 1.87
vectors.destinations *= vectors.origins.sum() / vectors.destinations.sum()

```

We simply apply the models to the same impedance matrix now

```

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

    outmatrix = join(proj_matrices.flldr, f"demand_{function}_model.aem")
    args = {
        "impedance": impedance,
        "rows": vectors,
        "row_field": "origins",
        "model": model,
        "columns": vectors,
        "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}_model_omx", file_name=f"demand_
↪{function}_model_omx")

```

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

```

proj_matrices.update_database()
print(proj_matrices.list())

```

```

          name  ... status
0      demand_omx  ...
1      demand_mc  ...
2          skims  ...
3      demand_aem  ...
4 demand_power_model_omx  ...
5 demand_expo_model_omx  ...

[6 rows x 8 columns]

```

We now run IPF for the future vectors

```
args = {
    "matrix": demand,
    "rows": vectors,
    "columns": vectors,
    "column_field": "destinations",
    "row_field": "origins",
    "nan_as_zero": True,
}

ipf = Ipf(**args)
ipf.fit()

ipf.save_to_project(name="demand_ipf", file_name="demand_ipf.aem")
ipf.save_to_project(name="demand_ipf_omx", file_name="demand_ipf.omx")
```

```
<aequilibrae.project.data.matrix_record.MatrixRecord object at 0x7fa9072f0b80>
```

```
print(proj_matrices.list())
```

```

      name  ... status
0      demand_omx  ...
1      demand_mc  ...
2          skims  ...
3      demand_aem  ...
4 demand_power_model_omx  ...
5 demand_expo_model_omx  ...
6      demand_ipf  ...
7      demand_ipf_omx  ...

[8 rows x 8 columns]
```

```
project.close()
```

**Total running time of the script:** (0 minutes 1.388 seconds)

## 1.7.4 Visualization

Examples in this session allows the user to plot some data visualization.

### Creating Delaunay Lines

In this example, we show how to create AequilibraE's famous Delaunay Lines, but in Python.

For more on this topic, the first publication is [here](#).

We use the Sioux Falls example once again.

```
# Imports
import pandas as pd
from uuid import uuid4
from os.path import join
```

(continues on next page)

(continued from previous page)

```
import sqlite3
from tempfile import gettempdir
import matplotlib.pyplot as plt
import shapely.wkb

from aequilibrae.utils.create_example import create_example
from aequilibrae.utils.create_delaunay_network import DelaunayAnalysis
```

We create an empty project on an arbitrary folder

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

Get the Delaunay Lines generation class

```
da = DelaunayAnalysis(project)

# Let's create the triangulation based on the zones, but we could create based on the
↪ network (centroids) too
da.create_network("zones")
```

Now we get the matrix we want and create the Delaunay Lines

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

And we will call it 'delaunay\_test'. It will also be saved in the results\_database.sqlite

```
da.assign_matrix(demand, "delaunay_test")
```

we retrieve the results

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

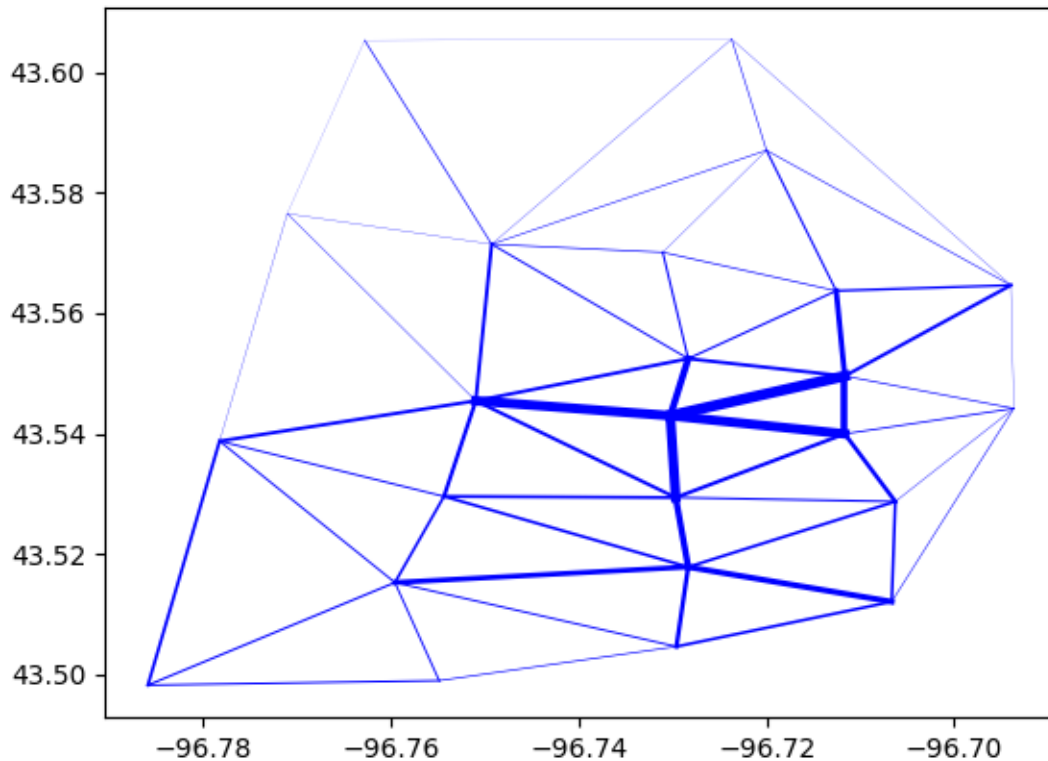
Now we get the matrix we want and create the Delaunay Lines

```
links = pd.read_sql("Select link_id, st_asBinary(geometry) geometry from delaunay_network
↪", project.conn)
links.geometry = links.geometry.apply(shapely.wkb.loads)
links.set_index("link_id", inplace=True)

df = links.join(results)

max_vol = df.matrix_tot.max()

for idx, lnk in df.iterrows():
    geo = lnk.geometry
    plt.plot(*geo.xy, color="blue", linewidth=4 * lnk.matrix_tot / max_vol)
plt.show()
```



Close the project

```
project.close()
```

**Total running time of the script:** (0 minutes 0.500 seconds)

### Exploring the network on a notebook

In this example, we show how to use Folium to plot a network for different modes.

We will need Folium for this example, and we will focus on creating a layer for each mode in the network, a layer for all links and a layer for all nodes.

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

We create an empty project on an arbitrary folder

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

Let's use the Nauru example project for display

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

We grab all the links data as a Pandas dataframe so we can process it easier

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

We create our Folium layers

```
network_links = folium.FeatureGroup("links")
network_nodes = folium.FeatureGroup("nodes")
car = folium.FeatureGroup("Car")
walk = folium.FeatureGroup("Walk")
bike = folium.FeatureGroup("Bike")
transit = folium.FeatureGroup("Transit")
layers = [network_links, network_nodes, car, walk, bike, transit]
```

We do some Python magic to transform this dataset into the format required by Folium We are only getting link\_id and link\_type into the map, but we could get other pieces of info as well

```
for i, row in links.iterrows():
    points = row.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "
    ↪").split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
    # we need to take from x/y to lat/long
    points = [[x[1], x[0]] for x in eval(points)]

    _ = folium.vector_layers.PolyLine(
        points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}", color=
    ↪"gray", weight=2
    ).add_to(network_links)

    if "w" in row.modes:
        _ = folium.vector_layers.PolyLine(
            points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}",
    ↪color="green", weight=4
        ).add_to(walk)

    if "b" in row.modes:
        _ = folium.vector_layers.PolyLine(
            points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}",
    ↪color="green", weight=4
        ).add_to(bike)

    if "c" in row.modes:
        _ = folium.vector_layers.PolyLine(
            points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}",
    ↪color="red", weight=4
        ).add_to(car)

    if "t" in row.modes:
        _ = folium.vector_layers.PolyLine(
            points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.modes}",
```

(continues on next page)

(continued from previous page)

```

↪color="yellow", weight=4
    ).add_to(transit)

# And now we get the nodes
for i, row in nodes.iterrows():
    point = (row.geometry.y, row.geometry.x)

    _ = folium.vector_layers.CircleMarker(
        point,
        popup=f"<b>link_id: {row.node_id}</b>",
        tooltip=f"{row.modes}",
        color="black",
        radius=5,
        fill=True,
        fillColor="black",
        fillOpacity=1.0,
    ).add_to(network_nodes)

```

We get the center of the region we are working with some SQL magic

```

curr = project.conn.cursor()
curr.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
long, lat = curr.fetchone()

```

We create the map

```

map_osm = folium.Map(location=[lat, long], zoom_start=14)

# add all layers
for layer in layers:
    layer.add_to(map_osm)

# And Add layer control before we display it
folium.LayerControl().add_to(map_osm)
map_osm

```

```
project.close()
```

**Total running time of the script:** (0 minutes 7.936 seconds)

## 1.7.5 AequilibraE without a Model

### 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 TNTF.

```

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

```

(continues on next page)



(continued from previous page)

```
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 = gettempdir()
```

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 our data

```
aemfile = os.path.join(folder, "demand.aem")
aem = AequilibraeMatrix()
kwargs = {'file_name': aemfile,
          'zones': zones,
          'matrix_names': ['matrix'],
          "memory_only": False} # We'll save it to disk so we can use it later

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)
```

(continues on next page)

(continued from previous page)

```
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 perform our assignment. Feel free to try different algorithms, as well as change the maximum number of iterations and the gap.

```
aem = AequilibraeMatrix()
aem.load(aemfile)
aem.computational_view(["matrix"])

assignclass = TrafficClass("car", g, aem)

assign = TrafficAssignment()

assign.set_classes([assignclass])
assign.set_vdf("BPR")
assign.set_vdf_parameters({"alpha": "b", "beta": "power"})
assign.set_capacity_field("capacity")
assign.set_time_field("free_flow_time")
```

(continues on next page)

(continued from previous page)

```

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

```
print(assig.results())
```

link_id	matrix_ab	matrix_ba	...	PCE_BA	PCE_tot
1	4532.416460	NaN	...	NaN	4532.416460
2	8124.962104	NaN	...	NaN	8124.962104
3	4528.444976	NaN	...	NaN	4528.444976
4	6001.323525	NaN	...	NaN	6001.323525
5	8128.933588	NaN	...	NaN	8128.933588
...	...	...	...	...	...
72	9643.868005	NaN	...	NaN	9643.868005
73	7855.662507	NaN	...	NaN	7855.662507
74	11101.449987	NaN	...	NaN	11101.449987
75	10255.489839	NaN	...	NaN	10255.489839
76	7923.866336	NaN	...	NaN	7923.866336

[76 rows x 18 columns]

And at the Assignment report

```
print(assig.report())
```

	iteration	rgap	alpha	warnings
0	1	inf	1.000000	
1	2	0.855131	0.328177	
2	3	0.476738	0.186185	
3	4	0.239622	0.229268	
4	5	0.139851	0.314341	
..	...	...	...	...
95	96	0.001999	0.011309	
96	97	0.001431	0.006948	
97	98	0.001405	0.014356	
98	99	0.001814	0.012088	
99	100	0.001577	0.007687	

[100 rows x 4 columns]

**Total running time of the script:** (0 minutes 1.125 seconds)

## 1.7.6 Assignment Workflows

### Public transport assignment with Optimal Strategies

In this example, we import a GTFS feed to our model, create a public transport network, create project match connectors, and perform a Spiess & Florian assignment.

We use data from Coquimbo, a city in La Serena Metropolitan Area in Chile.

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

from aequilibrae.paths import TransitAssignment, TransitClass
from aequilibrae.utils.create_example import create_example
import numpy as np

# Imports for GTFS import
from aequilibrae.transit import Transit

# Imports for SF transit graph construction
from aequilibrae.project.database_connection import database_connection
from aequilibrae.transit.transit_graph_builder import TransitGraphBuilder
```

Let's create an empty project on an arbitrary folder.

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

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

As the Coquimbo example already has a complete GTFS model, we shall remove its public transport database for the sake of this example.

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

Let's import the GTFS feed.

```
dest_path = join(fldr, "gtfs_coquimbo.zip")
```

Now we create our Transit object and import the GTFS feed into our model. This will automatically create a new public transport database.

```
data = Transit(project)

transit = data.new_gtfs_builder(agency="LISANCO", file_path=dest_path)
```

To load the data, we must choose one date. We're going to continue with 2016-04-13 but feel free to experiment with any other available dates. Transit class has a function allowing you to check dates for the GTFS feed. It should take approximately 2 minutes to load the data.

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

Let's save this model for later use.

```
transit.save_to_disk()
```

## 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()`. We drop geometry here for the sake of display.

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

```
graph.vertices.drop(columns="geometry")
```

```
graph.edges
```

The graphs 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()
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↳ network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↳ ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↳ objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↳ 'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
```

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")
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/transit/
↳ transit_graph_builder.py:1214: UserWarning: In its current implementation, the
↳ "connector project match" method may take a while for large networks.
warnings.warn()
```

### Saving and reloading

Lets save all graphs to the `public_transport.sqlite` database.

```
data.save_graphs()
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/transit/
↳ transit.py:91: UserWarning: Currently only a single transit graph can be saved and
↳ reloaded. Multiple graph support is plan for a future release.
warnings.warn(
```

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()
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/transit/
↳ transit.py:105: UserWarning: Currently only a single transit graph can be saved and
↳ reloaded. Multiple graph support is plan for a future release. `period_ids` argument
↳ is currently ignored.
warnings.warn(
```

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*.

```
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()
```

## Spiess & Florian assignment

### Mock demand matrix

We'll create a mock demand matrix with demand  $I$  for every zone. We'll also need to convert from *zone\_id*'s to *node\_id*'s.

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

```
zones_in_the_model = len(transit_graph.centroids)

names_list = ['pt']

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

### Hyperpath generation/assignment

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

```
assig = TransitAssignment()

# Create the assignment class
assigclass = TransitClass(name="pt", graph=transit_graph, matrix=mat)
assig.add_class(assigclass)

# We need to tell AequilibraE 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.
assigclass.set_demand_matrix_core("pt")
```

Let's perform the assignment with the mock demand matrix for all *TransitClass*'s added.

```
assig.execute()
```

View the results

```
assig.results()
```

We can also access the *TransitAssignmentResults* object from the *TransitClass*

```
assigclass.results
```

```
<aequilibrae.paths.results.assignment_results.TransitAssignmentResults object at 0x7fa8d0d5dd50>
```

### 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()
```

**Total running time of the script:** (0 minutes 11.919 seconds)

### Route Choice set generation

In this example, we show how to generate route choice sets for estimation of route choice models, using a city in La Serena Metropolitan Area in Chile.

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

We create the example project inside our temp folder

```
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "coquimbo")
```

### Choice set generation

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

Let's build all graphs

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



```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
```

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))

# We allow flows through "centroid connectors" because our centroids are not really.
↪centroids
# If we have actual centroid connectors in the network (and more than one per centroid) ,
↪ then we
# should remove them from the graph
graph.set_blocked_centroid_flows(False)
```

## Route Choice class

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

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

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)
```

Here we'll set the parameters of our set generation. There are two algorithms available: Link penalisation, and BFSLE based on the paper "Route choice sets for very high-resolution data" by Nadine Rieser-Schüssler, Michael Balmer & Kay W. Axhausen (2013). <https://doi.org/10.1080/18128602.2012.671383>

Our BFSLE implementation has been extended to allow applying link penalisation as well. 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. Pass set *penalty=1.0* to disable.

It is highly recommended to set either *max\_routes* or *max\_depth* to prevent runaway results.

```
# rc.set_choice_set_generation("link-penalisation", max_routes=5, penalty=1.02)
```

The 5% penalty (1.05) is likely a little too large, but it create 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().to_pandas()
```

## Plotting choice sets

Now we will plot the paths we just created for the second OD pair

```
import folium
import geopandas as gpd
```

```
# Let's create a separate for each route so we can visualize one at a time
rlyr1 = folium.FeatureGroup("route 1")
rlyr2 = folium.FeatureGroup("route 2")
rlyr3 = folium.FeatureGroup("route 3")
rlyr4 = folium.FeatureGroup("route 4")
rlyr5 = folium.FeatureGroup("route 5")
od_lyr = folium.FeatureGroup("Origin and Destination")
layers = [rlyr1, rlyr2, rlyr3, rlyr4, rlyr5]
```

```
# We get the data we will use for the plot: Links, Nodes and the route choice set
links = gpd.GeoDataFrame(project.network.links.data, crs=4326)
nodes = gpd.GeoDataFrame(project.network.nodes.data, crs=4326)

plot_routes = choice_set[(choice_set["origin id"] == 77011)]["route set"].values

# Let's create the layers
colors = ["red", "blue", "green", "purple", "orange"]
for i, route in enumerate(plot_routes):
    rt = links[links.link_id.isin(route)]
    routes_layer = layers[i]
    for wkt in rt.geometry.to_wkt().values:
        points = wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "").split(
            "\n", "\n")
        points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
        # we need to take from x/y to lat/long
        points = [[x[1], x[0]] for x in eval(points)]

        _ = folium.vector_layers.PolyLine(points, color=colors[i], weight=4).add_
            to(routes_layer)

# Creates the points for both origin and destination
for i, row in nodes[nodes.node_id.isin((77011, 74089))].iterrows():
    point = (row.geometry.y, row.geometry.x)

    _ = folium.vector_layers.CircleMarker(
        point,
        popup=f"<b>link_id: {row.node_id}</b>",
```

(continues on next page)

(continued from previous page)

```

        color="red",
        radius=5,
        fill=True,
        fillColor="red",
        fillOpacity=1.0,
    ).add_to(od_lyr)

```

It is worthwhile to notice that using distance as the cost function, the routes are not the fastest ones as the freeway does not get used

Create the map and center it in the correct place

```

long, lat = project.conn.execute("select avg(xmin), avg(ymin) from idx_links_geometry").
    ↪fetchone()

map_osm = folium.Map(location=[lat, long], tiles="Cartodb Positron", zoom_start=12)
for routes_layer in layers:
    routes_layer.add_to(map_osm)
od_lyr.add_to(map_osm)
folium.LayerControl().add_to(map_osm)
map_osm

```

```
project.close()
```

**Total running time of the script:** (0 minutes 4.941 seconds)

## 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.

```

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

```

```

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

project = create_example(fldr, "coquimbo")

```

```

import logging
import sys

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

```

## Route Choice

```
import numpy as np
```

### 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  $\theta$  \* distance And the path overlap factor (PSL) is equal to  $\beta$ .

```
# Distance factor
theta = 0.00011

# PSL parameter
beta = 1.1
```

Let's build all graphs

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
  df = pd.read_sql(sql, conn).fillna(value=np.nan)
2024-08-19 06:17:25,676;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:25,759;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:25,857;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:25,958;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

We grab the graph for cars

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

We also see what graphs are available

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

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

let's say that utility is just a function of distance So we build our *utility* field as the distance times 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))
```

```
2024-08-19 06:17:26,036;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
has(ve) at least one NaN value. Check your computations
```

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

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

We allow flows through “centroid connectors” because our centroids are not really centroids. If we have actual centroid connectors in the network (and more than one per centroid), then we should remove them from the graph

```
graph.set_blocked_centroid_flows(False)
```

## Mock demand matrix

We’ll create a mock demand matrix with demand 1 for every zone.

```
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()
```

## 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. 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’ll set the parameters of our set generation. There are two algorithms available: Link penalisation, or BFSLE based on the paper “Route choice sets for very high-resolution data” by Nadine Rieser-Schüssler, Michael Balmer & Kay W. Axhausen (2013). <https://doi.org/10.1080/18128602.2012.671383>

Our BFSLE implementation is slightly different and has extended to allow applying link penalisation as well. 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. Pass set *penalty=1.0* to disable.

To assist in filtering out bad results during the assignment, a *cutoff\_prob* parameter 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. A higher *cutoff\_prob* includes more routes. A value of 0.0 will only include the minimum cost route. A value of 1.0 includes all routes.

It is highly recommended to set either *max\_routes* or *max\_depth* to prevent runaway results.

```
rc.set_choice_set_generation("link-penalisation", max_routes=5, penalty=1.02)
```

```
rc.set_choice_set_generation("bfsle", max_routes=5)
```

All parameters are optional, the defaults are:

```
print(rc.default_parameters)
```

```
{'generic': {'seed': 0, 'max_routes': 0, 'max_depth': 0, 'max_misses': 100, 'penalty': 1.0, 'cutoff_prob': 0.0, 'beta': 1.0, 'store_results': True}, 'link-penalisation': {}, 'bfsle': {'penalty': 1.0}}
```

We can now perform a computation for single OD pair if we'd like. Here we do one between the first and last centroid as well as an assignment.

```
results = rc.execute_single(77011, 74089, demand=1.0)
print(results[0])
```

```
(24222, 30332, 30333, 10435, 30068, 30069, 14198, 14199, 31161, 30928, 30929, 30930,
→ 30931, 24172, 30878, 30879, 30880, 30881, 30882, 30883, 30884, 30885, 30886, 30887,
→ 30888, 30889, 30890, 30891, 5179, 5180, 5181, 5182, 26463, 26462, 26461, 26460, 26459,
→ 26458, 26457, 26456, 26480, 3341, 3342, 3339, 9509, 9510, 9511, 9512, 18487, 14972,
→ 14973, 32692, 32693, 32694, 2300, 2301, 33715, 19978, 19979, 19977, 19976, 19975,
→ 19974, 19973, 19972, 19971, 19970, 22082, 22080, 5351, 5352, 2280, 2281, 2282, 575,
→ 576, 577, 578, 579, 536, 537, 538, 539, 540, 541, 15406, 15407, 15408, 553, 552, 633,
→ 634, 635, 630, 631, 632, 623, 624, 625, 626, 471, 5363, 34169, 34170, 34171, 34785,
→ 6466, 6465, 29938, 29939, 29940, 29941, 1446, 1447, 1448, 1449, 1450, 939, 940, 941,
→ 9840, 9841, 26314, 26313, 26312, 26311, 26310, 26309, 26308, 26307, 26306, 26305,
→ 26304, 26303, 26302, 26301, 26300, 34079, 34147, 29962, 26422, 26421, 26420, 765, 764,
→ 763, 762, 761, 760, 736, 10973, 10974, 10975, 725, 10972, 727, 728, 26424, 733, 734,
→ 29899, 20970, 20969, 20968, 20967, 20966, 20965, 20964, 20963, 20962, 9584, 9583,
→ 20981, 21398, 20982, 34208, 35, 36, 59, 60, 61, 22363, 22364, 22365, 22366, 22367,
→ 28958, 28959, 28960, 28961, 28962, 28805, 28806, 28807, 28808, 28809, 28810, 28827,
→ 28828, 28829, 28830, 28874)
```

Because we asked it to also perform an assignment we can access the various results from that. The default return is a Pyarrow Table but Pandas is nicer for viewing.

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

let's define a function to plot assignment results

```

def plot_results(link_loads):
    import folium
    import geopandas as gpd

    link_loads = link_loads[link_loads.tot > 0]
    max_load = link_loads["tot"].max()
    links = gpd.GeoDataFrame(project.network.links.data, crs=4326)
    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

    # Let's create the layers
    for _, rec in loaded_links.iterrows():
        points = rec.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "")
        points = points.split(", ")
        points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
        # we need to take from x/y to lat/long
        points = [[x[1], x[0]] for x in eval(points)]
        _ = folium.vector_layers.PolyLine(
            points,
            tooltip=f"link_id: {rec.link_id}, Flow: {rec.tot:.3f}",
            color="red",
            weight=factor * rec.tot,
        ).add_to(loads_lyr)
    long, lat = project.conn.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
    map_osm = folium.Map(location=[lat, long], tiles="Cartodb Positron", zoom_start=12)
    loads_lyr.add_to(map_osm)
    folium.LayerControl().add_to(map_osm)
    return map_osm

```

```
plot_results(rc.get_load_results()["demand"])
```

To perform a batch operation we need to prepare the object first. We can either provide a list of tuple of the OD pairs we'd like to use, or we can provided a 1D list and the generation will be run on all permutations. `rc.prepare(graph.centroids[:5])`

```
rc.prepare()
```

Now we can perform a batch computation with an assignment

```

rc.execute(perform_assignment=True)
res = rc.get_results().to_pandas()
res.head()

```

Since we provided a matrix initially we can also perform link loading based on our assignment results.

```
rc.get_load_results()
```

```
plot_results(rc.get_load_results()["demand"])
```

## 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 “s11” and “s12” when (7369, 1) is utilised.

```
rc.set_select_links({"s11": [(7369, 1), (20983, 1)], "s12": [(7369, 1)]})
rc.execute(perform_assignment=True)
```

We can get then the results in a Pandas data frame 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. These matrices are constructed during `get_select_link_loading_results`.

```
rc.get_select_link_od_matrix_results()
```

```
{'s11': {'demand': <aequilibrae.matrix.sparse_matrix.C00 object at 0x7fa8dddc2c20>, '5x_
↳ demand': <aequilibrae.matrix.sparse_matrix.C00 object at 0x7fa8dddc3340>}, 's12': {
↳ 'demand': <aequilibrae.matrix.sparse_matrix.C00 object at 0x7fa8dddc1f60>, '5x demand
↳ ': <aequilibrae.matrix.sparse_matrix.C00 object at 0x7fa8dddc1600>}}
```

```
od_matrix = rc.get_select_link_od_matrix_results()["s11"]["demand"]
od_matrix.to_scipy().toarray()
```

```
array([[0.      , 0.      , 0.      , 3.04610785],
       [0.      , 0.      , 0.      , 0.      ],
       [0.      , 0.      , 0.      , 0.      ],
       [0.      , 0.      , 0.      , 0.      ]])
```

```
project.close()
```

```
INFO:aequilibrae:Closed project on /tmp/02423b8fe2eb473c9b82167cde97592f
```

**Total running time of the script:** (0 minutes 5.835 seconds)

## Forecasting

In this example, we present a full forecasting workflow for the Sioux Falls example model.

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

(continues on next page)



(continued from previous page)

```
import logging
import sys
```

We create the example project inside our temp folder

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

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

## Traffic assignment with skimming

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

We build all graphs

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
```

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)
```

We get the demand matrix directly from the project record. So let's inspect what we have in the project

```
proj_matrices = project.matrices
print(proj_matrices.list())
```

	name	file_name	...	description	status
0	demand_omx	demand.omx	...	Original data imported to OMX format	
1	demand_mc	demand_mc.omx	...		None

(continues on next page)

(continued from previous page)

```

2      skims      skims.omx ...      Example skim
3 demand_aem      demand.aem ... Original data imported to AEM format

[4 rows x 8 columns]
```

Let's get it in this better way

```

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

```

assig = TrafficAssignment()

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

# The first thing to do is to add at list of traffic classes to be assigned
assig.add_class(assigclass)

# We set these parameters only after adding one class to the assignment
assig.set_vdf("BPR") # This is not case-sensitive

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

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

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

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

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

Convergence report is easy to see

```
import pandas as pd
```

```

convergence_report = assig.report()
print(convergence_report.head())
```

	iteration	rgap	alpha	warnings	beta0	beta1	beta2
0	1	inf	1.000000		1.000000	0.000000	0.0
1	2	0.855075	0.328400		1.000000	0.000000	0.0
2	3	0.476346	0.186602		1.000000	0.000000	0.0
3	4	0.235513	0.241148		1.000000	0.000000	0.0
4	5	0.109241	0.818547		0.607382	0.392618	0.0

```
volumes = assig.results()
print(volumes.head())
```

```

      matrix_ab  matrix_ba  matrix_tot  ...      PCE_AB  PCE_BA      PCE_tot
link_id
1      4502.545113      NaN  4502.545113  ...  4502.545113      NaN  4502.545113
2      8222.240524      NaN  8222.240524  ...  8222.240524      NaN  8222.240524
3      4622.925028      NaN  4622.925028  ...  4622.925028      NaN  4622.925028
4      5897.692905      NaN  5897.692905  ...  5897.692905      NaN  5897.692905
5      8101.860609      NaN  8101.860609  ...  8101.860609      NaN  8101.860609

[5 rows x 18 columns]
```

We could export it to CSV or AequilibraE data, 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

### Calibration

We will calibrate synthetic gravity models using the skims for TIME that we just generated

```
import numpy as np
from aequilibrae.distribution import GravityCalibration
```

Let's take another look at what we have in terms of matrices in the model

```
print(proj_matrices.list())
```

```

      name  ... status
0      demand_omx  ...
1      demand_mc  ...
2      skims  ...
3      demand_aem  ...
4  base_year_assignment_skims_car  ...

[5 rows x 8 columns]
```

We need the demand

```
demand = proj_matrices.get_matrix("demand_aem")
```

And the skims

```
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
```

```
['distance_blended', 'distance_final', 'free_flow_time_blended', 'free_flow_time_final']
```

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 we give a new name to save it

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

This also updates these new matrices as those being used for computation as one can verify below

```
imped.view_names
```

```
['final_time_with_intrazonals']
```

We set the matrices for being used in computation

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

```
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"))

    # 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")
```

## 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
from aequilibrae.matrix import AequilibraeData
```

We compute the vectors from our matrix

```
origins = np.sum(demand.matrix_view, axis=1)
destinations = np.sum(demand.matrix_view, axis=0)

args = {
    "file_path": join(fldr, "synthetic_future_vector.aed"),
    "entries": demand.zones,
    "field_names": ["origins", "destinations"],
    "data_types": [np.float64, np.float64],
    "memory_mode": False,
}

vectors = AequilibraeData()
vectors.create_empty(**args)

vectors.index[:] = demand.index[:]

# Then grow them with some random growth between 0 and 10%, and balance them
vectors.origins[:] = origins * (1 + np.random.rand(vectors.entries) / 10)
vectors.destinations[:] = destinations * (1 + np.random.rand(vectors.entries) / 10)
vectors.destinations *= vectors.origins.sum() / vectors.destinations.sum()
```

## Impedance

```
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)
```

```
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,
        "rows": vectors,
        "row_field": "origins",
        "model": model,
        "columns": vectors,
```

(continues on next page)

(continued from previous page)

```

        "column_field": "destinations",
        "nan_as_zero": True,
    }

    gravity = GravityApplication(**args)
    gravity.apply()

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

```

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

```

proj_matrices.update_database()
print(proj_matrices.list())

```

```

           name  ... status
0          demand_omx  ...
1          demand_mc  ...
2           skims  ...
3      demand_aem  ...
4  base_year_assignment_skims_car  ...
5      demand_power_modeled  ...
6      demand_expo_modeled  ...

```

[7 rows x 8 columns]

### IPF for the future vectors

```

args = {
    "matrix": demand,
    "rows": vectors,
    "columns": vectors,
    "column_field": "destinations",
    "row_field": "origins",
    "nan_as_zero": True,
}

ipf = Ipfd(**args)
ipf.fit()

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")

```

```
<aequilibrae.project.data.matrix_record.MatrixRecord object at 0x7fa8dd5da950>
```

```
df = proj_matrices.list()
```

## Future traffic assignment

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

```
logger.info("\n\n\n TRAFFIC ASSIGNMENT FOR FUTURE YEAR")
```

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

# Let's see what is the core we ended up getting. It should be 'gravity'
demand.names
```

```
['matrix']
```

Let's use the IPF matrix

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

```
assig = TrafficAssignment()

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

# The first thing to do is to add at a list of traffic classes to be assigned
assig.add_class(assigclass)

assig.set_vdf("BPR") # This is not case-sensitive

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

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

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

# Since I haven't checked the parameters file, let's make sure convergence criteria is
↪ good
assig.max_iter = 500
assig.rgap_target = 0.00001
```

### Optional: Select link analysis

If we want to execute select link analysis on a particular TrafficClass, we set the links we are analyzing. The format of the input select links is a dictionary (str: list[tuple]). Each entry represents a separate set of selected links to compute. The str name will name the set of links. The list[tuple] is the list of links being selected, of the form (link\_id, direction), as it occurs in the Graph. Direction can be 0, 1, -1. 0 denotes bi-directionality. For example, let's use Select Link on two sets of links:

```
select_links = {
    "Leaving node 1": [(1, 1), (2, 1)],
    "Random nodes": [(3, 1), (5, 1)],
}
```

We call this command on the class we are analyzing with our dictionary of values

```
assignclass.set_select_links(select_links)

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳ traffic_class.py:167: UserWarning: Input string name has a space in it. Replacing with _
↳ _
warnings.warn("Input string name has a space in it. Replacing with _")
```

Now let us 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.

```
assign.save_select_link_results("select_link_analysis")
```

Say we just want to save our select link flows, we can call:

```
assign.save_select_link_flows("just_flows")

# Or if we just want the SL matrices:
assign.save_select_link_matrices("just_matrices")
# Internally, the save_select_link_results calls both of these methods at once.
```

We could export it to CSV or AequilibraE data, but let's put it directly into the results database

```
assign.save_results("future_year_assignment")
```

And save the skims

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



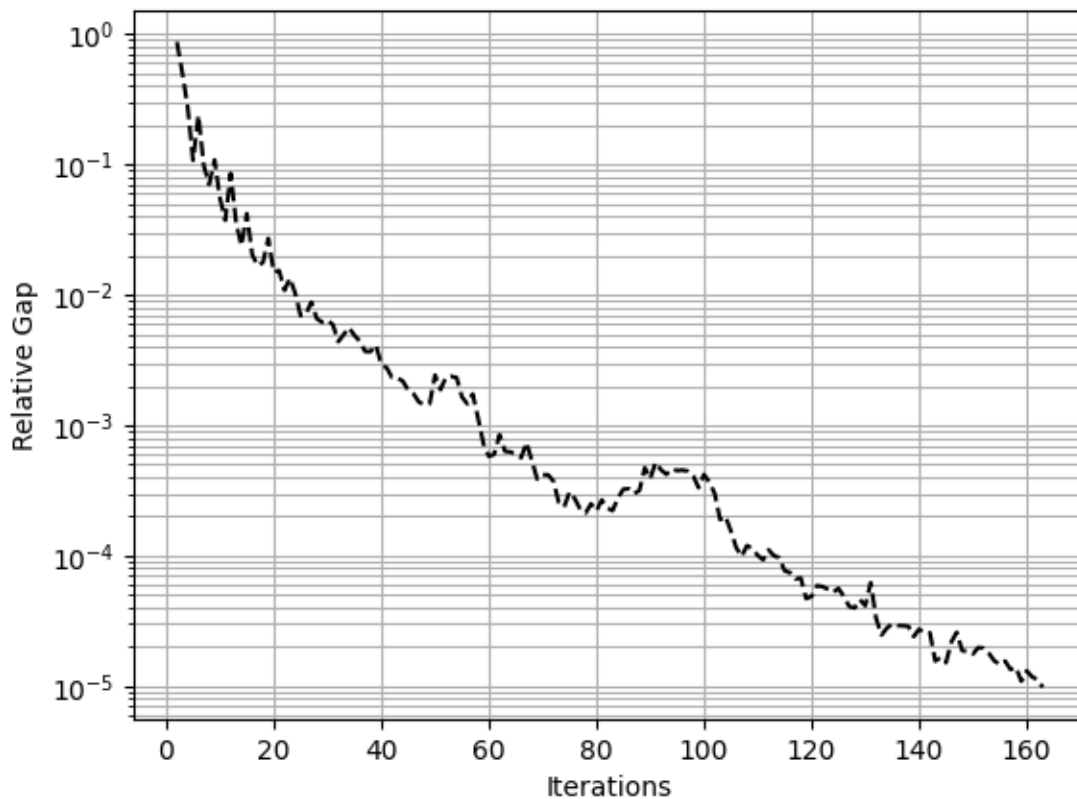
We can also plot convergence

```
import matplotlib.pyplot as plt
```

```
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(r"Iterations")
plt.ylabel(r"Relative Gap")
plt.show()
```



Close the project

```
project.close()
```

**Total running time of the script:** (0 minutes 3.465 seconds)

## 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.

Imports

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

import pandas as pd
import geopandas as gpd
import numpy as np
import folium

from aequilibrae.utils.create_example import create_example
```

We create the example project inside our temp folder

```
fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "coquimbo")
```

```
import logging
import sys

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

## Route Choice

### 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  $\theta$  \* distance And the path overlap factor (PSL) is equal to  $\beta$ .

```
# Distance factor
theta = 0.011

# PSL parameter
beta = 1.1
```

Let's build all graphs

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
    df = pd.read_sql(sql, conn).fillna(value=np.nan)
2024-08-19 06:17:35,300;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:35,386;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:35,486;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
2024-08-19 06:17:35,582;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

We grab the graph for cars

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

We also see what graphs are available

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

```
dict_keys(['b', 'c', 't', 'w'])
```

let's say that utility is just a function of distance So we build our *utility* field as the distance times 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)
```

```
2024-08-19 06:17:35,656;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↪has(ve) at least one NaN value. Check your computations
```

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

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

We allow flows through “centroid connectors” because our centroids are not really centroids If we have actual centroid connectors in the network (and more than one per centroid) , then we should remove them from the graph

```
graph.set_blocked_centroid_flows(False)
graph.graph.head()
```

## Mock demand matrix

We'll create a mock demand matrix with demand 10 for every zone.

```
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 our sub-area analysis, here we'll use a section of zones and create our 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 = gpd.GeoDataFrame(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 our 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.
from aequilibrae.paths import SubAreaAnalysis

subarea = SubAreaAnalysis(graph, zones, mat)
subarea.rc.set_choice_set_generation("lp", max_routes=5, penalty=1.02, store_
↳ results=False)
subarea.rc.execute(perform_assignment=True)
```

(continues on next page)

(continued from previous page)

```
demand = subarea.post_process()
demand
```

```
2024-08-19 06:17:36,419;INFO ; Created: 650 edge pairs from 26 edges
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳ route_choice.py:451: UserWarning: Two input links map to the same compressed link in
↳ the network, removing superfluous link 31425 and direction -1 with compressed id 9483
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳ route_choice.py:451: UserWarning: Two input links map to the same compressed link in
↳ the network, removing superfluous link 31425 and direction 1 with compressed id 14421
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳ route_choice.py:451: UserWarning: Two input links map to the same compressed link in
↳ the network, removing superfluous link 21724 and direction 1 with compressed id 14421
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳ route_choice.py:451: UserWarning: Two input links map to the same compressed link in
↳ the network, removing superfluous link 21724 and direction -1 with compressed id 9483
warnings.warn(
```

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
```

```
2024-08-19 06:17:58,250;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳ has(ve) at least one NaN value. Check your computations

array([ 29,  30,  31,  32,  33,  34,  37,  38,  39,
        40,  49,  50,  51,  52,  57,  58,  59,  60,
       61044, 67891, 68671, 72081, 72092, 72096, 72134, 72161, 73381,
       73394, 73432, 73506, 73541, 73565, 73589, 75548, 77285, 77287,
       77289, 79297, 79892])
```

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

```
from aequilibrae.paths import RouteChoice

rc = RouteChoice(graph)
rc.add_demand(demand)
rc.set_choice_set_generation("link-penalisation", max_routes=5, penalty=1.02, store_
↳ results=False, seed=123)
rc.execute(perform_assignment=True)
```

And plot the link loads for easy viewing

```
subarea_zone = folium.Polygon(
    locations=[(x[1], x[0]) for x in zones.unary_union.boundary.coords],
```

(continues on next page)

(continued from previous page)

```

        fill_color="blue",
        fill_opacity=0.5,
        fill=True,
        stroke=False,
    )

def plot_results(link_loads):
    link_loads = link_loads[link_loads.tot > 0]
    max_load = link_loads["tot"].max()
    links = gpd.GeoDataFrame(project.network.links.data, crs=4326)
    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

    # Let's create the layers
    for _, rec in loaded_links.iterrows():
        points = rec.geometry.wkt.replace("LINESTRING ", "").replace("(", "").replace(")", "")
        ↪ points = points.split(", ")
        points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
        # we need to take from x/y to lat/long
        points = [[x[1], x[0]] for x in eval(points)]
        _ = folium.vector_layers.PolyLine(
            points,
            tooltip=f"link_id: {rec.link_id}, Flow: {rec.tot:.3f}",
            color="red",
            weight=factor * rec.tot,
        ).add_to(loads_lyr)
    long, lat = project.conn.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
    ↪ ).fetchone()

    map_osm = folium.Map(location=[lat, long], tiles="Cartodb Positron", zoom_start=12)
    loads_lyr.add_to(map_osm)
    folium.LayerControl().add_to(map_osm)
    return map_osm

map = plot_results(rc.get_load_results()["demand"])
subarea_zone.add_to(map)
map

```

```

/home/runner/work/aequilibrae/aequilibrae/docs/source/examples/assignment_workflows/plot_
↪ subarea_analysis.py:161: DeprecationWarning: The 'unary_union' attribute is deprecated,
↪ use the 'union_all()' method instead.
    locations=[[x[1], x[0]] for x in zones.unary_union.boundary.coords],

```

## Manual sub-area analysis further preparation

%% We take the union of this GeoDataFrame as our polygon.

```
poly = zones.unary_union
poly
```

```
/home/runner/work/aequilibrae/aequilibrae/docs/source/examples/assignment_workflows/plot_
→ subarea_analysis.py:209: DeprecationWarning: The 'unary_union' attribute is deprecated,
→ use the 'union_all()' method instead.
    poly = zones.unary_union

<POLYGON ((-71.348 -29.993, -71.349 -29.993, -71.35 -29.992, -71.349 -29.991...>
```

It's useful later on to know which links from the network cross our polygon.

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

As well as which nodes are interior.

```
nodes = gpd.GeoDataFrame(project.network.nodes.data).set_index("node_id")
inside_nodes = nodes.sjoin(zones, how="inner").sort_index()
inside_nodes.head()
```

Here we 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()
```

## Sub-area visualisation

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.

```
points = [(link_id, list(x.coords)) for link_id, x in zip(inner_links.link_id, inner_
→ links.geometry)]
subarea_layer = folium.FeatureGroup("Cut links")

for link_id, line in points:
    _ = folium.vector_layers.PolyLine(
        [(x[1], x[0]) for x in line],
        tooltip=f"link_id: {link_id}",
        color="red",
    ).add_to(subarea_layer)
```

(continues on next page)

(continued from previous page)

```

long, lat = project.conn.execute("select avg(xmin), avg(ymin) from idx_links_geometry").
↳fetchone()

map_osm = folium.Map(location=[lat, long], tiles="Cartodb Positron", zoom_start=12)

subarea_zone.add_to(map_osm)

subarea_layer.add_to(map_osm)
_ = folium.LayerControl().add_to(map_osm)
map_osm

```

## Manual sub-area analysis

In order to perform our 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"

```

```
'Created: 650 edge pairs from 26 edges'
```

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

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

We'll re-prepare our graph quickly

```

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)

```

```

/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↳network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↳ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↳objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↳'future.no_silent_downcasting', True)`
  df = pd.read_sql(sql, conn).fillna(value=np.nan)
2024-08-19 06:18:05,302;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳has(ve) at least one NaN value. Check your computations
2024-08-19 06:18:05,384;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳has(ve) at least one NaN value. Check your computations
2024-08-19 06:18:05,484;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳has(ve) at least one NaN value. Check your computations
2024-08-19 06:18:05,580;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳has(ve) at least one NaN value. Check your computations

```

(continues on next page)



(continued from previous page)

```
2024-08-19 06:18:05,653;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳has(ve) at least one NaN value. Check your computations
```

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)
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳route_choice.py:451: UserWarning: Two input links map to the same compressed link in_
↳the network, removing superfluous link 31425 and direction -1 with compressed id 9483
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳route_choice.py:451: UserWarning: Two input links map to the same compressed link in_
↳the network, removing superfluous link 31425 and direction 1 with compressed id 14421
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳route_choice.py:451: UserWarning: Two input links map to the same compressed link in_
↳the network, removing superfluous link 21724 and direction 1 with compressed id 14421
warnings.warn(
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/paths/
↳route_choice.py:451: UserWarning: Two input links map to the same compressed link in_
↳the network, removing superfluous link 21724 and direction -1 with compressed id 9483
warnings.warn(
```

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("link-penalisation", max_routes=5, 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()["demand"])
subarea_zone.add_to(map)
map
```

We'll pull out just OD matrix results as well we need it for the post-processing, we'll also convert the sparse matrices to SciPy COO matrices.

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

For the post processing, we are interested in the demand of OD pairs that enter or exit the sub-area, or do both. For the single enters and exists we can extract that information from the single link select link results. We also need to map the links that cross the boundary to the origin/destination node and the node that appears on the outside of the sub-area.

```
from collections import defaultdict

entered = defaultdict(float)
exited = defaultdict(float)
for (link_id, dir), v in edge_totals.items():
    link = g.loc[link_id, dir]
    for (o, d), load in v.todok().items():
        o = graph.all_nodes[o]
        d = graph.all_nodes[d]

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

        if o_inside and not d_inside:
            exited[o, graph.all_nodes[link.b_node]] += load
        elif not o_inside and d_inside:
            entered[graph.all_nodes[link.a_node], d] += load
        elif not o_inside and not d_inside:
            pass
```

Here he have the load that entered the sub-area

```
entered
```

```
defaultdict(<class 'float'>, {(34, 37): 10.0, (20, 39): 9.88913345336914, (23, 39): 0.
↳ 1108664870262146})
```

and the load that exited the sub-area

```
exited
```

```
defaultdict(<class 'float'>, {(39, 20): 9.873265266418457, (37, 36): 0.07007550448179245,
↳ (39, 23): 0.12673552334308624, (37, 19): 9.929924011230469})
```

To find the load that both entered and exited we can look at the edge pair select link results.

```

through = defaultdict(float)
for (l1, l2), v in edge_pair_values.items():
    link1 = g.loc[l1]
    link2 = g.loc[l2]

    for (o, d), load in v.todok().items():
        o_inside = o in inside_nodes.index
        d_inside = d in inside_nodes.index

        if not o_inside and not d_inside:
            through[graph.all_nodes[link1.a_node], graph.all_nodes[link2.b_node]] += load

through

```

```

defaultdict(<class 'float'>, {(21, 23): 4.1274213790893555, (35, 22): 0.2188785821199417,
↳ (35, 25): 9.781121253967285, (22, 37): 0.435827374458313, (26, 36): 0.
↳ 2188785821199417, (23, 36): 5.337530136108398, (23, 38): 4.443591117858887, (25, 37):
↳ 9.564172744750977, (39, 26): 0.435827374458313, (39, 23): 5.436751365661621})

```

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

```

```

2024-08-19 06:18:11,807;WARNING ; Field(s) speed, travel_time, capacity, osm_id, lanes_
↳ has(ve) at least one NaN value. Check your computations

array([19, 20, 21, 22, 23, 25, 26, 34, 35, 36, 37, 38, 39])

```

Re-perform our assignment

```

rc = RouteChoice(graph)
rc.add_demand(demand)
rc.set_choice_set_generation("link-penalisation", max_routes=5, 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()["demand"])
subarea_zone.add_to(map)
map
```

**Total running time of the script:** (0 minutes 41.923 seconds)

## 1.7.7 Other Applications

### Logging to terminal

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

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

We create the example project inside our temp folder

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

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

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

```
project.close()
```

**Total running time of the script:** (0 minutes 0.117 seconds)

### Checking AequilibraE's log

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

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

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

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

(continues on next page)

(continued from previous page)

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

We create an empty project on an arbitrary folder

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

We build our graphs

```
project.network.build_graphs()

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

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↪network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↪ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↪objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↪'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
```

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)
```

```

1 - 2024-08-19 06:18:17,489;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

2 - 2024-08-19 06:18:17,517;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

3 - 2024-08-19 06:18:17,546;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

4 - 2024-08-19 06:18:17,574;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

5 - 2024-08-19 06:18:17,602;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

6 - 2024-08-19 06:18:17,631;WARNING ; Field(s) name, lanes has(ve) at least one NaN
↳value. Check your computations

7 - 2024-08-19 06:18:17,649;WARNING ; Cost field with wrong type. Converting to float64

8 - 2024-08-19 06:18:17,901;INFO ; Traffic Class specification

9 - 2024-08-19 06:18:17,902;INFO ; {'car': {'Graph': '{"Mode': 'c', 'Block through
↳centroids': False, 'Number of centroids': 24, 'Links': 76, 'Nodes': 24}", 'Matrix': "{
↳'Source': '/tmp/eeba37449b5242c09ede6af08cccc01e/matrices/demand.omx', 'Number of
↳centroids': 24, 'Matrix cores': ['matrix'], 'Matrix totals': {'matrix': 360600.0}}"}

10 - 2024-08-19 06:18:17,902;INFO ; Traffic Assignment specification

11 - 2024-08-19 06:18:17,902;INFO ; {'VDF parameters': {'alpha': 0.15, 'beta': 4.0},
↳'VDF function': 'bpr', 'Number of cores': 4, 'Capacity field': 'capacity', 'Time field
↳': 'free_flow_time', 'Algorithm': 'bfg', 'Maximum iterations': 250, 'Target RGAP': 0.
↳0001}

12 - 2024-08-19 06:18:17,904;WARNING ; Cost field with wrong type. Converting to float64

13 - 2024-08-19 06:18:17,904;INFO ; bfg Assignment STATS

14 - 2024-08-19 06:18:17,904;INFO ; Iteration, RelativeGap, stepsize

15 - 2024-08-19 06:18:17,912;INFO ; 1,inf,1.0

16 - 2024-08-19 06:18:17,917;INFO ; 2,0.8550751349428284,0.32839952448634563

17 - 2024-08-19 06:18:17,921;INFO ; 3,0.4763455007221067,0.18660240547488702

18 - 2024-08-19 06:18:17,926;INFO ; 4,0.2355126365951965,0.2411477440291793

19 - 2024-08-19 06:18:17,930;INFO ; 5,0.10924072010481088,0.8185470737942447

20 - 2024-08-19 06:18:17,935;INFO ; 6,0.1980945227617506,0.14054330572978305

21 - 2024-08-19 06:18:17,939;INFO ; 7,0.0668172221544687,0.36171152718899235

```

(continues on next page)

(continued from previous page)

22 - 2024-08-19 06:18:17,945;INFO ; 8,0.06792122267870576,0.9634685345644022  
23 - 2024-08-19 06:18:17,950;INFO ; 9,0.10705582933092841,0.13757153109677167  
24 - 2024-08-19 06:18:17,957;INFO ; 10,0.04038814432034621,0.16094034254279752  
25 - 2024-08-19 06:18:17,961;INFO ; 11,0.02795248113775691,0.3408928228700519  
26 - 2024-08-19 06:18:17,968;INFO ; 12,0.032699992065524604,0.5467680533028708  
27 - 2024-08-19 06:18:17,973;INFO ; 13,0.024040970172177347,0.13812236751253115  
28 - 2024-08-19 06:18:17,978;INFO ; 14,0.02145103090950847,0.1970528150890536  
29 - 2024-08-19 06:18:17,984;INFO ; 15,0.01711663825927409,0.339938165833639  
30 - 2024-08-19 06:18:17,993;INFO ; 16,0.01735082411129593,0.7287610532385608  
31 - 2024-08-19 06:18:17,998;INFO ; 17,0.021164705464372085,0.08183287977099543  
32 - 2024-08-19 06:18:18,004;INFO ; 18,0.012464530324249264,0.1511598580475933  
33 - 2024-08-19 06:18:18,009;INFO ; 19,0.012549789919850556,0.16834049481540092  
34 - 2024-08-19 06:18:18,017;INFO ; 20,0.01186071978971438,0.5399903522726657  
35 - 2024-08-19 06:18:18,025;INFO ; 21,0.012859165521051463,0.054966591996545584  
36 - 2024-08-19 06:18:18,030;INFO ; 22,0.007671197552803449,0.061255615573588974  
37 - 2024-08-19 06:18:18,037;INFO ; 23,0.0055291789072302415,0.07401911120606758  
38 - 2024-08-19 06:18:18,042;INFO ; 24,0.0054667973306647966,0.191709779243585  
39 - 2024-08-19 06:18:18,046;INFO ; 25,0.007073668823306543,0.4228720696283197  
40 - 2024-08-19 06:18:18,053;INFO ; 26,0.009664731222551466,0.9410177051614603  
41 - 2024-08-19 06:18:18,058;INFO ; 27,0.008756083467130159,0.0517261106187546  
42 - 2024-08-19 06:18:18,062;INFO ; 28,0.005105221228053528,0.06397929882334243  
43 - 2024-08-19 06:18:18,073;INFO ; 29,0.0035319062476952545,0.05059090498821875  
44 - 2024-08-19 06:18:18,081;INFO ; 30,0.0031482926233624984,0.058437487817954215  
45 - 2024-08-19 06:18:18,086;INFO ; 31,0.003063209044595543,0.09173138967981778  
46 - 2024-08-19 06:18:18,090;INFO ; 32,0.0026646507707733915,0.07094979246385001  
47 - 2024-08-19 06:18:18,097;INFO ; 33,0.002302802037873952,0.1241286415196512

(continues on next page)

(continued from previous page)

```

48 - 2024-08-19 06:18:18,101;INFO ; 34,0.0027510302560630273,0.12799355702549403
49 - 2024-08-19 06:18:18,106;INFO ; 35,0.002125634778303211,0.1662038793394487
50 - 2024-08-19 06:18:18,113;INFO ; 36,0.002099491223200739,0.10282963642091511
51 - 2024-08-19 06:18:18,121;INFO ; 37,0.0014407763657242768,0.1449210133686946
52 - 2024-08-19 06:18:18,128;INFO ; 38,0.001418044704398517,0.06529689866671036
53 - 2024-08-19 06:18:18,132;INFO ; 39,0.0009714813735968882,0.09399257335234756
54 - 2024-08-19 06:18:18,133;INFO ; bfw Assignment finished. 39 iterations and 0.
    ↪ 0009714813735968882 final gap

```

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 <https://oldschool.runescape.wiki/index.php?curid=66905#>

**Total running time of the script:** (0 minutes 0.751 seconds)



## 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>

```
# Imports
from uuid import uuid4
import os
from tempfile import gettempdir
from aequilibrae.utils.create_example import create_example
import pandas as pd
import folium
```

We load the example project inside a temp folder

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

project = create_example(fldr)
```

We export the network to csv files in GMNS format, that will be saved inside the project folder

```
output_fldr = os.path.join(gettempdir(), uuid4().hex)
if not os.path.exists(output_fldr):
    os.mkdir(output_fldr)

project.network.export_to_gmns(path=output_fldr)
```

Now, let's plot a map. This map can be compared with the images of the README.md file located in this example repository on GitHub: [https://github.com/zephyr-data-specs/GMNS/blob/develop/examples/Arlington\\_Signals/README.md](https://github.com/zephyr-data-specs/GMNS/blob/develop/examples/Arlington_Signals/README.md)

```
links = pd.read_csv(os.path.join(output_fldr, "link.csv"))
nodes = pd.read_csv(os.path.join(output_fldr, "node.csv"))
```

```
# We create our Folium layers
network_links = folium.FeatureGroup("links")
network_nodes = folium.FeatureGroup("nodes")
layers = [network_links, network_nodes]

# We do some Python magic to transform this dataset into the format required by Folium
# We are only getting link_id and link_type into the map, but we could get other pieces
# of info as well
for i, row in links.iterrows():
    points = row.geometry.replace("LINESTRING ", "").replace("(", "").replace(")", "").
    ↪split(", ")
    points = "[" + ",".join([p.replace(" ", ", ") for p in points]) + "]"
    # we need to take from x/y to lat/long
    points = [[x[1], x[0]] for x in eval(points)]

    _ = folium.vector_layers.PolyLine(
        points, popup=f"<b>link_id: {row.link_id}</b>", tooltip=f"{row.facility_type}",
        ↪color="black", weight=2
    ).add_to(network_links)
```

(continues on next page)

(continued from previous page)

```

# And now we get the nodes
for i, row in nodes.iterrows():
    point = (row.y_coord, row.x_coord)

    _ = folium.vector_layers.CircleMarker(
        point,
        popup=f"<b>link_id: {row.node_id}</b>",
        tooltip=f"{row.node_type}",
        color="red",
        radius=5,
        fill=True,
        fillColor="red",
        fillOpacity=1.0,
    ).add_to(network_nodes)

# We get the center of the region
curr = project.conn.cursor()
curr.execute("select avg(xmin), avg(ymin) from idx_links_geometry")
long, lat = curr.fetchone()

# We create the map
map_gmns = folium.Map(location=[lat, long], zoom_start=12)

# add all layers
for layer in layers:
    layer.add_to(map_gmns)

# And Add layer control before we display it
folium.LayerControl().add_to(map_gmns)
map_gmns

project.close()

```

**Total running time of the script:** (0 minutes 0.329 seconds)

## 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.

```

# Imports
from uuid import uuid4
from tempfile import gettempdir
from os.path import join
from datetime import datetime
import pandas as pd
import numpy as np
from aequilibrae.utils.create_example import create_example
from aequilibrae.paths.results import PathResults

```

We create an empty project on an arbitrary folder

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

# Let's use the Nauru example project for display
project = create_example(fldr, "nauru")

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

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

```
# We get an arbitrary node to set as centroid and allow for the construction of graphs
centroid_count = project.conn.execute("select count(*) from nodes where is_centroid=1").
↳fetchone()[0]

if centroid_count == 0:
    arbitrary_node = project.conn.execute("select node_id from nodes limit 1").
    ↳fetchone()[0]
    nodes = project.network.nodes
    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()
```

```
/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/project/
↳network/network.py:327: FutureWarning: Downcasting object dtype arrays on .fillna, .
↳ffill, .bfill is deprecated and will change in a future version. Call result.infer_
↳objects(copy=False) instead. To opt-in to the future behavior, set `pd.set_option(
↳'future.no_silent_downcasting', True)`
df = pd.read_sql(sql, conn).fillna(value=np.nan)
```

We set the graph for computation

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

Get the nodes that are part of the car network

```
missing_nodes = [
    x[0] for x in project.conn.execute(f"Select node_id from nodes where instr(modes, '
↳{mode}')").fetchall()
]
missing_nodes = np.array(missing_nodes)
```

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")
```

```
06:18:18 - Computing island: 0
    Nodes to find: 2
06:18:18 - Computing island: 1
    Nodes to find: 0
```

```
We found 2 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()
```

**Total running time of the script:** (0 minutes 0.170 seconds)

## MODELING WITH AEQUILIBRAE

AequilibraE is the first fully-featured Python package for transportation modeling, and it aims to provide all the resources not easily available from other open-source packages in the Python (NumPy, really) ecosystem.

AequilibraE has also a fully features interface available as a plugin for the open source software QGIS, which is separately maintained and discussed in detail its [documentation](#).

Contributions are welcome to the existing modules and/or in the form of new modules.

In this section you can find a deep dive into modeling with AequilibraE, from a start guide to a complete view into AequilibraE's data structure.

### 2.1 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.

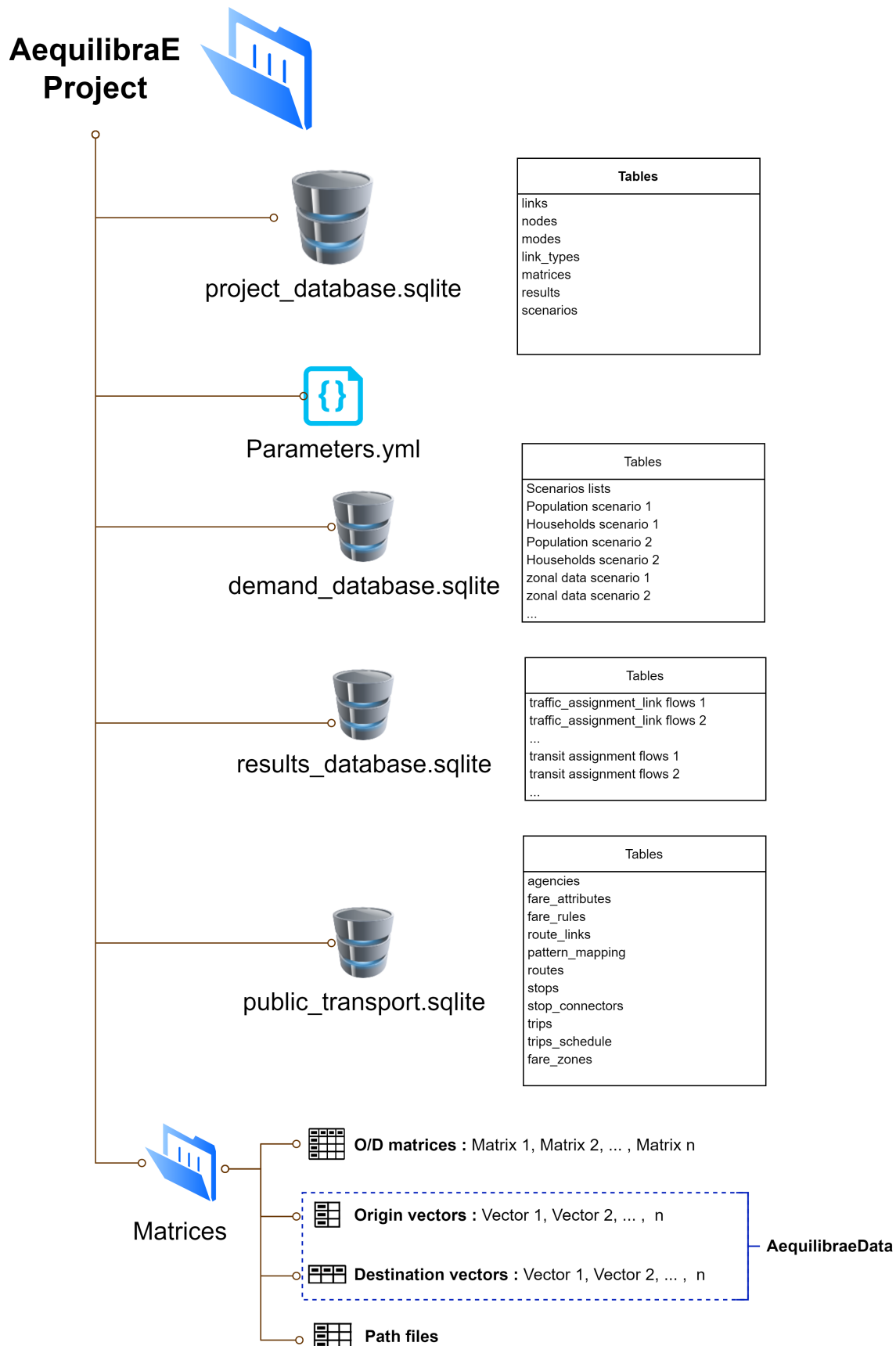
AequilibraE is built on the shoulder of much older and more established projects, such as [SQLite](#), [Spatialite](#) and [NumPy](#), as well as reasonably new industry standards such as the [Open-Matrix format](#).

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.

#### 2.1.1 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:



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 Open-Street Maps 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.

### Package components: A conceptual view

As all the components of an AequilibraE model 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 Parameters YAML File

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

- *Assignment*
- *Distribution*
- *Network*
- *System*

- *Open Street Maps*

### 2.2.1 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.

```
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*.

### 2.2.2 Distribution

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

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

### 2.2.3 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 be 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 [Open Street Maps](#) 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.

### Open Street Maps

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 [Open-Street Maps](#), 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 are 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.

### 2.2.4 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.

## 2.2.5 Open Street 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<sup>2</sup>) 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.

## 2.3 Project database

More details on the **project\_database.sqlite** are discussed on a nearly *per-table* basis below, and we recommend understanding the role of each table before setting an AequilibraE model you intend to use in anger.

### 2.3.1 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 but 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.

The screenshot shows the 'DB Browser for SQLite' application window. The title bar indicates the file path: 'C:\Users\pcamargo\Downloads\Freetown\project\_database.sqlite'. The menu bar includes 'File', 'Edit', 'View', 'Tools', and 'Help'. The toolbar contains icons for 'New Database', 'Open Database', 'Write Changes', 'Open Project', and 'Attach Database'. Below the toolbar, there are tabs for 'Database Structure', 'Browse Data', 'Edit Pragmas', and 'Execute SQL'. The 'Browse Data' tab is active, showing a table named 'about'. The table has two columns: 'infoname' and 'infovalue'. The data is displayed in a grid with 15 rows. The first row is a header with 'Filter' in both columns. The subsequent rows contain project metadata. At the bottom, there are navigation controls showing '1 - 15 of 15' and a 'Go to:' field with the value '1'. The status bar at the bottom right indicates 'UTF-8' encoding.

	infoname	infovalue
	Filter	Filter
1	model_name	Western Area, Sierra Leone
2	region	Freetown metropolitan area, Sierra Leone
3	description	
4	author	Outer Loop Consulting
5	year	2023
6	scenario_description	Base case. Full OSM network
7	model_version	NULL
8	project_id	NULL
9	aequilibrae_version	0.8.3
10	projection	4326
11	driving_side	right
12	license	OSM network. Refer to their license
13	scenario_name	base_case
14	country_name	Sierra Leone
15	country_code	SLE

An API for editing the contents of this database is available from the [API documentation](#).

### 2.3.2 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.**

#### Note

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

### 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 pages:

- *Importing files in GMNS format*
- *Importing from OpenStreetMaps*
- *Importing from link layers*
- *Exporting AequilibraE model to GMNS format*

### 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".

## 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 in *Change behavior*.

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

### Note

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

## 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

## Change behavior

In this section we present the mapping of 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
<i>Creating a node</i>	<i>Deleting a link</i>
<i>Deleting a node</i>	<i>Moving a link extremity</i>
<i>Moving a node</i>	<i>Re-shaping a link</i>
<i>Adding a data field</i>	<i>Deleting a required field</i>
<i>Deleting a data field</i>	
<i>Modifying a data entry</i>	

### 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 regarding modes and link types is discussed in their respective table descriptions.

### **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.

### 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.

### Deleting a required field

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

### Field-specific data consistency

Some data fields are specially sensitive to user changes.

#### Link distance

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

#### Link direction

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

#### *modes* field (Links and Nodes layers)

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

#### *link\_type* field (Links layer) & *link\_types* field (Nodes layer)

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

#### *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

### 2.3.3 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.

#### Note

**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.

	mode_name	mode_id	description	vot	pce
1	car	c	Passenger vehicles	0.04	1
2	motorcycle	M	Motorcycles	0.002	0.2
3	Trucks	T	All trucks	0.4	2.5

### Consistency triggers

As it happens with the links and nodes table (*Network consistency behaviour*), 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.

### 2.3.4 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.

DB Browser for SQLite - C:\Users\pcamargo\Downloads\Freetown\project\_database.sqlite

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes Open Project Save Project Attach Database

Database Structure Browse Data Edit Pragas Execute SQL

Table: link\_types

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

1 - 14 of 14 Go to: 1

UTF-8

**Note**

**Both link\_type and link\_type\_id MUST be unique**

## Consistency triggers

As it happens with the links and nodes tables, (*Network consistency behaviour*), 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 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

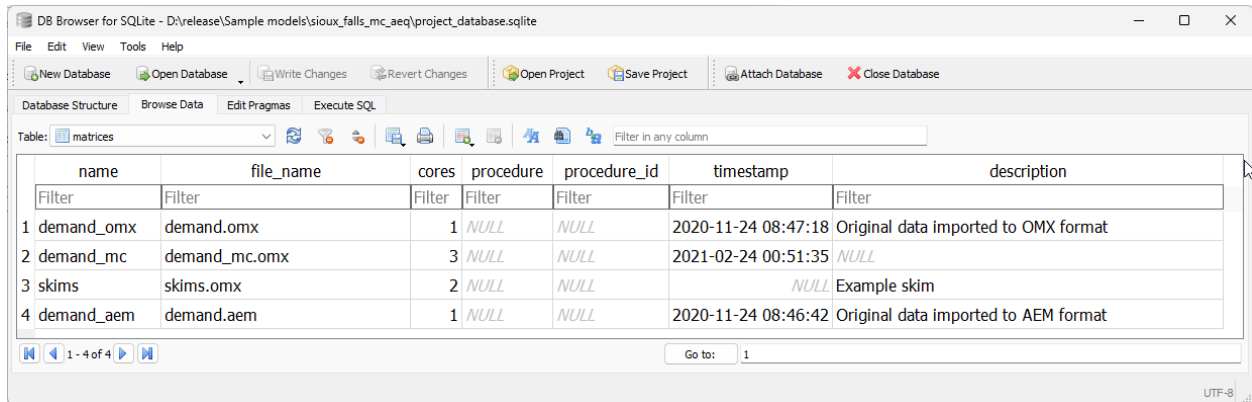
In this case, only the first behaviour mentioned above on *Editing a link\_type in the link\_types table* applies, the verification that the *link\_type\_id* is exactly one character long. Therefore only one new trigger is required.

## 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.

## 2.3.5 Matrices

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.



	name	file_name	cores	procedure	procedure_id	timestamp	description
1	demand_omx	demand.omx	1	NULL	NULL	2020-11-24 08:47:18	Original data imported to OMX format
2	demand_mc	demand_mc.omx	3	NULL	NULL	2021-02-24 00:51:35	NULL
3	skims	skims.omx	2	NULL	NULL	NULL	Example skim
4	demand_aem	demand.aem	1	NULL	NULL	2020-11-24 08:46:42	Original data imported to AEM format

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

## 2.3.6 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 .

## 2.3.7 Parameters metadata table

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.

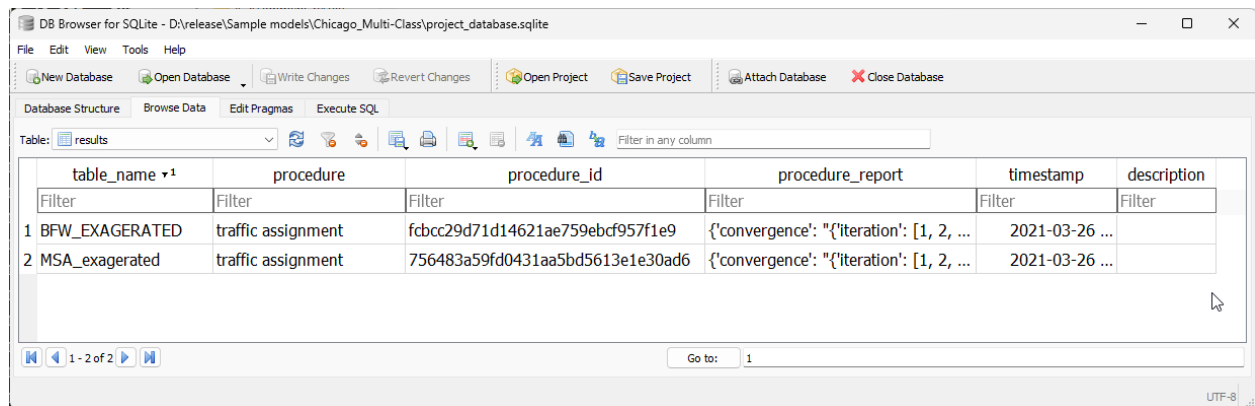
## 2.3.8 Results

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:





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.

## 2.3.9 SQL Data model

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

### Conventions

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

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

### Project tables

#### about table structure

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

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

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

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

(\* - Primary key)

The SQL statement for table and index creation is below.

```
CREATE TABLE if not exists about (infoname TEXT UNIQUE NOT NULL,
                                infovalue TEXT
                                );
```

(continues on next page)

(continued from previous page)

```

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 attributes can be added into the table.

The **name\_table** field holds the name of the table that has the attribute

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

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

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

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

(\* - Primary key)

The SQL statement for table and index creation is below.

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

CREATE INDEX 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 of 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');
```

(continues on next page)

(continued from previous page)

```

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(TYPEOF(a_node) == 'integer')
                                CHECK(TYPEOF(b_node) == 'integer')
                                CHECK(TYPEOF(direction) == 'integer')
                                CHECK(LENGTH(modes)>0)
                                CHECK(LENGTH(direction)==1));

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

CREATE UNIQUE INDEX idx_link ON links (link_id);

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

CREATE INDEX idx_link_anode ON links (a_node);

CREATE INDEX idx_link_bnode ON links (b_node);

CREATE INDEX idx_link_modes ON links (modes);

CREATE INDEX idx_link_link_type ON links (link_type);

CREATE INDEX idx_links_a_node_b_node ON links (a_node, b_node);

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

```

(continues on next page)

(continued from previous page)

```

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

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

The **file\_name** field holds the file name.

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

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

The **procedure\_id** field holds an unique alpha-numeric identifier for this procedure.

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

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

Field	Type	NULL allowed	Default Value
name*	TEXT	NO	
file_name	TEXT	NO	
cores	INTEGER	NO	1
procedure	TEXT	YES	
procedure_id	TEXT	YES	
timestamp	DATETIME	YES	current_timestamp
description	TEXT	YES	

(\* - Primary key)

The SQL statement for table and index creation is below.

```

create TABLE if not exists matrices (name
                                     file_name
                                     cores
                                     procedure
                                     procedure_id
                                     timestamp
                                     description
                                     TEXT
                                     TEXT
                                     INTEGER
                                     TEXT,
                                     TEXT,
                                     DATETIME
                                     TEXT);
                                     NOT NULL PRIMARY KEY,
                                     NOT NULL UNIQUE,
                                     NOT NULL DEFAULT 1,
                                     DEFAULT current_timestamp,

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');
```

(continues on next page)

(continued from previous page)

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

## 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');
```

(continues on next page)



(continued from previous page)

```

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','modes', 'Modes connected to the node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','link_types', 'Link types connected to the node');

```

### periods table structure

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

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

(\* - Primary key)

The SQL statement for table and index creation is below.

```

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

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

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

```

### results table structure

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

The **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');
```

## 2.4 Public Transport database

AequilibraE is capable of importing a General Transit Feed Specification (GTFS) feed into its database. The Transit module has been updated in version 0.9.0. More details on the **public\_transport.sqlite** are discussed on a nearly *per-table* basis below, and we recommend understanding the role of each table before setting an AequilibraE model you intend to use. If you don't know much about GTFS, we strongly encourage you to take a look at the documentation provided by [Google](#).

A more technical view of the database structure, including the SQL queries used to create each table and their indices are also available.

## 2.4.1 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 [here](#).

**agency\_id** identifies the agency for the specified route

**agency** contains the full name of the transit agency

**feed\_date** indicates the date for which the GTFS feed is being imported

**service\_date** indicates the date for the 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 attributes can be added into the table.

The **name\_table** field holds the name of the table that has the attribute

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

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

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

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

(\* - Primary key)

The SQL statement for table and index creation is below.

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

CREATE INDEX 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 [here](#).

**fare\_id** identifies a fare class

**fare** describes a fare class

**agency\_id** identifies a relevant agency for a fare.

**price** specifies the fare price

**currency\_code** specifies the currency used to pay the fare

**payment\_method** indicates when the fare must be paid.

**transfer** indicates the number of transfers permitted on the fare

**transfer\_duration** indicates the length of time in seconds before a transfer expires.

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

(\* - Primary key)

The SQL statement for table and index creation is below.

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

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

### fare rules table structure

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

The **fare\_id** identifies a fare class

The **route\_id** identifies a route associated with the fare class.

The **origin** field identifies the origin zone

The **destination** field identifies the destination zone

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	INTEGER	YES	
destination	INTEGER	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       INTEGER,
  destination  INTEGER,
  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,
  FOREIGN KEY(destination) REFERENCES fare_zones(fare_zone_id) deferrable initially_
↳deferred,
  FOREIGN KEY(origin) REFERENCES fare_zones(fare_zone_id) deferrable initially_
↳deferred,
  FOREIGN KEY(contains) REFERENCES fare_zones(fare_zone_id) deferrable initially_
↳deferred
);
```

### fare zones table structure

The *zones* tables holds information on the fare transit zones and the TAZs they are in.

**fare\_zone\_id** identifies the fare zone for a stop

**transit\_zone** identifies the TAZ for a fare zone

**agency\_id** identifies the agency for the specified route

Field	Type	NULL allowed	Default Value
fare_zone_id*	INTEGER	YES	
transit_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 (
  fare_zone_id  INTEGER PRIMARY KEY,
  transit_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 of lanes for the link type

The **lane\_capacity** field presents the number of lanes for the link type

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

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

(\* - Primary key)

The SQL statement for table and index creation is below.

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

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

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

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

(continues on next page)



(continued from previous page)

```
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('link_
↪types','speed', '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
<code>ogc_fid*</code>	INTEGER	YES	
<code>link_id</code>	INTEGER	NO	
<code>a_node</code>	INTEGER	YES	
<code>b_node</code>	INTEGER	YES	
<code>direction</code>	INTEGER	NO	0
<code>distance</code>	NUMERIC	YES	
<code>modes</code>	TEXT	NO	
<code>link_type</code>	TEXT	YES	
<code>line_id</code>	TEXT	YES	
<code>stop_id</code>	TEXT	YES	
<code>line_seg_idx</code>	INTEGER	YES	
<code>trav_time</code>	NUMERIC	NO	
<code>freq</code>	NUMERIC	NO	
<code>o_line_id</code>	TEXT	YES	
<code>d_line_id</code>	TEXT	YES	
<code>transfer_id</code>	TEXT	YES	
<code>geometry</code>	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,
```

(continues on next page)

(continued from previous page)

```

        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')
        CHECK(TYPEOF(a_node) == 'integer')
        CHECK(TYPEOF(b_node) == 'integer')
        CHECK(TYPEOF(direction) == 'integer')
        CHECK(LENGTH(modes)>0)
        CHECK(LENGTH(direction)==1));

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

CREATE UNIQUE INDEX idx_link ON links (link_id);

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

CREATE INDEX idx_link_anode ON links (a_node);

CREATE INDEX idx_link_bnode ON links (b_node);

CREATE INDEX idx_link_modes ON links (modes);

CREATE INDEX idx_link_link_type ON links (link_type);

CREATE INDEX idx_links_a_node_b_node ON links (a_node, b_node);

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

```

(continues on next page)

(continued from previous page)

```

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('links
↳','line_seg_idx', 'Line segment index');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('links
↳','trav_time', 'Travel time');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('links
↳','freq', 'Frequency of link traversal');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('links
↳','*_line_id', 'Origin/Destination line ID for transfer links');
INSERT 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

```

(continues on next page)

(continued from previous page)

```

↪');
INSERT INTO 'modes' (mode_name, mode_id, description) VALUES('bicycle', 'b', 'Biking_
↪links');

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

```

### node types table structure

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

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

The **node\_type\_id** field presents the identification of the node type

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

Attributes follow

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

(\* - Primary key)

The SQL statement for table and index creation is below.

```

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

INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('default', 'y',
↪'Default general node type');
INSERT INTO 'node_types' (node_type, node_type_id, description) VALUES('od', 'n',
↪'Origin/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');

```

(continues on next page)

(continued from previous page)

```

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

INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('node_
↪ types','node_type', 'Node type name. E.g stop or boarding');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('node_
↪ types','node_type_id', 'Single letter identifying the mode. E.g. a, for alighting');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('node_
↪ types','description', 'Description of the same. E.g. Stop nodes connect ODs and
↪ walking nodes to boarding and alighting nodes via boarding and alighting links.');
```

## nodes table structure

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

The **node\_id** field is an identifier of the node.

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

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

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

The **line\_seg\_idx** field indexes the segment of line **line\_id**. Zero based.

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

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

The **node\_type** field identifies the types of this node.

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

Field	Type	NULL allowed	Default Value
ogc_fid*	INTEGER	YES	
node_id	INTEGER	NO	
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	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,
                                   stop_id      TEXT,
                                   line_id      TEXT,
                                   line_seg_idx  INTEGER,
                                   modes        TEXT,
                                   link_types   TEXT,
                                   node_type    TEXT,
                                   taz_id      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
↪','stop_id', 'ID of the Stop this node belongs to');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','line_id', 'ID of the Line this node belongs to');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','line_seg_idx', 'Index of the line segment this node belongs to');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','modes', 'Modes connected to the node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','link_types', 'Link types connected to the node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','node_type', 'Node types of this node');
INSERT INTO 'attributes_documentation' (name_table, attribute, description) VALUES('nodes
↪','taz_id', 'Transit assignemnt 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,
```

(continues on next page)



(continued from previous page)

```

    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 it [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 the 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);
```

(continues on next page)

(continued from previous page)

```

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 it [here](#).

**stop\_id** is an unique identifier for a stop

**stop** identifies 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

**fare\_zone\_id** identifies the fare zone for a stop

**transit\_zone** identifies the TAZ for a fare zone

**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	
fare_zone_id	INTEGER	YES	
transit_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,
  fare_zone_id     INTEGER,
  transit_zone     TEXT,
  route_type       INTEGER   NOT NULL DEFAULT -1,
  FOREIGN KEY(agency_id) REFERENCES agencies(agency_id),
  FOREIGN KEY("fare_zone_id") REFERENCES fare_zones("fare_zone_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 it [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  
);
```

## 2.5 Static Traffic Assignment

Performing static traffic assignment with AequilibraE is not dissimilar than doing so with traditional commercial packages, as we strive to make it as easy as possible for seasoned modelers to migrate their models and workflows to AequilibraE.

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.

Further, many AequilibraE users are new to the *craft*, so we have elected to start creating documentation on the most important topics in the transportation modeling practice, where we detail how these concepts are translated into the AequilibraE tools and recommended workflows.

### 2.5.1 Multi-class Equilibrium assignment

While single-class equilibrium traffic assignment<sup>1</sup> is mathematically simple, multi-class traffic assignment<sup>7</sup>, 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.

#### Cost function

AequilibraE supports class-specific cost functions, where each class can include the following:

- PCE
- Link-based fixed financial cost components
- Value-of-Time (VoT)

---

<sup>1</sup> 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>

<sup>7</sup> 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](https://doi.org/10.1016/S0927-0507(06)14010-4)

## Technical requirements

This documentation is not intended to discuss in detail the mathematical requirements of multi-class traffic assignment, which can be found discussed in detail on<sup>4</sup>.

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.

## Convergence criteria

Convergence in AequilibrE is measured solely in terms of relative gap, which is a somewhat old recommendation<sup>5</sup>, 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](#).

## Algorithms available

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 Averages	All-or-Nothing Assignment (AoN)	Function of the iteration number
Frank-Wolfe	All-or-Nothing Assignment (AoN)	Optimal value derived from Wardrop's principle
Conjugate Frank-Wolfe	Conjugate direction (Current and previous 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

### Note

Our implementations of the conjugate and Biconjugate-Frank-Wolfe methods should be inherently proportional<sup>6</sup>, but we have not yet carried the appropriate testing that would be required for an empirical proof.

<sup>4</sup> 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/10.1177/0361198119837496>

<sup>5</sup> 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](https://doi.org/10.1016/0191-2615(88)90003-3)

<sup>6</sup> 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>

### 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<sup>2</sup>.

### Conjugate Frank-Wolfe

The conjugate direction algorithm was introduced in 2013<sup>3</sup>, 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 as its Biconjugate evolution, so it was born outdated.

### Biconjugate Frank-Wolfe

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.

### 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.

---

<sup>2</sup> LeBlanc L. J., Morlok E. K. and 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](https://doi.org/10.1016/0041-1647(75)90030-1)

<sup>3</sup> Mitradjieva, M. and 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>



## 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.

## 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**.

## Super-network

We deal with a super-network by 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 graphs will be built in a consistent manner and multi-class assignment is possible.

## References

### Traffic assignment and equilibrium

## 2.5.2 Path-finding and assignment mechanics

Performing traffic assignment, or even just computing paths through a network is always a little different in each platform, and in AequilibraE is not different.

The complexity in computing paths through a network comes from the fact that transportation models usually house networks for multiple transport modes, so the toads (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 Graphs

As mentioned above, 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.).

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

```
from aequilibrae.paths import Graph

g = Graph()

# g.network
# g.graph
```

## 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**.

## 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.

```
from aequilibrae import Project

project = Project.from_path("/tmp/test_project")
project.network.build_graphs(modes=["c"]) # We build the graph for cars only

graph = project.network.graphs['c'] # we grab the graph for cars
```

## 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])
```

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

```
graph.network.loc[graph.network.link_type="highway", "speed_AB"] = 100
graph.network.loc[graph.network.link_type="highway", "speed_BA"] = 100

graph.prepare_graph(graph.centroids)
if graph.skim_fields:
    graph.set_skimming(graph.skim_fields)
```

## 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(["tolls", "distance", "free_flow_travel_time"])
```

## 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, 371293], np.int))
graph.set_blocked_centroid_flows(False)
```

## 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 (we know *complex* is subjective).

Below we detail the components that go into performing traffic assignment, but for a comprehensive use case for the traffic assignment module, please see the complete application in [this example](#).

## Traffic Assignment Class

Traffic assignment is organized within a object introduced on version 0.6.1 of the AequilibraE, and includes a small list of member variables which should be populated by the user, providing a complete specification of the assignment procedure:

- **classes:** List of objects *Traffic class* , each of which are a completely specified traffic class
- **vdf:** The Volume delay function (VDF) to be used
- **vdf\_parameters:** The parameters to be used in the volume delay function, other than volume, capacity and free flow time

- **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
- **capacity\_field**: The field of the graph that corresponds to **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
- **algorithm**: The assignment algorithm to be used. (e.g. “all-or-nothing”, “bfw”)

Assignment parameters such as maximum number of iterations and target relative gap come from the global software parameters, that can be set using the [Parameters YAML File](#).

There are also some strict technical requirements for formulating the multi-class equilibrium assignment as an unconstrained convex optimization problem, as we have implemented it. These requirements are loosely listed in [Technical requirements](#).

If you want to see the assignment log on your terminal during the assignment, please look in the [logging to terminal](#) example.

To begin building the assignment it is easy:

```
from aequilibrae.paths import TrafficAssignment

assig = TrafficAssignment()
```

## Volume Delay Function

For now, the only VDF functions available in AequilibraE are the

- BPR<sup>3</sup>

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

- Spiess' conical<sup>2</sup>

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

- 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.

Setting the volume delay function is one of the first things you should do after instantiating an assignment problem in AequilibraE, and it is as simple as:

<sup>3</sup> Hampton Roads Transportation Planning Organization, Regional Travel Demand Model V2 (2020). Available in: [https://www.hrtpo.org/uploads/docs/2020\\_HamptonRoads\\_Modelv2\\_MethodologyReport.pdf](https://www.hrtpo.org/uploads/docs/2020_HamptonRoads_Modelv2_MethodologyReport.pdf)

<sup>2</sup> 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>

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

The implementation of the VDF functions in AequilibraE is written in Cython and fully multi-threaded, and therefore descent methods that may evaluate such function multiple times per iteration should not become unnecessarily slow, especially in modern multi-core systems.

## Traffic class

The Traffic class 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. 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, but which can have an arbitrary number of user-classes, setup as different layers of the matrix object

Example:

```
tc = TrafficClass("car", graph_car, matrix_car)
tc2 = TrafficClass("truck", graph_truck, matrix_truck)
```

- **pce** - The passenger-car equivalent is the standard way of modeling multi-class traffic assignment equilibrium in a consistent manner (see<sup>1</sup> for the technical detail), and it is set to 1 by default. If the **pce** for a certain class should be different than one, one can make a quick method call.
- **fixed\_cost** - 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.
- **vot** - 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, and can be set to the appropriate value (1.0, value-of-time. If the **vot** or whatever conversion factor is appropriate) with a method call.

```
tc2.set_pce(2.5)
tc2.set_fixed_cost("truck_toll")
tc2.set_vot(0.35)
```

To add traffic classes to the assignment instance it is just a matter of making a method call:

```
assig.set_classes([tc, tc2])
```

<sup>1</sup> 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 VDF Parameters

Parameters for VDF functions can be passed as a fixed value to use for all links, or as graph fields. As it is the case for the travel time and capacity fields, VDF parameters need to 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.

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**.

To choose a field that exists in the graph, we just pass the parameters as follows:

```
assig.set_vdf_parameters({"alpha": "alphas", "beta": "betas"})
```

To pass global values, it is simply a matter of doing the following:

```
assig.set_vdf_parameters({"alpha": 0.15, "beta": 4})
```

## Setting final parameters

There are still three parameters missing for the assignment.

- Capacity field
- Travel time field
- Equilibrium algorithm to use

```
assig.set_capacity_field("capacity")
assig.set_time_field("free_flow_time")
assig.set_algorithm(algorithm)
```

## 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 8am to 10am, do the following:

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

Next, add the preload to the assignment.

```
assig.add_preload(preload, 'PT_vehicles')
```

## Executing an Assignment

Finally, one can execute assignment:

```
assig.execute()
```

*Convergence criteria* is discussed in a different section.

## References

## 2.6 Transit assignment

### 2.6.1 Hyperpath routing in the context of transit assignment

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? These are some of the questions that transit assignment models aim to answer. Transit assignment models are mathematical tools that predict how passengers behave and travel in a transit network, given some assumptions and inputs.

One of the basic concepts in transit assignment models is hyperpath routing. Hyperpath routing 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 in this following page 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 classic algorithm by Spiess & Florian<sup>1</sup> implemented in AequilibraE.

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<sup>2</sup>.

Let's start by importing some Python packages.

### Imports

```
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
```

(continues on next page)

<sup>1</sup> Spiess, H., 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](https://doi.org/10.1016/0191-2615(89)90034-9)

<sup>2</sup> Ma, J., Fukuda, D., 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>

(continued from previous page)

```
RS = 124 # random seed
FS = (6, 6) # figure size
```

## 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  $f_a \geq 0$

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<sup>Page 143, 2<sup>3</sup></sup>. 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  $\alpha$  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:

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

A smaller  $\alpha$  value implies higher edge frequencies, and vice versa. Next, we will create the network as a pandas dataframe.

## Vertices

```
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
```

```
n = 10
vertices = create_vertices(n)
vertices.head(3)
```

<sup>3</sup> 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>.



	x	y
0	0.000000	0.0
1	0.111111	0.0
2	0.222222	0.0

```
@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
```

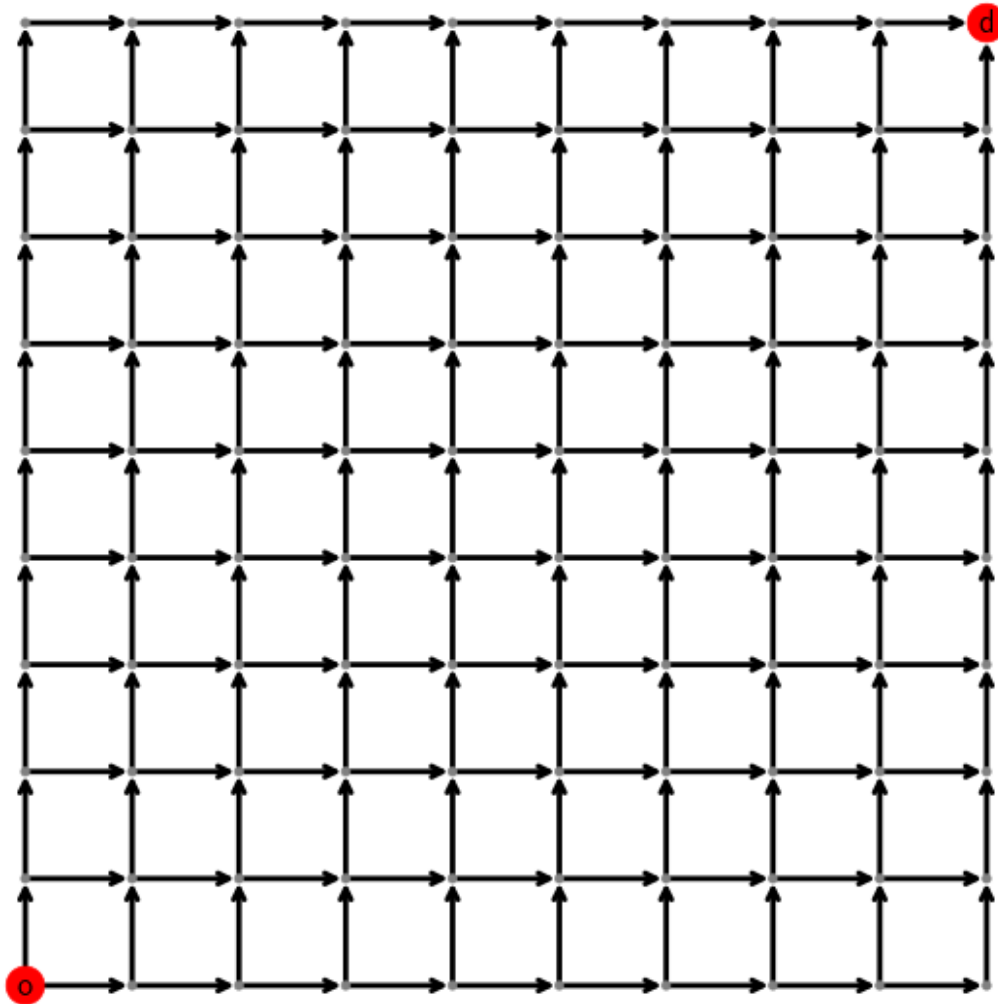
```
edges = create_edges(n, seed=RS)
edges.head(3)
```

	tail	head	trav_time	delay_base
0	0	1	0.785253	0.287917
1	0	10	0.785859	0.970429
2	10	11	0.969136	0.854512

## Plot the network

We use the [NetworkX](#) package to plot the network. The bottom left vertex is the origin ('o') and the top right vertex is the destination ('d') for the hyperpath computation.

```
# NetworkX
n_vertices = n * n
pos = vertices[["x", "y"]].values
G = nx.from_pandas_edgelist(
    edges,
    source="tail",
    target="head",
    edge_attr=["trav_time", "delay_base"],
    create_using=nx.DiGraph,
)
widths = 2
figure = plt.figure(figsize=FS)
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(f"Bell's network with  $n={n}$ ", color="k")
```

Bell's network with  $n=10$ 

We can also visualize the edge travel time:

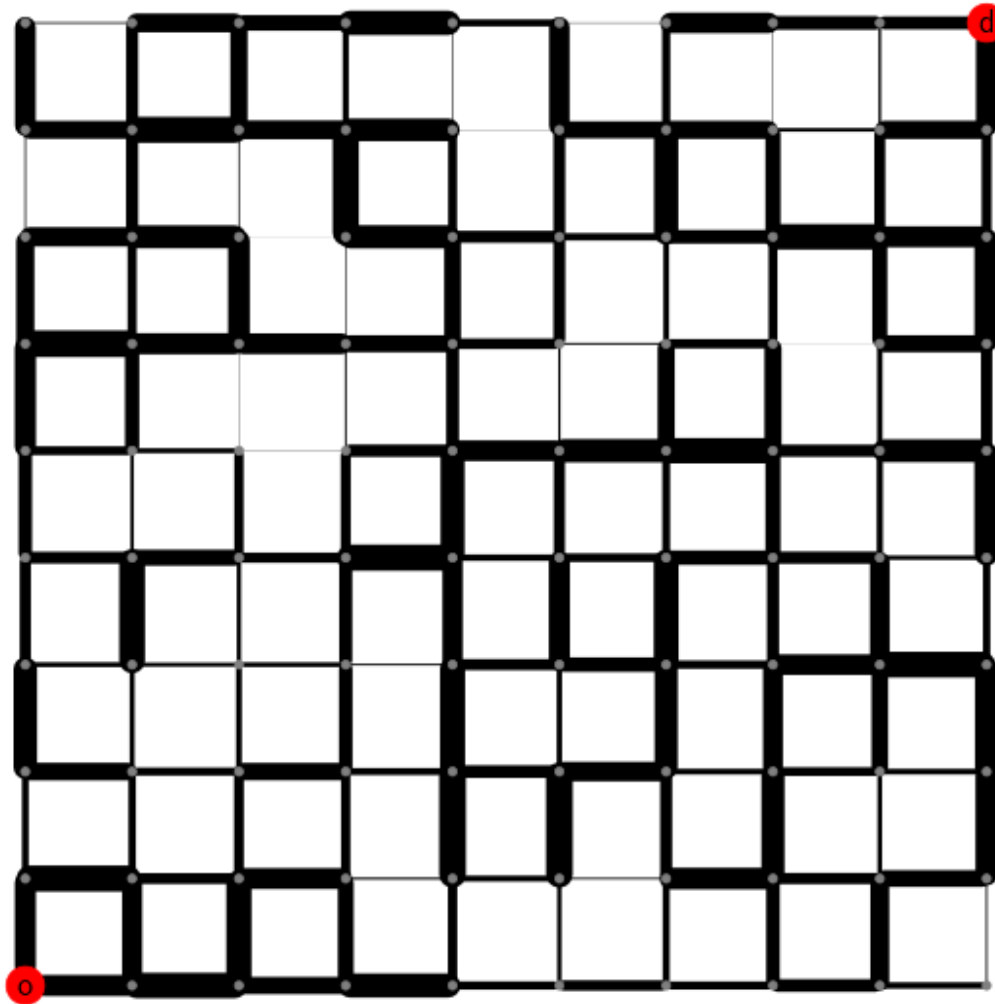
```
widths = 1e2 * np.array([G[u][v]["trav_time"] for u, v in G.edges()]) / n
_ = plt.figure(figsize=FS)
node_colors = n_vertices * ["gray"]
node_colors[0] = "r"
node_colors[-1] = "r"
ns = 100 / n
node_size = n_vertices * [ns]
node_size[0] = 20 * ns
node_size[-1] = 20 * ns
labeldict = {}
labeldict[0] = "o"
```

(continues on next page)

(continued from previous page)

```
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(
    "Bell's network - edge travel time :  $\textit{trav\_time}$ ", color="k"
)
```

Bell's network - edge travel time : *trav\_time*



And the base delay:

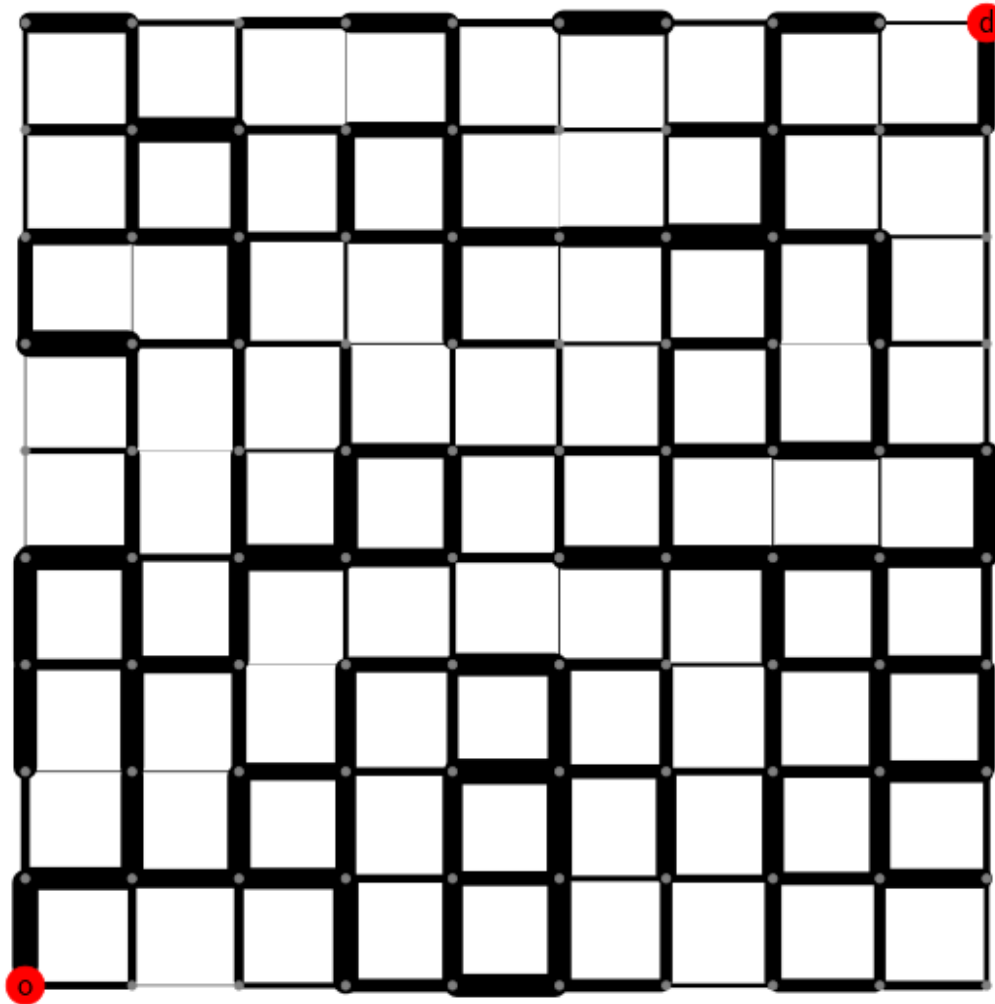
```
widths = 1e2 * np.array([G[u][v]["delay_base"] for u, v in G.edges()]) / n
_ = plt.figure(figsize=FS)
node_colors = n_vertices * ["gray"]
node_colors[0] = "r"
node_colors[-1] = "r"
ns = 100 / n
node_size = n_vertices * [ns]
node_size[0] = 20 * ns
node_size[-1] = 20 * ns
labeldict = {}
labeldict[0] = "o"
```

(continues on next page)

(continued from previous page)

```
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("Bell's network - edge base delay :  $\textit{delay\_base}$ ", color="k")
```

## Bell's network - edge base delay : *delay\_base*



### Hyperpath computation

We now introduce a function `plot_shortest_hyperpath` that:

- creates the network,
- computes the edge frequency given an input value for  $\alpha$ ,
- compute the shortest hyperpath,
- plot the network and hyperpath with NetworkX.

```
def plot_shortest_hyperpath(n=10, alpha=10.0, figsize=FS, seed=RS):
```

(continues on next page)

(continued from previous page)

```

# network creation
vertices = create_vertices(n)
n_vertices = n * n
edges = create_edges(n, seed=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

# 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)

# NetworkX
pos = vertices[["x", "y"]].values
G = nx.from_pandas_edgelist(
    sf._edges,
    source="tail",
    target="head",
    edge_attr="volume",
    create_using=nx.DiGraph,
)
widths = 1e2 * np.array([G[u][v]["volume"] for u, v in G.edges()]) / n
figure = 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,
)

```

(continues on next page)



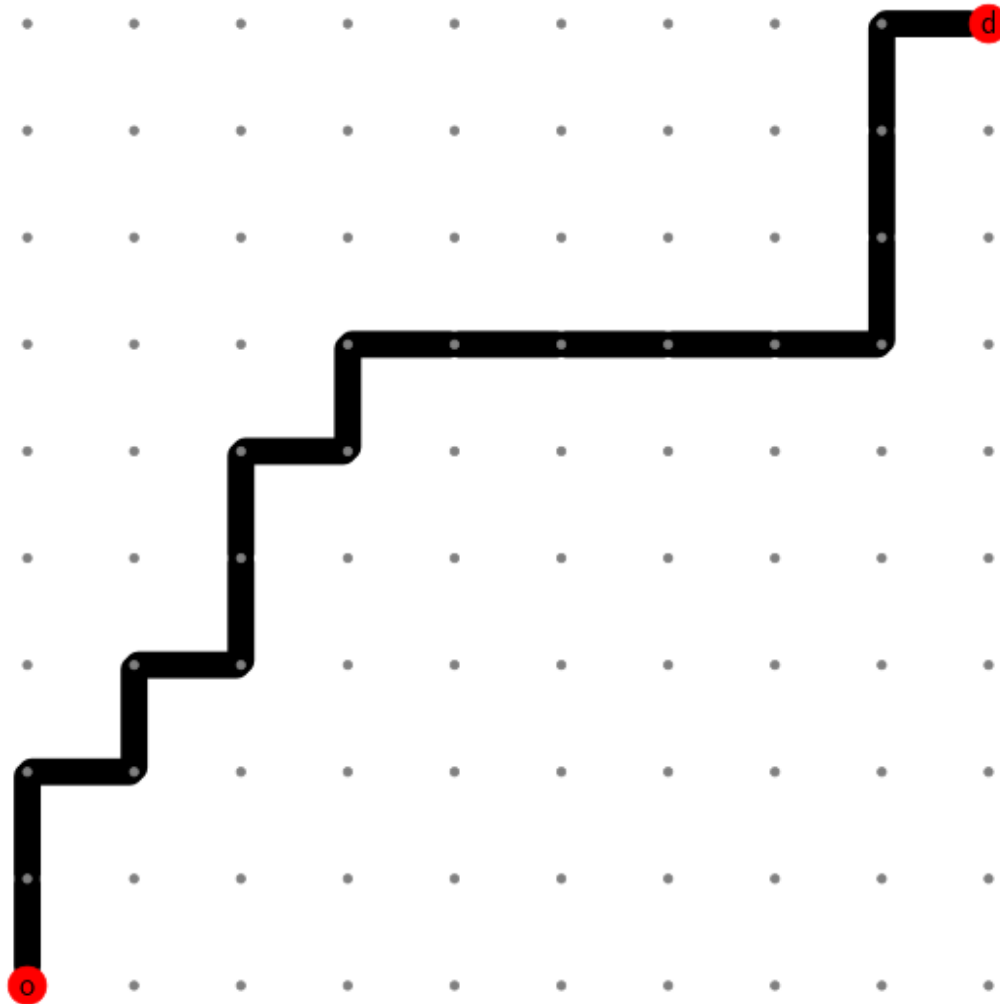
(continued from previous page)

```
ax = plt.gca()
_ = ax.set_title(
    f"Shortest hyperpath - Bell's network  $\alpha={alpha}$ ", color="k"
)
```

We start with  $\alpha = 0$ . This implies that there is no delay over all the network.

```
plot_shortest_hyperpath(n=10, alpha=0.0)
```

## Shortest hyperpath - Bell's network $\alpha=0.0$



The hyperpath that we obtain is the same as the shortest path that Dijkstra's algorithm would have computed. We call NetworkX's `dijkstra_path` method in order to compute the shortest path:

```

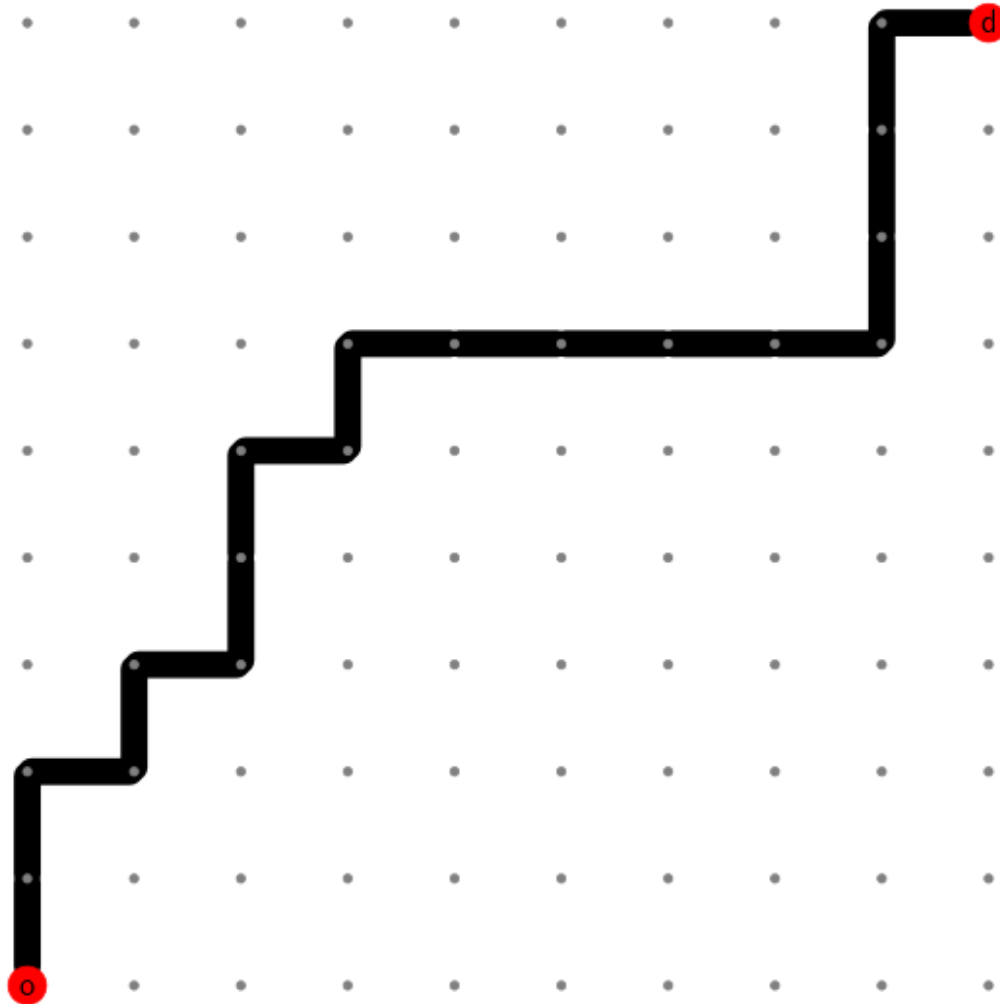
G = nx.from_pandas_edgelist(
    sf._edges,
    source="tail",
    target="head",
    edge_attr="trav_time",
    create_using=nx.DiGraph,
)

# Dijkstra
nodes = nx.dijkstra_path(G, 0, n*n-1, weight='trav_time')
edges = list(nx.utils.pairwise(nodes))

# plot
figure = plt.figure(figsize=FS)
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"
widths = 1e2 * np.array([1 if (u,v) in edges else 0 for u, v in G.edges()]) / n
pos = vertices[["x", "y"]].values
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(
    f"Shortest path - Bell's network", color="k"
)

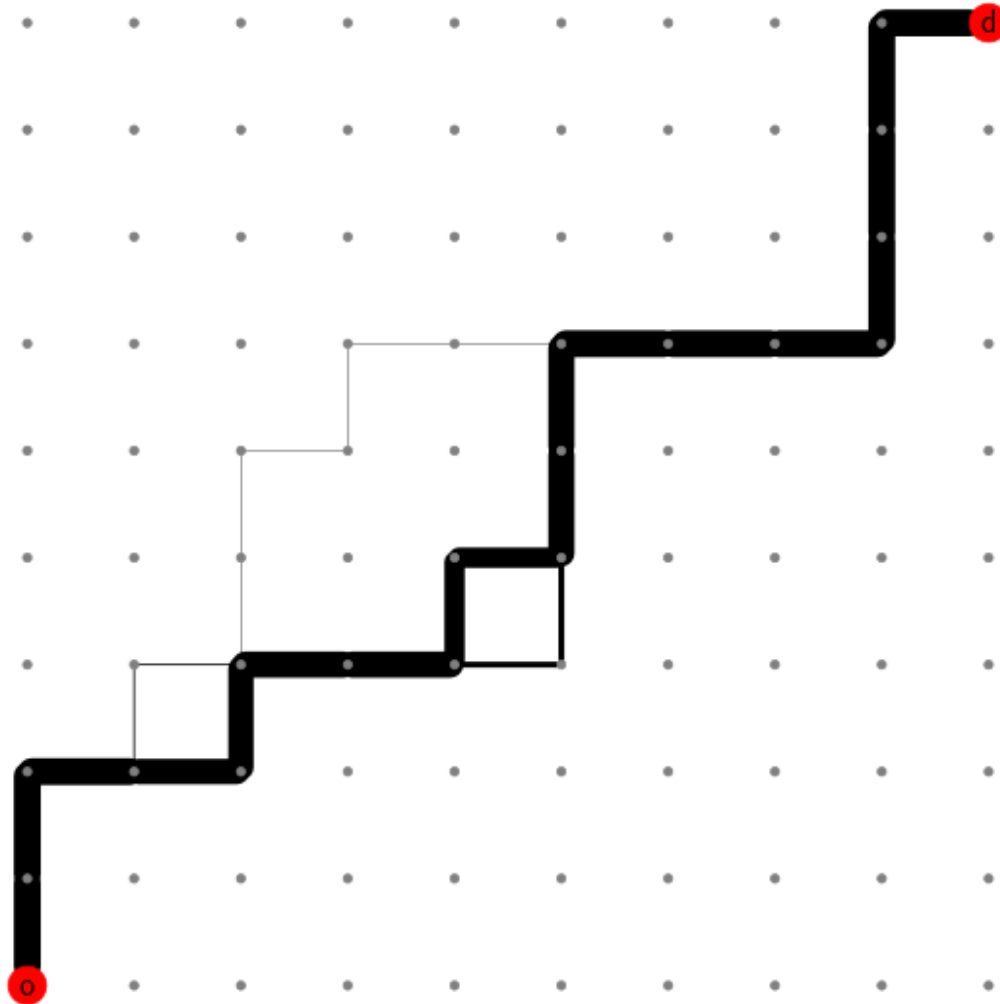
```

## Shortest path - Bell's network



Let's introduce some delay by increasing the value of  $\alpha$ :

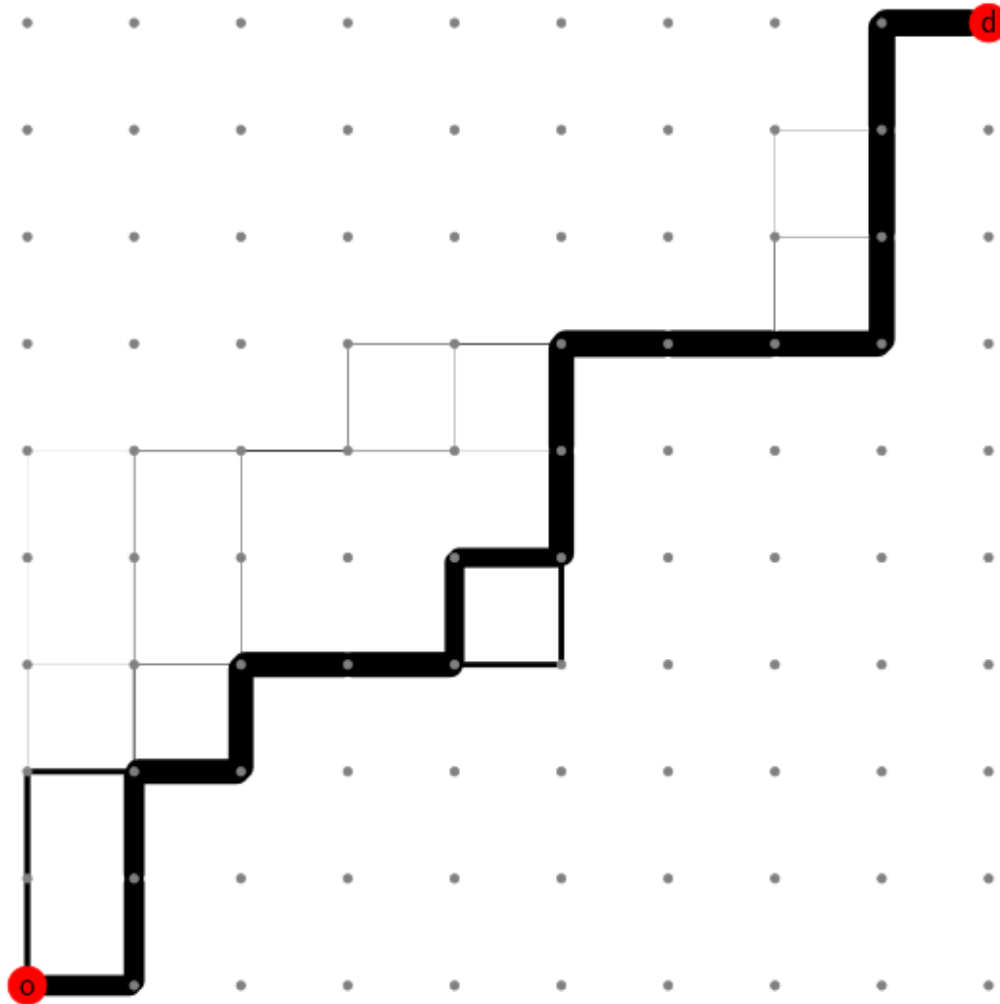
```
plot_shortest_hyperpath(n=10, alpha=0.5)
```

Shortest hyperpath - Bell's network  $\alpha=0.5$ 

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  $\alpha$ :

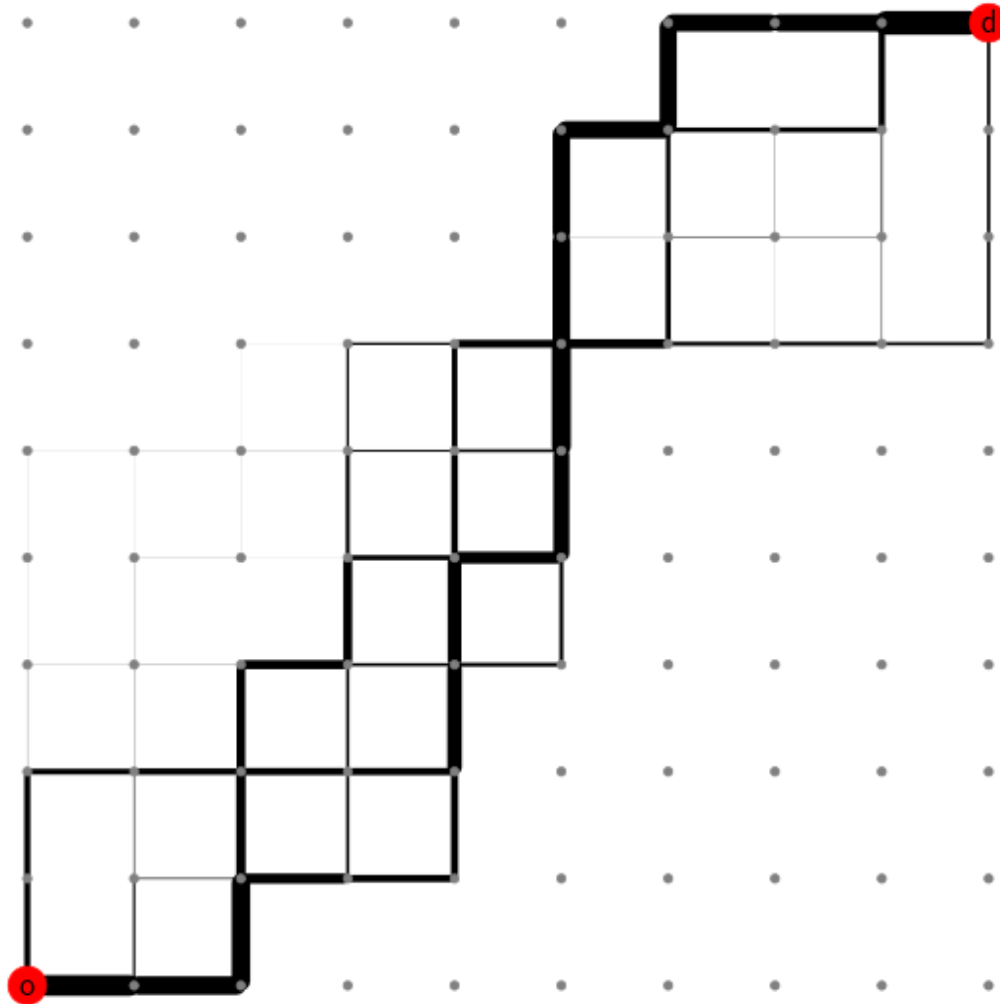
```
plot_shortest_hyperpath(n=10, alpha=1.0)
```

## Shortest hyperpath - Bell's network $\alpha=1.0$



```
plot_shortest_hyperpath(n=10, alpha=100.0)
```

## Shortest hyperpath - Bell's network $\alpha=100.0$



### References

#### 2.6.2 The Transit assignment graph

This page is a description of 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 & Florian<sup>1</sup>.

Let’s start by giving a few definitions:

- *transit* definition from [Wikipedia](#):

<sup>1</sup> Spiess, H., 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](https://doi.org/10.1016/0191-2615(89)90034-9)

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.

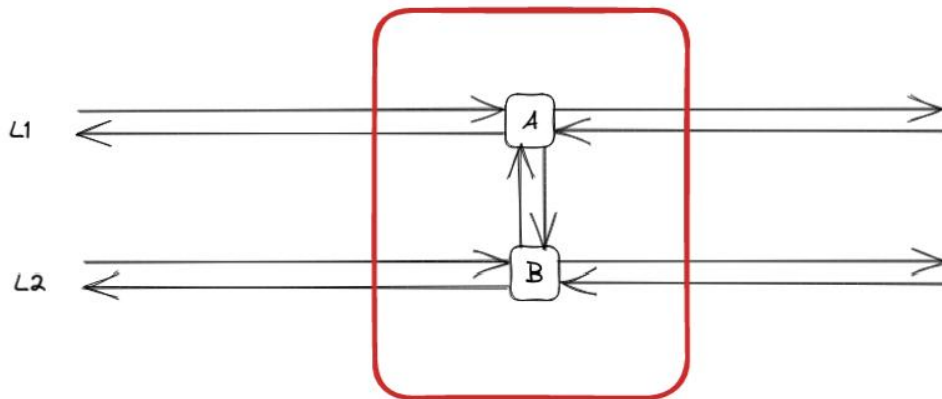
- *transit network*: a set of transit lines and stops, where passengers can board, alight or change vehicles.
- *assignment*: distribution of the passengers (demand) on the network (supply), knowing that transit users attempt to minimize total travel time, time or distance walking, time waiting, number 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.
- *frequency-based* (or *headway-based*) as opposed to schedule-based : schedules are averaged in order to get line frequencies. In the schedule-based 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.
- *link-based*: 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)  $c$  [s] and frequency  $f$  [1/s].

We are going at first to describe the input transit network, which is mostly composed of stops, lines and zones.

## Elements of a transit network

### Transit stops and stations

Transit stops are points where passenger can board, alight or change vehicles. Also, they can be part of larger stations:



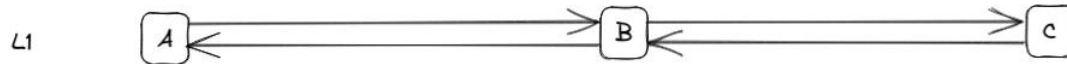
In the illustration above, two distinct stops, A and B, are highlighted, both affiliated with the same station (depicted in red).

## Transit lines

A transit line is a set of services that may use different routes, decomposed into segments.

## Transit routes

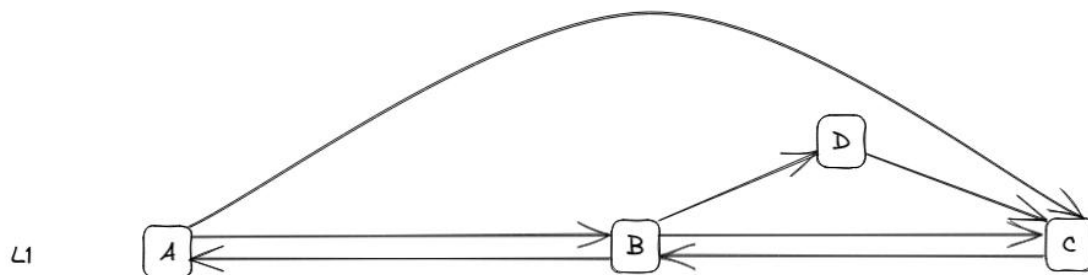
A route is described by a sequence of stop nodes. We assume here the routes to be directed. For example, we can take a simple case with 3 stops:



In this case, the L1 line is made of two distinct routes: - ABC - CBA.

Various configurations are possible, 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 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	3600
L1_a3	L1	AB	3600
L1_a4	L1	AC	3600
L1_b1	L1	CBA	600

Headway, associated with each sub-line, 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:

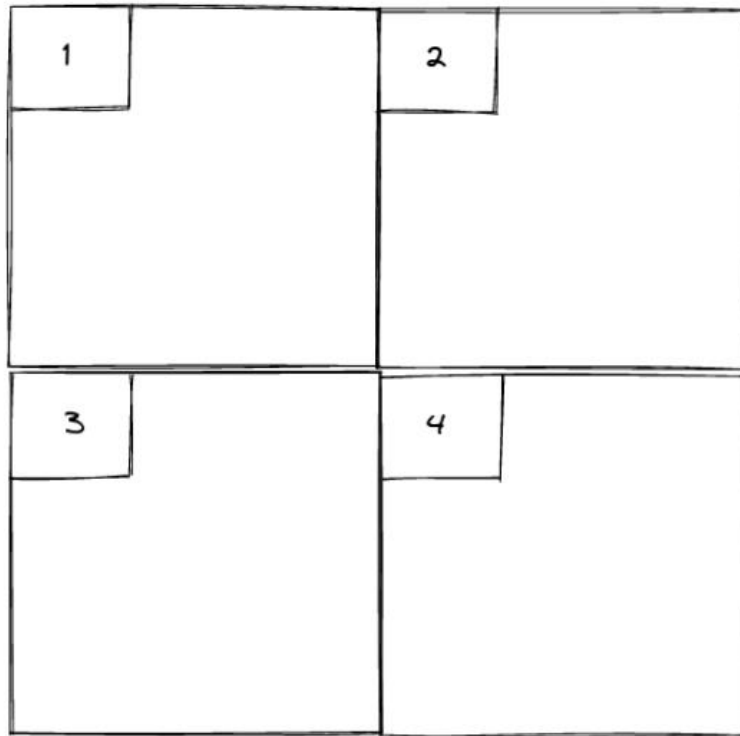
line id	segment index	origin stop	destination stop	travel_time (s)
L1_a1	1	A	B	300
L1_a1	2	B	C	600

Note that a travel time is included for each line segment, serving as another link attribute used by the assignment algorithm.

Note that a travel time is included for each line segment, serving as another link attribute used by the assignment algorithm.

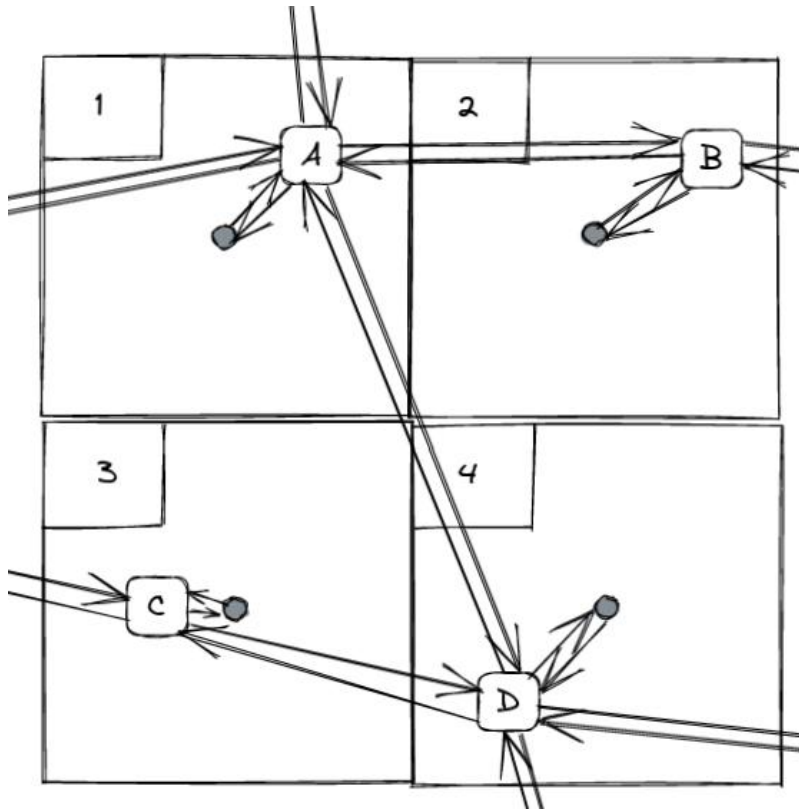
## Transit assignment zones and connectors

To effectively assign passengers on the network, expressing demand between regions is crucial. This is achieved by first decomposing the network area into a partition of transit assignment zones, as illustrated below with 4 non-overlapping zones:



The demand is then expressed as a number of trips from each zone to every other zone, forming a 4 by 4 Origin/Destination (OD) matrix in this case.

Additionally, each zone centroid is connected to specific network nodes to facilitate the connection between supply and demand. These connection points are referred to as *connectors*.



With these components, we now have all the elements required to describe the assignment graph.

## The Assignment graph

### Link and node types

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.

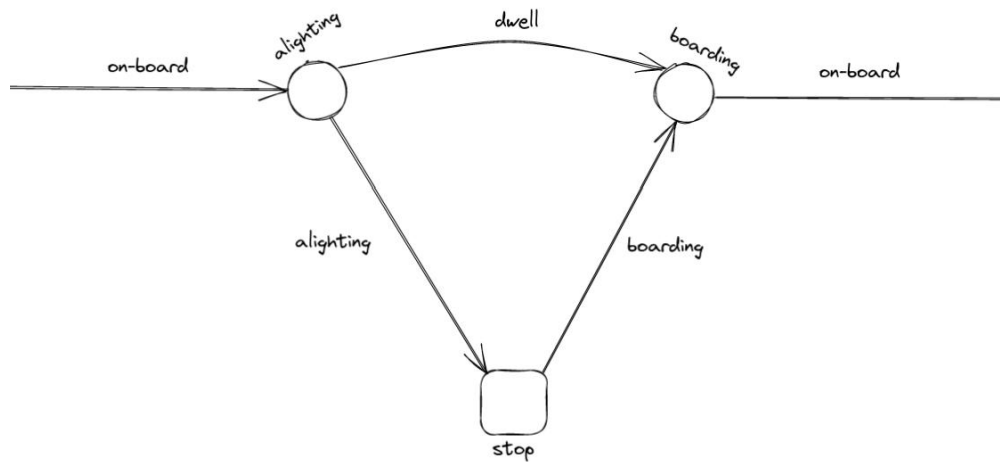
#### Link types:

- *on-board*
- *boarding*
- *alighting*
- *dwell*
- *transfer*
- *connector*
- *walking*

#### Nodes types:

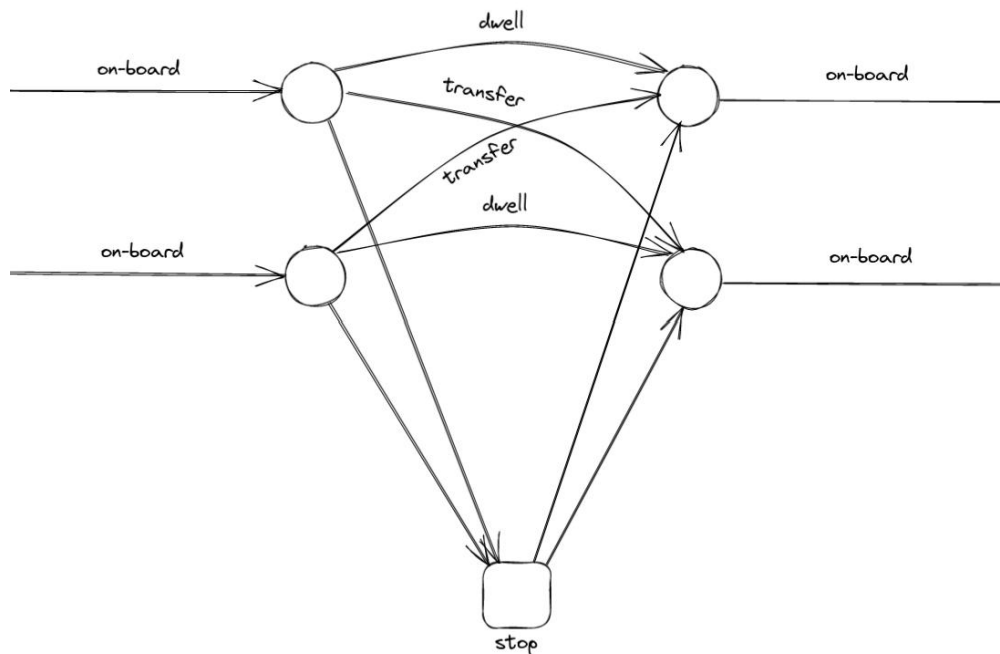
- *stop*
- *boarding*
- *alighting*
- *od*
- *walking*

To illustrate, consider the anatomy of a simple stop:



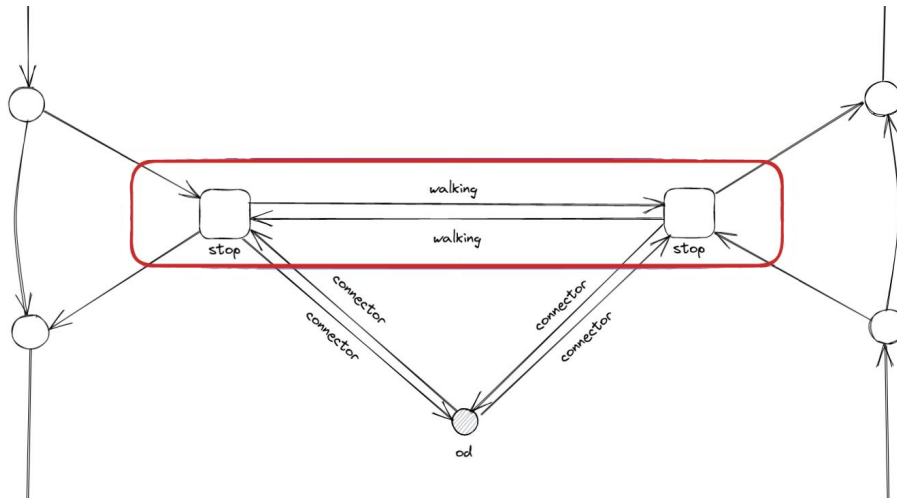
Waiting links encompass *boarding* and *transfer* links. Each line segment is associated with a *boarding*, an *on-board* and an *alighting* link.

*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.

## Link attributes

The table below summarizes link characteristics and attributes based on link types:

link type	from node type	to node type	cost	frequency
<i>on-board</i>	<i>boarding</i>	<i>alighting</i>	trav. time	$\infty$
<i>boarding</i>	<i>stop</i>	<i>boarding</i>	const.	line freq.
<i>alighting</i>	<i>alighting</i>	<i>stop</i>	const.	$\infty$
<i>dwell</i>	<i>alighting</i>	<i>boarding</i>	const.	$\infty$
<i>transfer</i>	<i>alighting</i>	<i>boarding</i>	const. + trav. time	dest. line freq.
<i>connector</i>	<i>od or stop</i>	<i>od or stop</i>	trav. time	$\infty$
<i>walking</i>	<i>stop or walking</i>	<i>stop or walking</i>	trav. time	$\infty$

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.

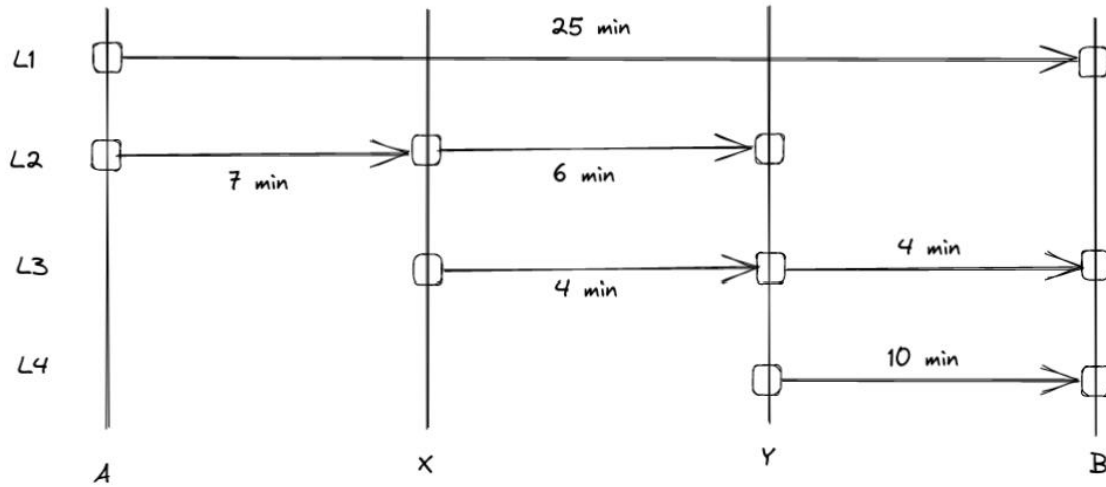
Additional attributes can be introduced for specific link types, such as:

- *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

In the next section, we will explore a small classic transit network example featuring four stops and four lines.

### A Small example : Spiess and Florian

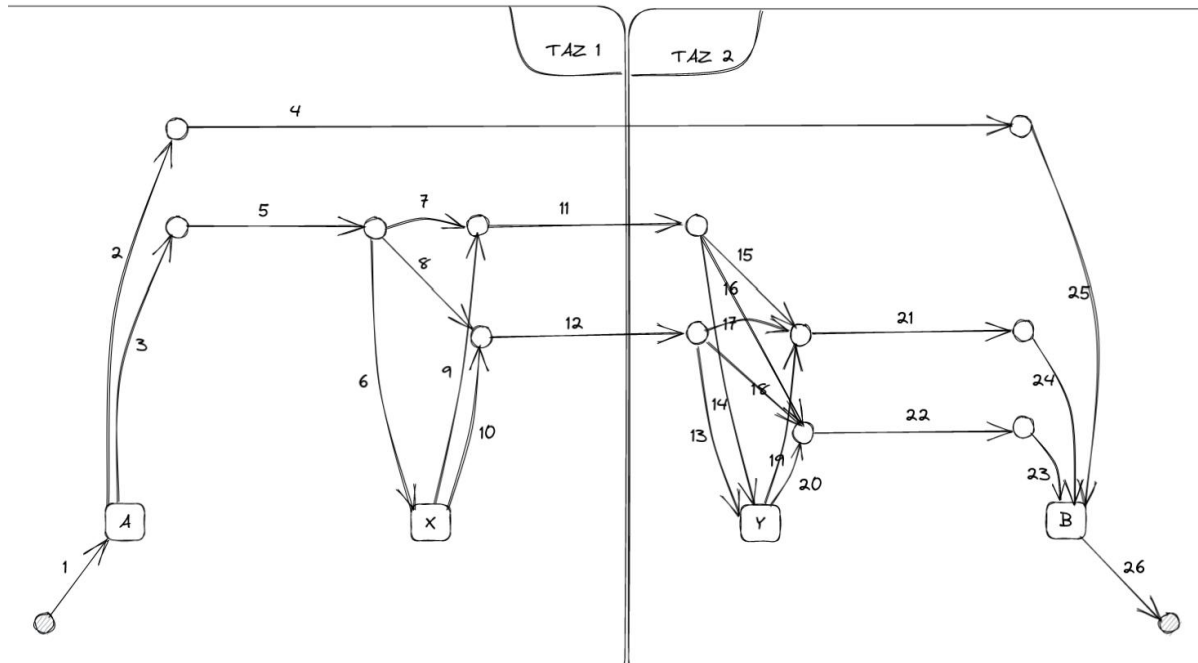
This illustrative example is taken from *Spiess and Florian*<sup>Page 158, 1</sup>:



Travel time are indicated on the figure. 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	$\infty$
2	boarding	L1	0	0.001388889
3	boarding	L2	0	0.001388889
4	on-board	L1	1500	$\infty$
5	on-board	L2	420	$\infty$
6	alighting	L2	0	$\infty$
7	dwell	L2	0	$\infty$
8	transfer		0	0.000555556
9	boarding	L2	0	0.001388889
10	boarding	L3	0	0.000555556
11	on-board	L2	360	$\infty$
12	on-board	L3	240	$\infty$
13	alighting	L3	0	$\infty$
14	alighting	L2	0	$\infty$
15	transfer	L3	0	0.000555556
16	transfer		0	0.002777778
17	dwell	L3	0	$\infty$
18	transfer		0	0.002777778
19	boarding	L3	0	0.000555556
20	boarding	L4	0	0.002777778
21	on-board	L3	240	$\infty$
22	on-board	L4	600	$\infty$
23	alighting	L4	0	$\infty$
24	alighting	L3	0	$\infty$
25	alighting	L1	0	$\infty$
26	connector		0	$\infty$

## 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.

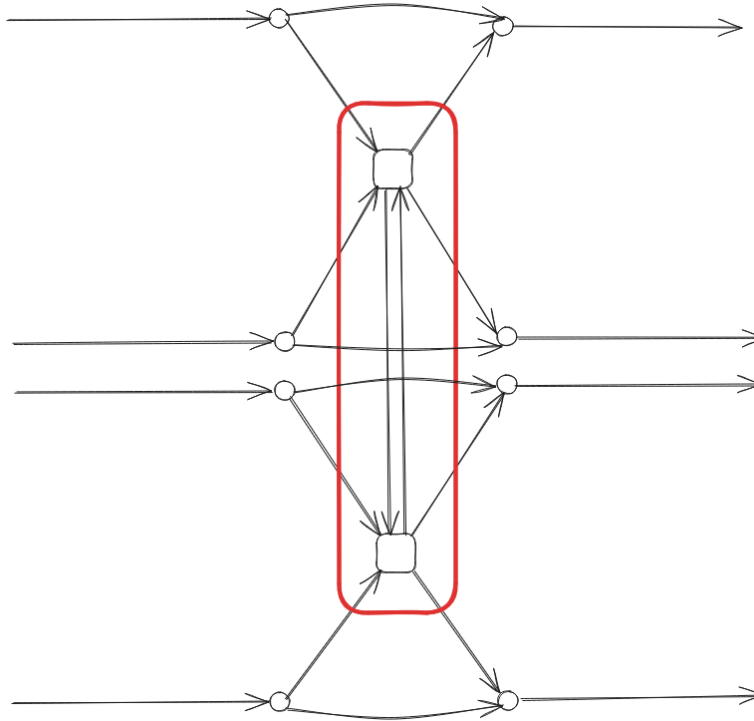
Note that during the assignment, if passengers have the choice between a transfer edge or a walking edge for a line change, they will always be assigned to the transfer edge.

This leads to these possible edge types:

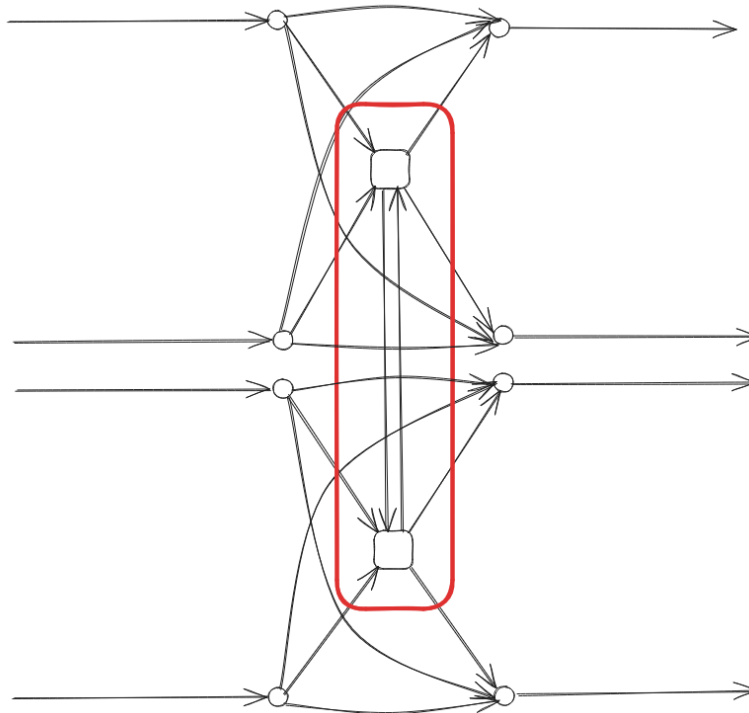
- on-board
- boarding
- alighting
- dwell
- access\_connector
- egress\_connector
- inner\_transfer
- outer\_transfer
- walking

Here is a simple example of a station with two stops, with two lines each:

- walking edges only:

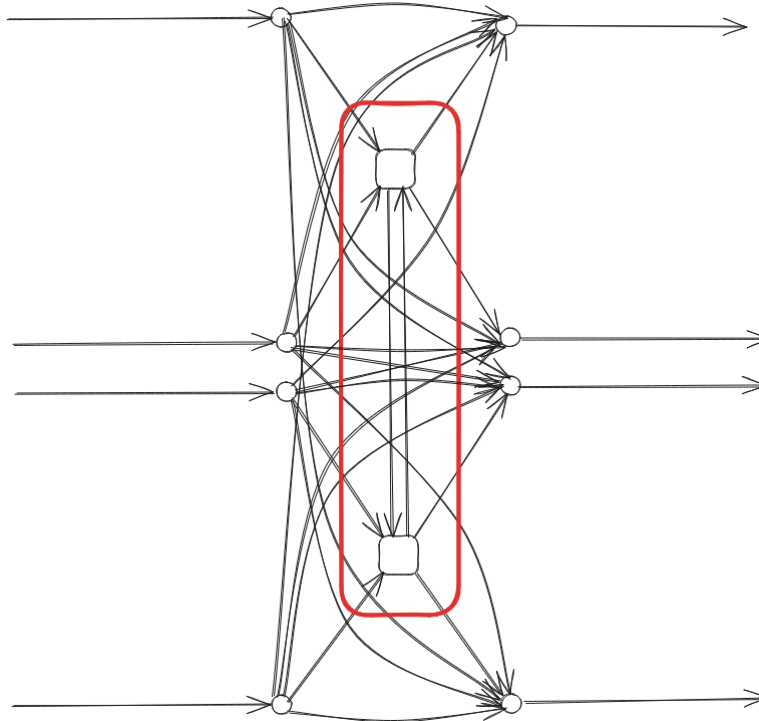


- inner transfer edges, but no outer transfer ones:



- both inner and outer transfer edges:





As an illustrative example, if we build the graph for the city of Lyon France (GTFS files from 2022) on a given day, we get 20196 vertices and 91107 edges, with `with_walking_edges=True`, `with_inner_stop_transfers=True`, `with_outer_stop_transfers=True` and `blocking_centroid_flow=False`. Here is the distribution of edge types:

Edge type	Count
outer_transfer	27287
inner_transfer	10721
walking	9140
on-board	7590
boarding	7590
alighting	7590
dwell	7231
access_connector	6979
egress_connector	6979

and vertex types:

Vertex type	Count
alighting	7590
boarding	7590
stop	4499
od	517

## References

## 2.7 Route Choice

As argued in the literature<sup>3</sup>, the route choice problem does not have a closed solution, and the selection of one of the many modelling frameworks<sup>4</sup> depends on many factors. 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.

### 2.7.1 Costs, utilities and signs

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.

### 2.7.2 Choice set generation algorithms available

All algorithms have been implemented as a single software class

Algorithm	Brief description
Link-Penalisation	Classical link penalisation.
Breadth-First Search with Link Removal	As described in <sup>4</sup> .
Breadth-First Search with Link Removal + Link-Penalisation	A combination of BFS-LE and LP See <i>RouteChoice</i> documentation

## Imports

```
from aequilibrae.paths.route_choice_set import RouteChoiceSet
from aequilibrae import Project

proj = Project()
proj.load('path/to/project/folder')

proj.network.build_graphs()
graph = proj.network.graphs['c']

# Measure that will be used to compute paths
theta = 0.0014
graph.network = graph.network.assign(utility=graph.network.distance * theta)
graph.set_graph('utility')

graph.prepare_graph(centroids=[list_of_all_nodes_network_centroids])
```

(continues on next page)

---

<sup>3</sup> Zill, J. C., and P. V. de Camargo. State-Wide Route Choice Models (Submitted). Presented at the ATRF, Melbourne, Australia, 2024.

<sup>4</sup> Rieser-Schüssler, N., Balmer, M., & 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>

(continued from previous page)

```
rc = RouteChoice(graph, mat)
rc.set_choice_set_generation("bfsle", max_routes=5)
rc.execute(perform_assignment=True)
```

## 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<sup>1, 2</sup>, Page 170, 3.

## Choice set generation algorithms

The generation of choice sets for route choice algorithms is the most time-consuming step of most well-established route choice algorithms, and that's certainly the case in AequilibraE's implementation of the Path Size Logit.

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.

## Choice set construction algorithms

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.<sup>1</sup> 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.

<sup>1</sup> Camargo, P. V. de, and R. Imai. Map-Matching Large Streams of Vehicle GPS Data into Bespoke Networks (Submitted). Presented at the ATRF, Melbourne, 2024.

<sup>2</sup> Moss, J., P. V. de Camargo, C. de Freitas, and R. Imai. High-Performance Route Choice Set Generation on Large Networks (Submitted). Presented at the ATRF, Melbourne, 2024.

<sup>1</sup> Rieser-Schüssler, N., Balmer, M., & 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>

## 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.

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.

## 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.

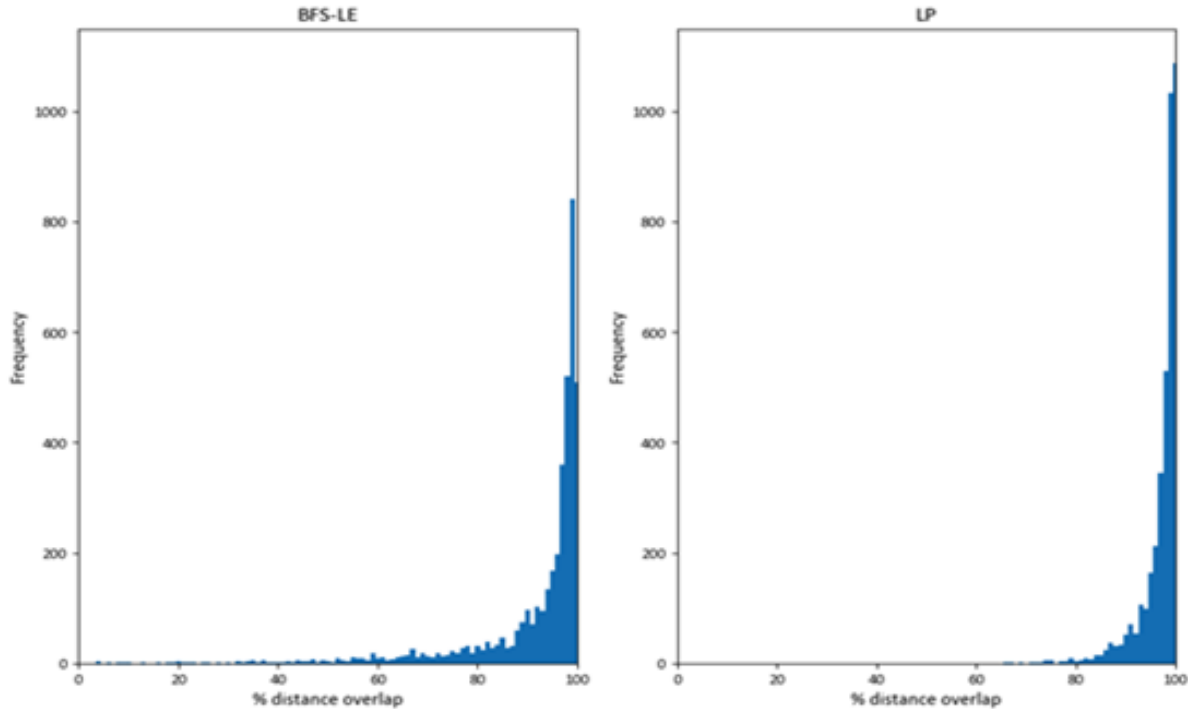
To efficiently store and determine the uniqueness of a new route or removed link sets, we used modified hash functions with properties that allowed us to store and nest them within standard C++ data structures. We used a commutative hash function for the removed link sets to allow for amortised  $O(1)$  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.<sup>[Page 171, 1](#)</sup>, which is also a reference for further details on the BFS-LE algorithm.

## 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.



## Code example

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

fldr = join(gettempdir(), uuid4().hex)
project = create_example(fldr, "coquimbo")

project.network.build_graphs()
graph = project.network.graphs["c"]
graph.set_graph("free_flow_time")

graph.prepare_graph(np.array([1, 2, 3, 50, 100, 150]))

nodes = [(1, 50), (2, 100), (3, 150)] # List of tuples with (origin, destination) nodes
max_routes = 10 # Maximum number of routes to be computed for each OD pair
penalty = 1.01 # Penalty to be applied to links used in paths.
cores = 60 # Number of threads to be used in the computation
```

(continues on next page)

(continued from previous page)

```

psl = True # If True, the path size logit will be used to compute probabilities already
# This is only useful if you are already using an utility measure to compute paths
bfsle = True # Should we use BFSLE? If False, defaults to Link Penalization

rc = RouteChoiceSet(graph) # Builds data structures -> can take a minute
rc.batched(nodes, max_routes=max_routes, cores=cores, bfsle=bfsle, penalty=penalty, path_
    size_logit=psl)

results = rc.get_results().to_pandas()
results.to_parquet(Path("/my_choice_set.parquet"))

```

## References

### Route choice models

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<sup>1</sup>. 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<sup>2</sup> 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

### Path-Size Logit (PSL)

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,  $\varepsilon_i$  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  $\gamma$  can be theoretically derived by aggregation of alternatives under certain assumptions, see<sup>3</sup> and references therein.

#### Note

**AequilibraE uses cost to compute path overlaps rather than distance**

<sup>1</sup> Ben-Akiva, M., and S. Lerman. Discrete Choice Analysis. The MIT Press, 1985.

<sup>2</sup> Ramming, M. S. Network Knowledge and Route Choice. Massachusetts Institute of Technology, 2002.

<sup>3</sup> Frejinger, E. (2008) Route Choice Analysis : Data , Models , Algorithms and Applications.

## **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 filter 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.

## **References**





## API REFERENCE

`aequilibrae.setup()`  
`aequilibrae.cleaning()`

### 3.1 Project

---

*Project*

AequilibraE project class

---

#### 3.1.1 `aequilibrae.project.Project`

**class** `aequilibrae.project.Project`  
AequilibraE project class

Listing 1: Create Project

```
>>> newfile = Project()
>>> newfile.new('/tmp/new_project')
```

Listing 2: Open Project

```
>>> from aequilibrae.project import Project

>>> existing = Project()
>>> existing.open('/tmp/test_project')

>>> #Let's check some of the project's properties
>>> existing.network.list_modes()
['M', 'T', 'b', 'c', 't', 'w']
>>> existing.network.count_links()
76
>>> existing.network.count_nodes()
24
```

`__init__()`

## Methods

<code>__init__()</code>	
<code>activate()</code>	
<code>check_file_indices()</code>	Makes results_database.sqlite and the matrices folder compatible with project database
<code>close()</code>	Safely closes the project
<code>connect()</code>	
<code>deactivate()</code>	
<code>from_path(project_folder)</code>	
<code>load(project_path)</code>	Loads project from disk
<code>log()</code>	Returns a log object
<code>new(project_path)</code>	Creates a new project
<code>open(project_path)</code>	Loads project from disk

## Attributes

<code>parameters</code>
<code>project_parameters</code>
<code>zoning</code>

**classmethod** `from_path(project_folder)`

`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.

**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

`close()` → None

Safely closes the project

**load**(project\_path: str) → None

Loads project from disk

Deprecated since version 0.7.0: Use `open()` instead.

**Arguments**

**project\_path** (str): Full path to the project data folder. If the project inside does not exist, it will fail.

**connect()**

**activate()**

**deactivate()**

**log()** → *Log*

Returns a log object

allows the user to read the log or clear it

**property project\_parameters:** *Parameters*

**property parameters:** dict

**check\_file\_indices()** → None

Makes results\_database.sqlite and the matrices folder compatible with project database

**property zoning**

### 3.1.2 Project Components

<i>About</i>	Provides an interface for querying and editing the <b>about</b> table of an AequilibraE project
<i>FieldEditor</i>	Allows user to edit the project data tables
<i>Log</i>	API entry point to the log file contents
<i>Matrices</i>	Gateway into the matrices available/recorded in the model
<i>Network</i>	Network class.
<i>Zoning</i>	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

```
>>> from aequilibrae import Project

>>> project = Project.from_path("/tmp/test_project")

# 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

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

### `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

### `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)

```
>>> from aequilibrae import Project

>>> p = Project.from_path("/tmp/test_project")
>>> p.about.add_info_field('a_cool_field')
>>> p.about.a_cool_field = 'super relevant information'
>>> p.about.write_back()
```

### `write_back()`

Saves the information parameters back to the project database

```
>>> from aequilibrae import Project

>>> p = Project.from_path("/tmp/test_project")
>>> p.about.description = 'This is the example project. Do not use for forecast'
>>> p.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:

```
>>> from aequilibrae import Project

>>> proj = Project.from_path("/tmp/test_project")

# To edit the fields of the link_types table
>>> lt_fields = proj.network.link_types.fields

# To edit the fields of the modes table
>>> m_fields = proj.network.modes.fields
```

Field descriptions are kept in the table *attributes\_documentation*

**\_\_init\_\_**(*project*, *table\_name*: *str*) → None

**Methods**

<b>__init__</b> ( <i>project</i> , <i>table_name</i> )	
<b>add</b> ( <i>field_name</i> , <i>description</i> [, <i>data_type</i> ])	Adds new field to the data table
<b>all_fields</b> ()	Returns the list of fields available in the database
<b>remove</b> ( <i>field_name</i> )	
<b>save</b> ()	Saves any field descriptions which may have been changed to the database

**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"

**remove**(*field\_name*: *str*) → None

**save**() → None

Saves any field descriptions which may have been changed to the database

**all\_fields**() → List[*str*]

Returns the list of fields available in the database

## aequilibrae.project.Log

**class** aequilibrae.project.Log(*project\_base\_path: str*)

API entry point to the log file contents

```
>>> from aequilibrae import Project

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

>>> 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: str*)

### Methods

<code>__init__</code> ( <i>project_base_path</i> )	
<code>clear</code> ()	Clears the log file.
<code>contents</code> ()	Returns contents of log file

**contents**() → list

Returns contents of log file

#### Returns

**log\_contents** (list): List with all entries in the log file

**clear**()

Clears the log file. Use it wisely

## aequilibrae.project.Matrices

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

Gateway into the matrices available/recorded in the model

**\_\_init\_\_**(*project*)

## Methods

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

### **reload()**

Discards all memory matrices in memory and loads recreate them

### **clear\_database()** → None

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

### **update\_database()** → None

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

### **list()** → DataFrame

List of all matrices available

#### **Returns**

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

### **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

### **check\_exists(name: str)** → bool

Checks whether a matrix with a given name exists

#### **Returns**

**exists** (bool): Does the matrix exist?

**delete\_record**(*matrix\_name: str*) → None

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

**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

## aequilibrae.project.Network

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

Network class. Member of an AequilibraE Project

**\_\_init\_\_**(*project*) → None

### Methods

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



## Attributes

*link\_types*

*netsignal*

*protected\_fields*

*req\_link\_flds*

*req\_node\_flds*

```
netsignal = <aequilibrae.utils.python_signal.PythonSignal object>
```

```
req_link_flds = ['link_id', 'a_node', 'b_node', 'direction', 'distance', 'modes', 'link_type']
```

```
req_node_flds = ['node_id', 'is_centroid']
```

```
protected_fields = ['ogc_fid', 'geometry']
```

```
link_types: LinkTypes = None
```

```
skimmable_fields()
```

Returns a list of all fields that can be skimmed

### Returns

list: List of all fields that can be skimmed

```
list_modes()
```

Returns a list of all the modes in this model

### Returns

list: List of all modes

```
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 Open-Street Maps

### 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

```
>>> from aequilibrae import Project
```

```
>>> p = Project()
```

(continues on next page)

(continued from previous page)

```

>>> p.new("/tmp/new_project")

# We now choose a different overpass endpoint (say a deployment in your local
↳network)
>>> par = Parameters()
>>> par.parameters['osm']['overpass_endpoint'] = "http://192.168.1.234:5678/api"

# Because we have our own server, we can set a bigger area for download (in M2)
>>> par.parameters['osm']['max_query_area_size'] = 10000000000

# And have no pause between successive queries
>>> par.parameters['osm']['sleeptime'] = 0

# Save the parameters to disk
>>> par.write_back()

# Now we can import the network for any place we want
# p.network.create_from_osm(place_name="my_beautiful_hometown")

>>> p.close()

```

**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

**export\_to\_gmns**(*path: str*)

Exports AequilibraE network to csv files in GMNS format.

#### Arguments

**path** (str): Output folder path.

**build\_graphs**(*fields: list | None = None, modes: list | 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

To use the *fields* parameter, a minimalistic option is the following

```
>>> from aequilibrae import Project

>>> p = Project.from_path("/tmp/test_project")
>>> fields = ['distance']
>>> p.network.build_graphs(fields, modes = ['c', 'w'])
```

**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

**count\_links**() → int

Returns the number of links in the model

**Returns**

int: Number of links

**count\_centroids**() → int

Returns the number of centroids in the model

**Returns**

int: Number of centroids

**count\_nodes**() → int

Returns the number of nodes in the model

**Returns**

int: Number of nodes

**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.

**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.

### aequilibrae.project.Zoning

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

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

```
>>> from aequilibrae import Project

>>> project = Project.from_path("/tmp/test_project")

>>> zoning = project.zoning

>>> zone_downtown = zoning.get(1)
>>> zone_downtown.population = 637
```

(continues on next page)

(continued from previous page)

```

>>> zone_downtown.employment = 10039
>>> zone_downtown.save()

# changing the value for an existing value/field
>>> project.about.scenario_name = 'Just a better scenario name'
>>> project.about.write_back()

# 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

<code>__init__(network)</code>	
<code>all_zones()</code>	Returns a dictionary with all Zone objects available in the model.
<code>coverage()</code>	Returns a single polygon for the entire zoning coverage
<code>create_zoning_layer()</code>	Creates the 'zones' table for project files that did not previously contain it
<code>extent()</code>	Queries the extent of the layer included in the model
<code>get(zone_id)</code>	Get a zone from the model by its <b>zone_id</b>
<code>get_closest_zone(geometry)</code>	Returns the zone in which the given geometry is located.
<code>new(zone_id)</code>	Creates a new zone
<code>refresh_geo_index()</code>	
<code>save()</code>	

## Attributes

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

**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)

**create\_zoning\_layer**()

Creates the 'zones' table for project files that did not previously contain it

**coverage()** → Polygon

Returns a single polygon for the entire zoning coverage

**Returns**

**model coverage** (Polygon): Shapely (Multi)polygon of the zoning system.

**get(zone\_id: str)** → *Zone*

Get a zone from the model by its **zone\_id**

**all\_zones()** → dict

Returns a dictionary with all Zone objects available in the model. *zone\_id* as key

**save()**

**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

**refresh\_geo\_index()**

**property data:** DataFrame

Returns all zones data as a Pandas DataFrame

**Returns**

**table** (DataFrame): Pandas DataFrame with all the zones, complete with Geometry

**extent()** → Polygon

Queries the extent of the layer included in the model

**Returns**

**model extent** (Polygon): Shapely polygon with the bounding box of the layer.

**property fields:** *FieldEditor*

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

### 3.1.3 Project Objects

<i>Zone</i>	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**

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

**delete()**

Removes the zone from the database

**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*) → 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 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 zone, the search area is 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

**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

**connect\_db**()

## 3.2 Network Data

<i>Modes</i>	Access to the API resources to manipulate the modes table in the network
<i>LinkTypes</i>	Access to the API resources to manipulate the link_types table in the network.
<i>Links</i>	Access to the API resources to manipulate the links table in the network
<i>Nodes</i>	Access to the API resources to manipulate the links table in the network
<i>Periods</i>	Access to the API resources to manipulate the links table in the network

### 3.2.1 aequilibrae.project.network.Modes

**class** aequilibrae.project.network.Modes(*net*)

Access to the API resources to manipulate the modes table in the network

```
>>> from aequilibrae import Project

>>> p = Project.from_path("/tmp/test_project")

>>> modes = p.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'
```

(continues on next page)

(continued from previous page)

```

# 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

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

## Attributes

<code>fields</code>	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

**delete**(*mode\_id*: str) → None

Removes the mode with *mode\_id* from the project

**property fields:** FieldEditor

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



**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*

**all\_modes**() → dict

Returns a dictionary with all mode objects available in the model. *mode\_id* as key

**new**(*mode\_id: str*) → *Mode*

Returns a new mode with *mode\_id* that can be added to the model later

### 3.2.2 aequilibrae.project.network.LinkTypes

**class** aequilibrae.project.network.LinkTypes(*net*)

Access to the API resources to manipulate the link\_types table in the network.

```
>>> from aequilibrae import Project

>>> p = Project.from_path("/tmp/test_project")

>>> link_types = p.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
```

(continues on next page)

(continued from previous page)

```
# 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

<code>__init__(net)</code>	
<code>all_types()</code>	Returns a dictionary with all LinkType objects available in the model.
<code>delete(link_type_id)</code>	Removes the link_type with <i>link_type_id</i> from the project
<code>fields()</code>	Returns a FieldEditor class instance to edit the Link_Types table fields and their metadata
<code>get(link_type_id)</code>	Get a link_type from the network by its <i>link_type_id</i>
<code>get_by_name(link_type)</code>	Get a link_type from the network by its <i>link_type</i> (i.e. name).
<code>new(link_type_id)</code>	
<code>save()</code>	

**new**(*link\_type\_id: str*) → *LinkType*

**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)

**fields**() → *FieldEditor*

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

**save**()

### 3.2.3 aequilibrae.project.network.Links

**class** aequilibrae.project.network.Links(*net*)

Access to the API resources to manipulate the links table in the network

```
>>> from aequilibrae import Project

>>> proj = Project.from_path("/tmp/test_project")

>>> all_links = proj.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

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

#### Attributes

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

**sql** = ''

Query sql for retrieving links

**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)**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) → NoneRemoves the link with **link\_id** from the project**Arguments****link\_id** (int): Id of a link to delete**refresh\_fields**() → None

After adding a field one needs to refresh all the fields recognized by the software

**property data:** **DataFrame**

Returns all links data as a Pandas DataFrame

**Returns****table** (DataFrame): Pandas dataframe with all the links, complete with Geometry**refresh**()

Refreshes all the links in memory

**save**()**extent**() → Polygon

Queries the extent of the layer included in the model

**Returns****model extent** (Polygon): Shapely polygon with the bounding box of the layer.**property fields:** *FieldEditor*

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

### 3.2.4 aequilibrae.project.network.Nodes

**class** aequilibrae.project.network.Nodes(*net*)

Access to the API resources to manipulate the links table in the network

```
>>> from aequilibrae import Project

>>> proj = Project.from_path("/tmp/test_project")

>>> all_nodes = proj.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

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

#### Attributes

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

**sql** = ''

Query sql for retrieving nodes

**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**refresh\_fields()** → None

After adding a field one needs to refresh all the fields recognized by the software

**refresh()**

Refreshes all the nodes in memory

**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**save()****property data:** **DataFrame**

Returns all nodes data as a Pandas DataFrame

**Returns****table** (DataFrame): Pandas DataFrame with all the nodes, complete with Geometry**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**extent()** → Polygon

Queries the extent of the layer included in the model

**Returns****model extent** (Polygon): Shapely polygon with the bounding box of the layer.**property fields:** *FieldEditor*

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

### 3.2.5 aequibrae.project.network.Periods

**class** aequibrae.project.network.**Periods**(net)

Access to the API resources to manipulate the links table in the network

```
>>> from aequibrae import Project
>>> proj = Project.from_path("/tmp/test_project")
>>> all_periods = proj.network.periods

# We can just get one link in specific
>>> period = all_periods.get(21)

# We can save changes for all periods we have edited so far
>>> all_periods.save()
```

`__init__(net)`

## Methods

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

## Attributes

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

`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

`refresh_fields() → None`

After adding a field one needs to refresh all the fields recognized by the software

`refresh()`

Refreshes all the periods in memory

**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’

**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

## 3.3 Network Items

<i>Mode</i>	A mode object represents a single record in the <i>modes</i> table
<i>LinkType</i>	A link_type object represents a single record in the <i>link_types</i> table
<i>Link</i>	A Link object represents a single record in the <i>links</i> table
<i>Node</i>	A Node object represents a single record in the <i>nodes</i> table
<i>Period</i>	A Period object represents a single record in the <i>periods</i> table

### 3.3.1 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()**

### 3.3.2 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)
connect_db()
delete()
save()
```

**delete()**

**save()**

**connect\_db()**

### 3.3.3 aequilibrae.project.network.Link

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

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

```
>>> from aequilibrae import Project
>>> proj = Project.from_path("/tmp/test_project")
>>> all_links = proj.network.links

# Let's get a mode to work with
>>> modes = proj.network.modes
>>> car_mode = modes.get('c')
```

(continues on next page)

(continued from previous page)

```

# 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

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

### **delete()**

Deletes link from database

### **save()**

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

**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

**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

**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

**connect\_db**()

### 3.3.4 aequilibrae.project.network.Node

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

A Node object represents a single record in the *nodes* table

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

>>> proj = Project.from_path("/tmp/test_project")

>>> all_nodes = proj.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

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

### **save()**

Saves node to database

### **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

### **connect\_mode**(*area*: Polygon, *mode\_id*: str, *link\_types*="", *connectors*=1, *conn*: Connection | 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**

**area** (Polygon): Initial area where AequilibraE will look for nodes to connect

**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

### **connect\_db()**

### 3.3.5 aequilibrae.project.network.Period

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

A Period object represents a single record in the *periods* table

```
>>> from aequilibrae import Project

>>> proj = Project.from_path("/tmp/test_project")

>>> all_periods = proj.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()

# It succeeds if the period_id already does not exist
>>> period1.renumber(998877)

# We can just save the period
>>> period1.save()
```

**\_\_init\_\_**(*dataset, project*)

#### Methods

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

**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*

**connect\_db()**

## 3.4 Parameters

Parameters

Global parameters module

### 3.4.1 aequilibrae.Parameters

**class** aequilibrae.Parameters(*project=None*)

Global parameters module

Parameters are used in many procedures, and are often defined only in the parameters.yml file ONLY Parameters are organized in the following groups:

- assignment
- distribution
- system
- report zeros
- temp directory

```
>>> from aequilibrae import Project, Parameters

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

>>> p = Parameters(project)

>>> p.parameters['system']['logging_directory'] = "/tmp/other_folder"
>>> p.parameters['osm']['overpass_endpoint'] = "http://192.168.0.110:32780/api"
>>> p.parameters['osm']['max_query_area_size'] = 100000000000
>>> p.parameters['osm']['sleep_time'] = 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

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

## Attributes

*file\_default*

**file\_default:** str = '/opt/hostedtoolcache/Python/3.10.14/x64/lib/python3.10/site-packages/aequilibrae/parameters.yml'

**write\_back()**

Writes the parameters back to file

**restore\_default()**

Restores parameters to generic default

## 3.5 Distribution

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

### 3.5.1 aequilibrae.distribution.Ipf

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

Iterative proportional fitting procedure

```
>>> from aequilibrae import Project
>>> from aequilibrae.distribution import Ipf
>>> from aequilibrae.matrix import AequilibraeMatrix, AequilibraeData

>>> project = Project.from_path("/tmp/test_project_ipf")

>>> matrix = AequilibraeMatrix()

# Here we can create from OMX or load from an AequilibraE matrix.
>>> matrix.load('/tmp/test_project/matrices/demand.omx')
>>> matrix.computational_view()

>>> args = {"entries": matrix.zones, "field_names": ["productions", "attractions"],
...        "data_types": [np.float64, np.float64], "memory_mode": True}

>>> vectors = AequilibraeData()
>>> vectors.create_empty(**args)

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

# We assume that the indices would be sorted and that they would match the matrix_
```

(continues on next page)

(continued from previous page)

```

→indices
>>> vectors.index[:] = matrix.index[:]

>>> args = {
...     "matrix": matrix, "rows": vectors, "row_field": "productions", "columns
→": vectors,
...     "column_field": "attractions", "nan_as_zero": False}

>>> fratar = Ipf(**args)

>>> fratar.fit()

# We can get back to our OMX matrix in the end
>>> fratar.output.export("/tmp/to_omx_output.omx")
>>> fratar.output.export("/tmp/to_aem_output.aem")

```

**\_\_init\_\_**(*project=None, \*\*kwargs*)

Instantiates the IPF problem

#### Arguments

**matrix** (AequilibraeMatrix): Seed Matrix

**rows** (AequilibraeData): Vector object with data for row totals

**row\_field** (str): Field name that contains the data for the row totals

**columns** (AequilibraeData): Vector object with data for column 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

<code>__init__</code> ([project])	Instantiates the IPF problem
<code>fit</code> ()	Runs the IPF instance problem to adjust the matrix
<code>save_to_project</code> (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**3.5.2 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. Row and Column vectors are instances of AequilibraeData.

```

>>> import pandas as pd
>>> from aequilibrae import Project
>>> from aequilibrae.matrix import AequilibraeMatrix, AequilibraeData
>>> from aequilibrae.distribution import SyntheticGravityModel, GravityApplication

>>> project = Project.from_path("/tmp/test_project_ga")

# We define the model we will use
>>> model = SyntheticGravityModel()

# Before adding a parameter to the model, you need to define the model functional_
↳ form
>>> model.function = "GAMMA" # "EXPO" or "POWER"

# Only the parameter(s) applicable to the chosen functional form will have any_
↳ effect
>>> model.alpha = 0.1
>>> model.beta = 0.0001

# Or you can load the model from a file
# model.load('path/to/model/file')

# We load the impedance matrix
>>> matrix = AequilibraeMatrix()
>>> matrix.load('/tmp/test_project_ga/matrices/skims.omx')
>>> matrix.computational_view(['distance_blended'])

# We create the vectors we will use
>>> query = "SELECT zone_id, population, employment FROM zones;"
>>> df = pd.read_sql(query, project.conn)
>>> df.sort_values(by="zone_id", inplace=True)

# You create the vectors you would have
>>> df = df.assign(production=df.population * 3.0)
>>> df = df.assign(attraction=df.employment * 4.0)

>>> zones = df.index.shape[0]

```

(continues on next page)

(continued from previous page)

```

# We create the vector database
>>> args = {"entries": zones, "field_names": ["productions", "attractions"],
...         "data_types": [np.float64, np.float64], "memory_mode": True}
>>> vectors = AequilibraeData()
>>> vectors.create_empty(**args)

# Assign the data to the vector object
>>> vectors.productions[:] = df.production.values[:]
>>> vectors.attractions[:] = df.attraction.values[:]
>>> vectors.index[:] = df.zone_id.values[:]

# Balance the vectors
>>> vectors.attractions[:] *= vectors.productions.sum() / vectors.attractions.sum()

# Create the problem object
>>> args = {"impedance": matrix,
...         "rows": vectors,
...         "row_field": "productions",
...         "model": model,
...         "columns": vectors,
...         "column_field": "attractions",
...         "output": '/tmp/test_project_ga/matrices/matrix.aem',
...         "nan_as_zero": True
...         }
>>> gravity = GravityApplication(**args)

# Solve and save the outputs
>>> gravity.apply()
>>> gravity.output.export('/tmp/test_project_ga/matrices/omx_file.omx')

# To save your report into a file, you can do the following:
# with open('/tmp/test_project_ga/report.txt', 'w') as file:
#     for line in gravity.report:
#         file.write(f"{line}\n")

```

**\_\_init\_\_**(*project=None, \*\*kwargs*)

Instantiates the IPF problem

#### Arguments

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

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

**rows** (*AequilibraeData*): Vector object with data for row totals

**row\_field** (*str*): Field name that contains the data for the row totals

**columns** (*AequilibraeData*): Vector object with data for column 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

<code>__init__([project])</code>	Instantiates the IPF problem
<code>apply()</code>	Runs the Gravity Application instance as instantiated
<code>save_to_project(name, file_name[, project])</code>	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

### 3.5.3 aequilibrae.distribution.GravityCalibration

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

Calibrate a traditional gravity model

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

```
>>> from aequilibrae import Project
>>> from aequilibrae.matrix import AequilibraeMatrix
>>> from aequilibrae.distribution import GravityCalibration

>>> project = Project.from_path("/tmp/test_project_gc")

# We load the impedance matrix
>>> matrix = AequilibraeMatrix()
>>> matrix.load('/tmp/test_project_gc/matrices/demand.omx')
>>> matrix.computational_view(['matrix'])

# We load the impedance matrix
>>> impedmatrix = AequilibraeMatrix()
```

(continues on next page)

(continued from previous page)

```

>>> impedmatrix.load('/tmp/test_project_gc/matrices/skims.omx')
>>> impedmatrix.computational_view(['time_final'])

# Creates the problem
>>> args = {"matrix": matrix,
...         "impedance": impedmatrix,
...         "row_field": "productions",
...         "function": 'expo',
...         "nan_as_zero": True}
>>> gravity = GravityCalibration(**args)

# Solve and save outputs
>>> gravity.calibrate()
>>> gravity.model.save('/tmp/test_project_gc/dist_expo_model.mod')

# To save the model report in a file
# with open('/tmp/test_project_gc/report.txt', 'w') as f:
#     for line in gravity.report:
#         f.write(f'{line}\n')

```

**\_\_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

<code>__init__([project])</code>	Instantiates the Gravity calibration problem
<code>calibrate()</code>	Calibrate the model

### calibrate()

Calibrate the model

Resulting model is in *output* class member

### 3.5.4 aequilibrae.distribution.SyntheticGravityModel

**class** aequilibrae.distribution.SyntheticGravityModel

Simple class object to represent synthetic gravity models

`__init__()`

#### Methods

<code>__init__()</code>	
<code>load(file_name)</code>	Loads model from disk.
<code>save(file_name)</code>	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

## 3.6 Matrix

<i>AequilibraeData</i>	AequilibraE dataset
<i>AequilibraeMatrix</i>	Matrix class

### 3.6.1 aequilibrae.matrix.AequilibraeData

**class** aequilibrae.matrix.AequilibraeData

AequilibraE dataset

`__init__()`

#### Methods

<code>__init__()</code>	
<code>create_empty([file_path, entries, ...])</code>	Creates a new empty dataset
<code>empty(*args, **kwargs)</code>	
<code>export(file_name[, table_name])</code>	Exports the dataset to another format.
<code>load(file_path)</code>	Loads dataset from file
<code>random_name()</code>	Returns a random name for a dataset with root in the temp directory of the user

**classmethod** `empty(*args, **kwargs)`

**create\_empty**(*file\_path=None, entries=1, field\_names=None, data\_types=None, memory\_mode=False, fill=None, index=None*)

Creates a new empty dataset

**Arguments**

**file\_path** (str, *Optional*): Full path for the output data file. If *memory\_mode* is 'false' and path is missing, then the file is created in the temp folder

**entries** (int, *Optional*): Number of records in the dataset. Default is 1

**field\_names** (list, *Optional*): List of field names for this dataset. If no list is provided, the field 'data' will be created

**data\_types** (np.dtype, *Optional*): List of data types for the dataset. Types need to be NumPy data types (e.g. np.int16, np.float64). If no list of types are provided, type will be np.float64

**memory\_mode** (bool, *Optional*): If True, dataset will be kept in memory. If False, the dataset will be a memory-mapped numpy array

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

>>> mat = AequilibraeMatrix()
>>> mat.load('/tmp/test_project/matrices/demand.omx')
>>> mat.computational_view()

>>> vectors = "/tmp/test_project/vectors.aed"

>>> args = {
...     "file_path": vectors,
...     "entries": mat.zones,
...     "field_names": ["origins", "destinations"],
...     "data_types": [np.float64, np.float64]
... }

>>> dataset = AequilibraeData()
>>> dataset.create_empty(**args)
```

**load**(*file\_path*)

Loads dataset from file

**Arguments**

**file\_path** (str): Full file path to the AequilibraeData to be loaded

```
>>> from aequilibrae.matrix import AequilibraeData

>>> dataset = AequilibraeData()
>>> dataset.load("/tmp/test_project/vectors.aed")
```

**export**(*file\_name, table\_name='aequilibrae\_table'*)

Exports the dataset to another format. Supports CSV and SQLite

**Arguments**

**file\_name** (str): File name with PATH and extension (csv, or sqlite3, sqlite or db)

**table\_name** (str): It only applies if you are saving to an SQLite table. Otherwise ignored

```
>>> from aequilibrae.matrix import AequilibraeData

>>> dataset = AequilibraeData()
>>> dataset.load("/tmp/test_project/vectors.aed")
>>> dataset.export("/tmp/test_project/vectors.csv")
```

**static random\_name()**

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

```
>>> from aequilibrae.matrix import AequilibraeData

>>> name = AequilibraeData().random_name()

# This is an example of output
# '/tmp/Aequilibrae_data_5werr5f36-b123-asdf-4587-adfglkjhqwe.aed'
```

### 3.6.2 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

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

**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

**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

```
>>> 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='/tmp/path_to_matrix.aem',
...                  zones=zones_in_the_model,
...                  matrix_names=names_list,
...                  memory_only=False,)
>>> mat.num_indices
1
>>> mat.zones
3317
```

**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

**create\_from\_omx**(file\_path: str, omx\_path: str, 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

**file\_path** (str): Path for the output AequilibraE Matrix

**omx\_path** (str): Path to the OMX file one wants to import

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

**mappings** (list): 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

**from\_column** (str): trip list file column containing the destination zones numbers

**list\_cores** (list): list of core columns in the trip list file

**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="/tmp/path_to_new_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'
```

**close()**

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

**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='/tmp/path_to_matrix.aem',
...                  zones=zones_in_the_model,
...                  matrix_names=names_list)
>>> mat.cores
5
>>> mat.export('/tmp/my_new_path.aem', ['Car trips', 'bike trips'])

>>> mat2 = AequilibraeMatrix()
>>> mat2.load('/tmp/my_new_path.aem')
>>> mat2.cores
2

```

**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

>>> mat = AequilibraeMatrix()
>>> mat.load('/tmp/path_to_matrix.aem')
>>> mat.computational_view(["bike trips"])
>>> mat.names
['Car trips', 'pt trips', 'DRT trips', 'bike trips', 'walk trips']

```

**is\_omx()**

Returns True if matrix data source is OMX, False otherwise

**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']

```

(continues on next page)

(continued from previous page)

```

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name='/tmp/path_to_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

```

>>> 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='/tmp/path_to_matrix.aem', zones=zones_in_the_
↳model, matrix_names= names_list)
>>> mat.copy('/tmp/path_to_copy.aem',
...          cores=['bike trips', 'walk trips'],
...          names=['bicycle', 'walking'],
...          memory_only=False)
<aequilibrae.matrix.aequilibrae_matrix.AequilibraeMatrix object at 0x...>

>>> mat2 = AequilibraeMatrix()
>>> mat2.load('/tmp/path_to_copy.aem')
>>> mat2.cores
2

```

**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

>>> mat = AequilibraeMatrix()
>>> mat.load('/tmp/test_project/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])
```

**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

>>> mat = AequilibraeMatrix()
>>> mat.load('/tmp/test_project/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 ])
```

**nan\_to\_num()**

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

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

>>> mat = AequilibraeMatrix()
>>> mat.load('/tmp/path_to_matrix.aem')
>>> mat.computational_view(["bike trips"])
>>> mat.nan_to_num()
```

**setName(matrix\_name: str)**

Sets the name for the matrix itself

#### Arguments

**matrix\_name** (str): matrix name. Maximum length is 50 characters

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

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name="matrix.aem", zones=3317, memory_only=False)
```

(continues on next page)

(continued from previous page)

```
>>> mat.setName('This is my example')
>>> mat.name
''
```

**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

>>> mat = AequilibraeMatrix()
>>> mat.create_empty(file_name="matrix.aem", zones=3317, memory_only=False)
>>> mat.setDescription('This is some text about this matrix of mine')
>>> mat.save()
>>> mat.close()

>>> mat = AequilibraeMatrix()
>>> mat.load("matrix.aem")
>>> mat.description.decode('utf-8')
'This is some text ab'
```

**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

>>> name = AequilibraeMatrix().random_name()

# This is an example of output
# '/tmp/Aequilibrae_matrix_54625f36-bf41-4c85-80fb-7fc2e3f3d76e.aem'
```

## 3.7 Paths

<i>Graph</i>	
<i>TransitGraph</i>	
<i>AssignmentResults</i>	Assignment result holder for a single <i>TrafficClass</i> with multiple user classes
<i>TransitAssignmentResults</i>	Assignment result holder for a single <i>Transit</i>
<i>SkimResults</i>	Network skimming result holder.
<i>PathResults</i>	Path computation result holder
<i>VDF</i>	Volume-Delay function
<i>TrafficClass</i>	Traffic class for equilibrium traffic assignment
<i>TransitClass</i>	
<i>TrafficAssignment</i>	Traffic assignment class.
<i>TransitAssignment</i>	
<i>HyperpathGenerating</i>	A class for hyperpath generation.
<i>OptimalStrategies</i>	

### 3.7.1 aequilibrae.paths.Graph

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

**\_\_init\_\_**(\*args, \*\*kwargs)

#### Methods

<b>__init__</b> (*args, **kwargs)	
<i>available_skims</i> ()	Returns graph fields that are available to be set as skims
<i>create_compressed_link_network_mapping</i> ()	Create two arrays providing a mapping of compressed id to link id.
<i>default_types</i> (tp)	Returns the default integer and float types used for computation
<i>exclude_links</i> (links)	Excludes a list of links from a graph by setting their B node equal to their A node
<i>load_from_disk</i> (filename)	Loads graph from disk
<i>prepare_graph</i> (centroids)	Prepares the graph for a computation for a certain set of centroids
<i>save_compressed_correspondence</i> (path, ...)	Save graph and nodes_to_indices to disk
<i>save_to_disk</i> (filename)	Saves graph to disk
<i>set_blocked_centroid_flows</i> (block_centroid_flow)	Chooses whether we want to block paths to go through centroids or not.
<i>set_graph</i> (cost_field)	Sets the field to be used for path computation
<i>set_skimming</i> (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

**create\_compressed\_link\_network\_mapping()**

Create two 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]`.

```
>>> idx, data = graph.compressed_link_network_mapping
>>> data[idx[id]:idx[id + 1]] # ==> Slice of network ID's corresponding to the_
↳ compressed ID
```

Links not in the compressed graph are not contained within the data array.

**Returns**

**idx** (np.array): index array for data **data** (np.array): array of link 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

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

**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

### 3.7.2 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

<code>__init__([config, od_node_mapping])</code>	
<code>available_skims()</code>	Returns graph fields that are available to be set as skims
<code>create_compressed_link_network_mapping()</code>	Create two arrays providing a mapping of compressed id to link id.
<code>default_types(tp)</code>	Returns the default integer and float types used for computation
<code>exclude_links(links)</code>	Excludes a list of links from a graph by setting their B node equal to their A node
<code>load_from_disk(filename)</code>	Loads graph from disk
<code>prepare_graph(centroids)</code>	Prepares the graph for a computation for a certain set of centroids
<code>save_compressed_correspondence(path, ...)</code>	Save graph and nodes_to_indices to disk
<code>save_to_disk(filename)</code>	Saves graph to disk
<code>set_blocked_centroid_flows(block_centroid_flow)</code>	Chooses whether we want to block paths to go through centroids or not.
<code>set_graph(cost_field)</code>	Sets the field to be used for path computation
<code>set_skimming(skim_fields)</code>	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

**create\_compressed\_link\_network\_mapping()**

Create two 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]`.

```
>>> idx, data = graph.compressed_link_network_mapping
>>> data[idx[id]:idx[id + 1]] # ==> Slice of network ID's corresponding to the_
↳ compressed ID
```

Links not in the compressed graph are not contained within the `data` array.

**Returns**

**idx** (np.array): index array for **data** **data** (np.array): array of link 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

Prepares the graph for a computation for a certain set of centroids

Under the hood, it 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

**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

### 3.7.3 aequilibrae.paths.AssignmentResults

**class** aequilibrae.paths.AssignmentResults

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

**\_\_init\_\_**()

#### Methods

<code>__init__()</code>	
<code>get_graph_to_network_mapping()</code>	
<code>get_load_results()</code>	Translates the assignment results from the graph format into the network format
<code>get_sl_results()</code>	
<code>prepare(graph, matrix)</code>	Prepares the object with dimensions corresponding to the assignment matrix and graph objects
<code>reset()</code>	Resets object to prepared and pre-computation state
<code>save_to_disk([file_name, output])</code>	Function to write to disk all outputs computed during assignment.
<code>set_cores(cores)</code>	Sets number of cores (threads) to be used in computation
<code>total_flows()</code>	Totals all link flows for this class into a single link load

**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

**total\_flows()** → None

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

Results are placed into *total\_link\_loads* class member

**get\_graph\_to\_network\_mapping()**

**get\_load\_results()** → *AequilibraeData*

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

**Returns**

**dataset** (*AequilibraeData*): AequilibraE data with the traffic class assignment results

**get\_sl\_results()** → *AequilibraeData*

**save\_to\_disk**(*file\_name=None, output='loads'*) → None

Function to write to disk all outputs computed during assignment.

Deprecated since version 0.7.0.

**Arguments**

**file\_name** (*str*): Name of the file, with extension. Valid extensions are: ['aed', 'csv', 'sqlite']

**output** (*str, Optional*): Type of output ('loads', 'path\_file'). Defaults to 'loads'

**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

### 3.7.4 aequilibrae.paths.TransitAssignmentResults

**class** aequilibrae.paths.TransitAssignmentResults

Assignment result holder for a single Transit

**\_\_init\_\_**()

## Methods

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

**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

**get\_load\_results**() → [AequilibraeData](#)

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

### Returns

**dataset** ([AequilibraeData](#)): AequilibraE data with the transit class assignment results

**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

## 3.7.5 aequilibrae.paths.SkimResults

**class** `aequilibrae.paths.SkimResults`

Network skimming result holder.

```
>>> from aequilibrae import Project
>>> from aequilibrae.paths.results import SkimResults

>>> proj = Project.from_path("/tmp/test_project")
>>> proj.network.build_graphs()

# Mode c is car in this project
```

(continues on next page)

(continued from previous page)

```

>>> car_graph = proj.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('/tmp/test_project/matrix.aem')

```

```
__init__()
```

## Methods

<code>__init__()</code>	
<code>prepare(graph)</code>	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)

## 3.7.6 aequilibrae.paths.PathResults

**class** aequilibrae.paths.PathResults

Path computation result holder

```

>>> from aequilibrae import Project
>>> from aequilibrae.paths.results import PathResults

>>> proj = Project.from_path("/tmp/test_project")
>>> proj.network.build_graphs()

# Mode c is car in this project
>>> car_graph = proj.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'])

```

(continues on next page)

(continued from previous page)

```

>>> res = PathResults()
>>> res.prepare(car_graph)
>>> res.compute_path(1, 17)

# res.milepost contains the milepost corresponding to each node along the path
# res.path_nodes contains the sequence of nodes that form the path
# res.path contains the sequence of links that form the path
# res.path_link_directions contains the link directions corresponding to the above
# links
# res.skims contain all skims requested when preparing the graph

# 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

<code>__init__()</code>	
<code>compute_path(origin, destination[, ...])</code>	Computes the path between two nodes in the network.
<code>get_heuristics()</code>	Return the available heuristics.
<code>prepare(graph)</code>	Prepares the object with dimensions corresponding to the graph object
<code>reset()</code>	Resets object to prepared and pre-computation state
<code>set_heuristic(heuristic)</code>	Set the heuristics to be used in A*.
<code>update_trace(destination)</code>	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.

**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

**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**set\_heuristic**(*heuristic: str*) → NoneSet the heuristics to be used in A\*. Must be one of *get\_heuristics()*.**Arguments****heuristic** (str): Heuristic to use in A\*.**get\_heuristics()** → List[str]

Return the available heuristics.

### 3.7.7 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



### 3.7.8 aequilibrae.paths.TrafficClass

**class** aequilibrae.paths.TrafficClass(*name: str, graph: Graph, matrix: AequilibraeMatrix*)

Traffic class for equilibrium traffic assignment

```
>>> from aequilibrae import Project
>>> from aequilibrae.matrix import AequilibraeMatrix
>>> from aequilibrae.paths import TrafficClass

>>> project = Project.from_path("/tmp/test_project")
>>> 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 = AequilibraeMatrix()
>>> 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

<code>__init__(name, graph, matrix)</code>	Instantiates the class
<code>set_fixed_cost(field_name[, multiplier])</code>	Sets value of time
<code>set_pce(pce)</code>	Sets Passenger Car equivalent
<code>set_select_links(links)</code>	Set the selected links.
<code>set_vot(value_of_time)</code>	Sets value of time

## Attributes

*info*

**set\_pce**(*pce: float | int*) → None

Sets Passenger Car equivalent

### Arguments

**pce** (Union[float, int]): PCE. Defaults to 1 if not set

**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\_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

**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

**property info:** dict

### 3.7.9 aequilibrae.paths.TransitClass

**class** aequilibrae.paths.**TransitClass**(*name: str, graph: TransitGraph, matrix: AequilibraeMatrix, matrix\_core: str | None = None*)

**\_\_init\_\_**(*name: str, graph: TransitGraph, matrix: AequilibraeMatrix, matrix\_core: str | None = 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

<code>__init__(name, graph, matrix[, matrix_core])</code>	Instantiates the class
<code>set_demand_matrix_core(core)</code>	Set the matrix core to use for demand.

## Attributes

<code>info</code>
-------------------

property info: dict

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

### Arguments

`core (str):`

### 3.7.10 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 import Project
>>> from aequilibrae.matrix import AequilibraeMatrix
>>> from aequilibrae.paths import TrafficAssignment, TrafficClass

>>> project = Project.from_path("/tmp/test_project")
>>> 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 = AequilibraeMatrix()
>>> 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
>>> assignclass = TrafficClass("car", graph, demand)
```

(continues on next page)

(continued from previous page)

```

>>> 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')

# Since we haven't checked the parameters file, let's make sure convergence_
↪criteria is good
>>> assig.max_iter = 1000
>>> assig.rgap_target = 0.00001

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

# If you want, it is possible to access the convergence report
>>> import pandas as pd
>>> 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

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

## Attributes

<code>all_algorithms</code>
<code>bpr_parameters</code>

`bpr_parameters = ['alpha', 'beta']`

**all\_algorithms** = ['all-or-nothing', 'msa', 'frank-wolfe', 'fw', 'cfw', 'bfw']

**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\_classes**(*classes: List[TrafficClass]*) → None

Sets Traffic classes to be assigned

**Arguments**

**classes** (List[TrafficClass]): List of Traffic classes for assignment

**add\_class**(*traffic\_class: TrafficClass*) → None

Adds a traffic class to the assignment

**Arguments**

**traffic\_class** (*TrafficClass*): Traffic 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\_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

**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\_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\_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\_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\_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

**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)

**log\_specification**()

**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

**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*

**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

**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

**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. Note the traffic class

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

**save\_select\_link\_matrices**(*file\_name: str*) → None

Saves the Select Link matrices for each TrafficClass in the current TrafficAssignment class

**save\_select\_link\_results**(*name: str*) → None

Saves both the Select Link matrices and flow results at the same time, using the same name.

**Note**

Note the Select Link matrices will have `_SL_matrices.omx` appended to the end for ease of identification. e.g. `save_select_link_results("Car")` will result in the following names for the flows and matrices:  
Select Link Flows: inserts the select link flows for each class into the database with the table name: Car  
Select Link Matrices (only exports to OMX format): Car.omx

**Arguments**

**name** (str): name of the matrices

**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

**report()** → DataFrame

Returns the assignment convergence report

**Returns**

**DataFrame** (pd.DataFrame): Convergence report

### 3.7.11 aequilibrae.paths.TransitAssignment

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

**\_\_init\_\_**(\*args, project=None, \*\*kwargs)



## Methods

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

## Attributes

<code>all_algorithms</code>
-----------------------------

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

`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_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

`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()**

**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

**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*

**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

**set\_frequency\_field**(*frequency\_field: str*) → None

Sets the graph field that contains the frequency -> e.g. 'freq'

**Arguments**

**frequency\_field** (str): Field name

**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

**report()** → DataFrame

Returns the assignment convergence report

**Returns**

**DataFrame** (pd.DataFrame): Convergence report

**set\_classes**(*classes*: List[TransportClassBase]) → None

Sets Transport classes to be assigned

#### Arguments

**classes** (List[TransportClassBase]): List of TransportClass's for assignment

### 3.7.12 aequilibrae.paths.HyperpathGenerating

**class** aequilibrae.paths.**HyperpathGenerating**(*edges*, *tail*='tail', *head*='head', *trav\_time*='trav\_time', *freq*='freq', *check\_edges*=False)

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)

#### Methods

<code>__init__(edges[, tail, head, trav_time, ...])</code>	
<code>assign(origin_column, destination_column, ...)</code>	Assigns demand to the edges of the graph.
<code>info()</code>	
<code>run(origin, destination, volume[, return_inf])</code>	
<code>save_results(table_name[, keep_zero_flows, ...])</code>	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 sparse 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).

**info()** → dict

**run**(*origin, destination, volume, return\_inf=False*)

**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

### 3.7.13 aequilibrae.paths.OptimalStrategies

**class** aequilibrae.paths.OptimalStrategies(*assig\_spec*)

**\_\_init\_\_**(*assig\_spec*)

#### Methods

<code>__init__(assig_spec)</code>
<code>execute()</code>

**execute()**

## 3.8 Transit

<i>Transit</i>	
<i>TransitGraphBuilder</i>	Graph builder for the transit assignment Spiess & Florian algorithm.

### 3.8.1 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

<code>__init__(project)</code>	
<code>build_pt_preload(start, end[, inclusion_cond])</code>	Builds a preload vector for the transit network over the specified time period
<code>create_graph(**kwargs)</code>	
<code>create_transit_database()</code>	Creates the public transport database
<code>load([period_ids])</code>	
<code>new_gtfs_builder(agency, file_path[, day, ...])</code>	Returns a GTFSRouteSystemBuilder object compatible with the project
<code>save_graphs([period_ids])</code>	

#### Attributes

<code>default_capacities</code>
<code>default_pces</code>
<code>graphs</code>
<code>pt_con</code>

```
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] = {1:
<aequilibrae.transit.transit_graph_builder.TransitGraphBuilder object>}
```

```
pt_con: Connection
```

```
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

**create\_transit\_database()**

Creates the public transport database

**create\_graph**(\*\*kwargs) → *TransitGraphBuilder*

**save\_graphs**(period\_ids: List[int] | None = None)

**load**(period\_ids: List[int] | None = None)

**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

Minimal example: .. code-block:: python

```
>>> from aequilibrae import Project
>>> from aequilibrae.utils.create_example import create_example
```

```
>>> proj = create_example(str(tmp_path / "test_traffic_assignment"), from_model=
↳ "coquimbo")
```

```
>>> start = int(6.5 * 60 * 60) # 6.30am
>>> end = int(8.5 * 60 * 60)   # 8.30 am
```

```
>>> preload = proj.transit.build_pt_preload(start, end)
```

### 3.8.2 aequilibrae.transit.TransitGraphBuilder

```
class aequilibrae.transit.TransitGraphBuilder(public_transport_conn, period_id: int = 1, time_margin:
int = 0, projected_crs: str = 'EPSG:3857', num_threads:
int = -1, seed: int = 124, geometry_noise: bool = True,
noise_coef: float = 1e-05, 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): Seed for self.rng. Defaults to 124.

**geometry\_noise** (bool): Whether to use noise in geometry creation, in order to avoid colocated nodes. Defaults to True.

**noise\_coef** (float): Scaling factor of the noise. Defaults to 1.0e-5.

**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, period_id: int = 1, time_margin: int = 0, projected_crs: str =
'EPSG:3857', num_threads: int = -1, seed: int = 124, geometry_noise: bool = True, noise_coef:
float = 1e-05, 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

<code>__init__(public_transport_conn[, period_id, ...])</code>	
<code>add_zones(zones[, from_crs])</code>	Add zones as ODs.
<code>convert_demand_matrix_from_zone_to_node_id</code>	Convert a sparse demand matrix from <code>zone_id</code> 's to the corresponding <code>node_id</code> 's.
<code>create_additional_db_fields()</code>	Create the additional required entries in the tables.
<code>create_graph()</code>	Create the SF transit graph (vertices and edges).
<code>create_line_geometry([method, graph])</code>	Create the LineString for each edge.
<code>create_od_node_mapping()</code>	Build a dataframe mapping the centroid node ids with to transport assignment zone ids.
<code>from_db(public_transport_conn, period_id, ...)</code>	Create a SF graph instance from an existing database save.
<code>save([robust])</code>	Save the current graph to the public transport database.
<code>save_config()</code>	
<code>save_edges([recreate_line_geometry])</code>	Save the contents of <code>self.edges</code> to the public transport database.
<code>save_vertices([robust])</code>	Write the vertices DataFrame to the public transport database.
<code>to_transit_graph()</code>	Create an AequilibraE (TransitGraph) object from an SF graph builder.

## Attributes

<code>config</code>
---------------------

**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`.

**create\_od\_node\_mapping**()

Build a dataframe mapping the centroid node ids with to transport assignment zone ids.

**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\_additional\_db\_fields()**

Create the additional required entries in the tables.

### **save\_vertices(robust=True)**

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): Whether to move stack nodes slightly before saving. Defaults to True.

### **save\_edges(recreate\_line\_geometry=False)**

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.

### **save\_config()**

### **save(robust=True)**

Save the current graph to the public transport database.

#### Arguments

**recreate\_line\_geometry** (bool): Whether to recreate the line strings for the edges as direct lines. Defaults to False.

### **to\_transit\_graph()** → *TransitGraph*

Create an AequilibraE (*TransitGraph*) object from an SF graph builder.

### **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.

### **convert\_demand\_matrix\_from\_zone\_to\_node\_ids(demand\_matrix, o\_zone\_col='origin\_zone\_id', d\_zone\_col='destination\_zone\_id', demand\_col='demand')**

Convert a sparse demand matrix from `zone_id`'s to the corresponding `node_id`'s.

property config

## INSTALLATION

In this section we describe how to install AequilibraE.

### Note

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.

### Note

The recommendations on this page are current as of December 2023.

## A.1 Installation

1. Install [Python 3.8, 3.9, 3.10, 3.11 or 3.12](#). We recommend Python 3.10 or 3.11
2. Install AequilibraE

```
pip install aequilibrae
```

### A.1.1 Dependencies

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

#### 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:

```
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 StackOverflow](#).

```
brew install libspatialite
```

## A.2 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.



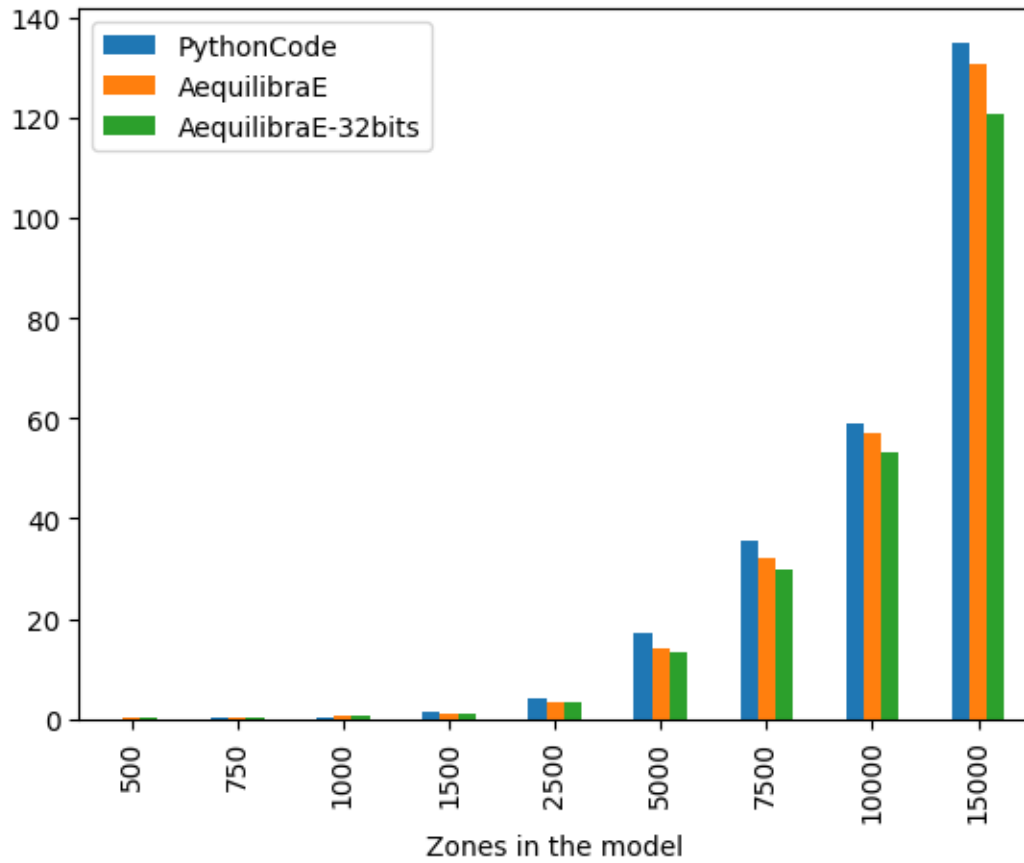
## IPF PERFORMANCE

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.

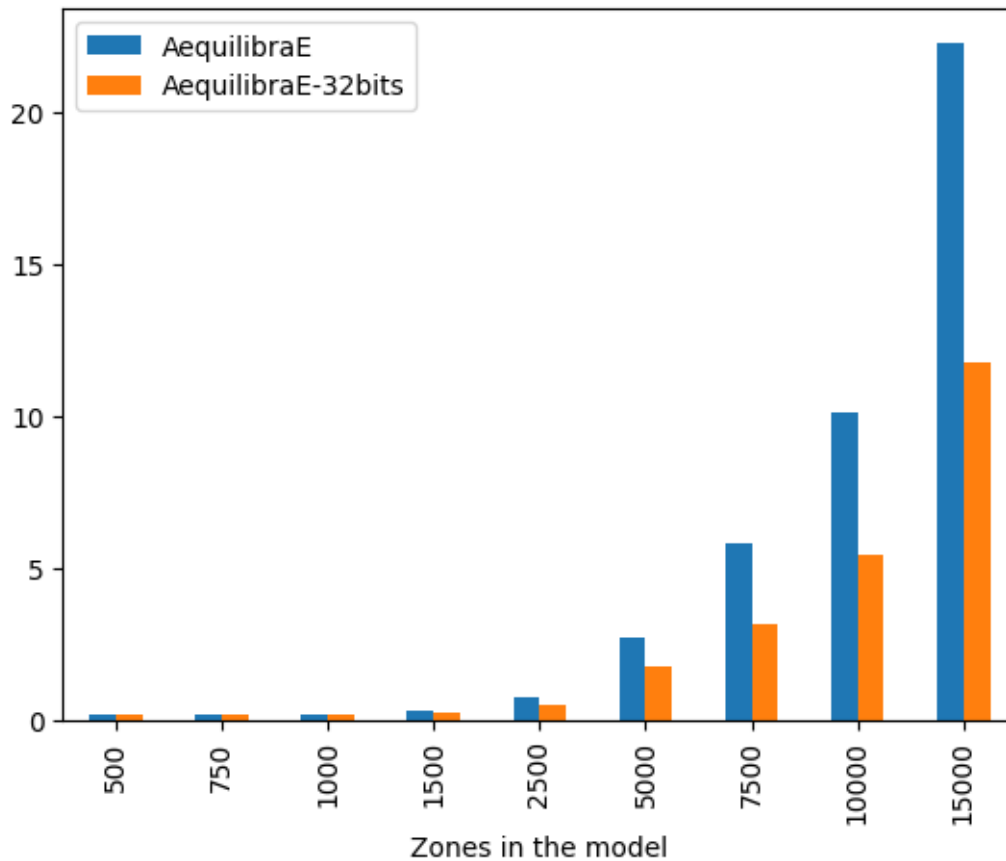
In this section, we compare the runtime of AequilibraE's current implementation of IPF, with a general IPF algorithm written in pure Python, available [here](#).

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.



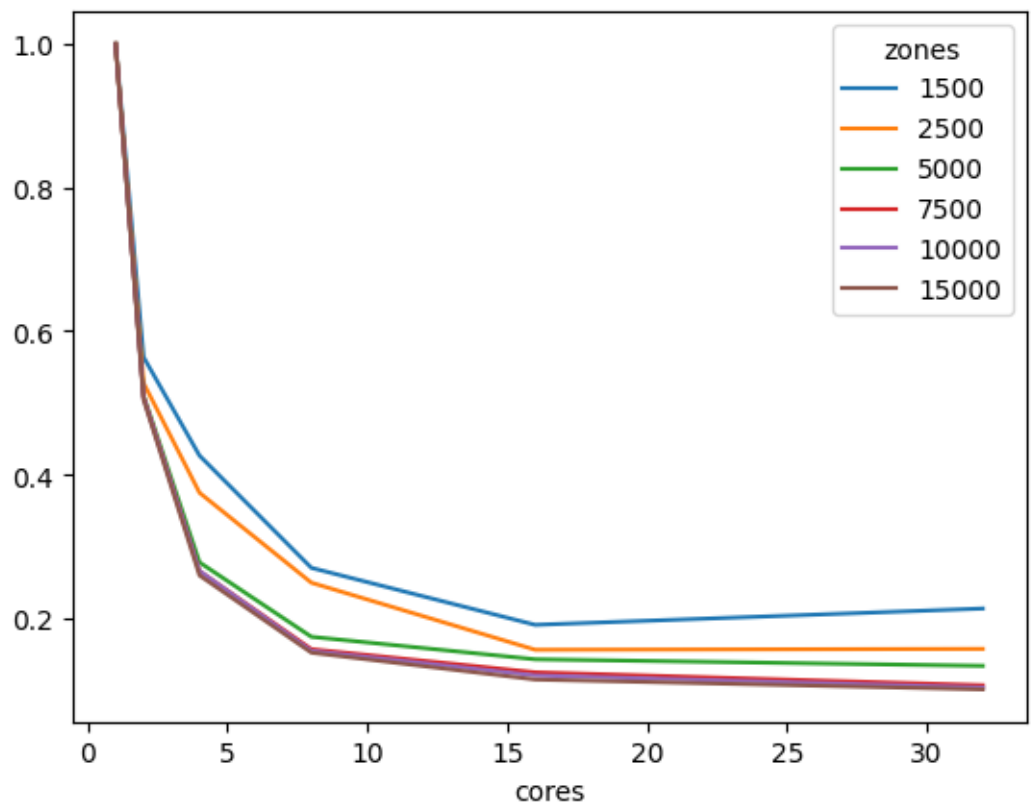
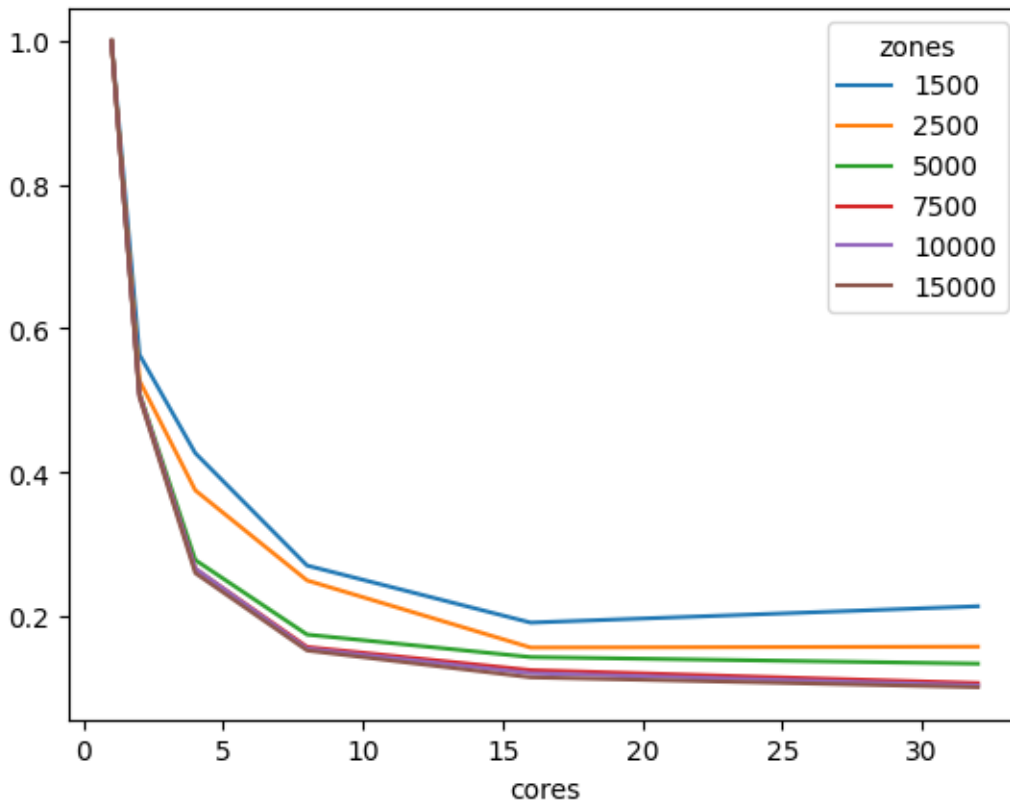
As IPF is an embarrassingly-parallel workload, it is more relevant to look at the performance of the AequilibraE implementations, starting by comparing the implementation performance for inputs in 32 vs 64 bits using 32 threads.





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.



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.



## TRAFFIC ASSIGNMENT

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.

### C.1 Validation

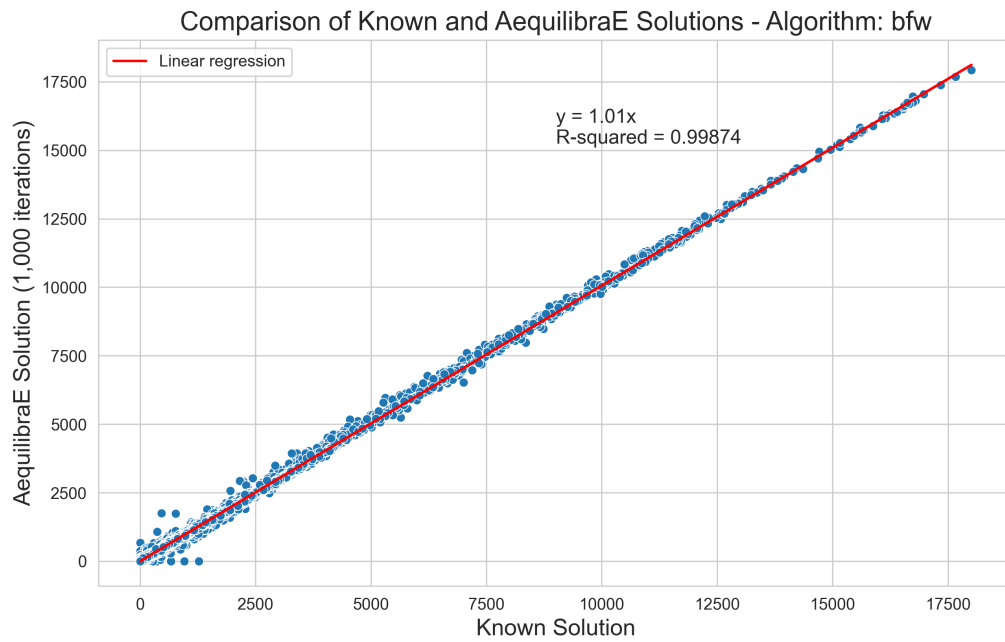
As shown below, the results produced by AequilibraE are within expected, although some differences have been found, particularly for Winnipeg. We suspect that there are issues with the reference results and welcome further investigations.

Chicago

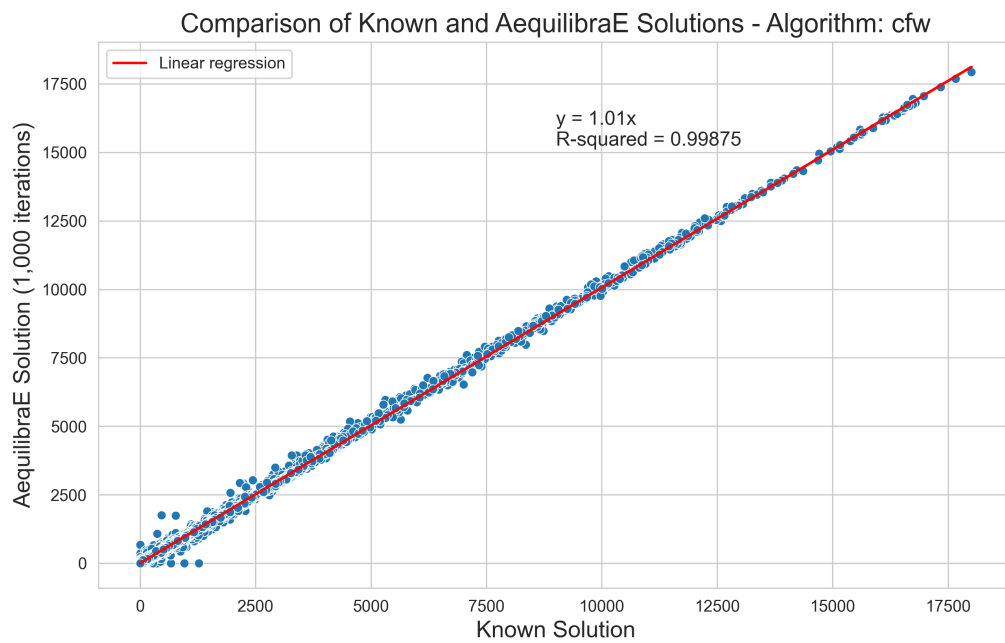
Network stats

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

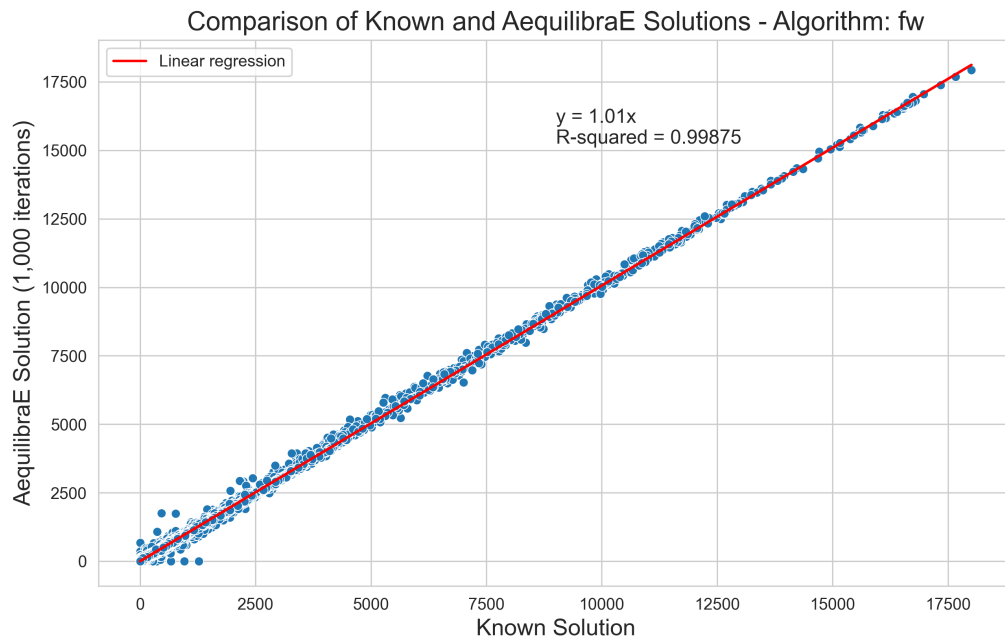
Bi-conjugate Frank-Wolfe



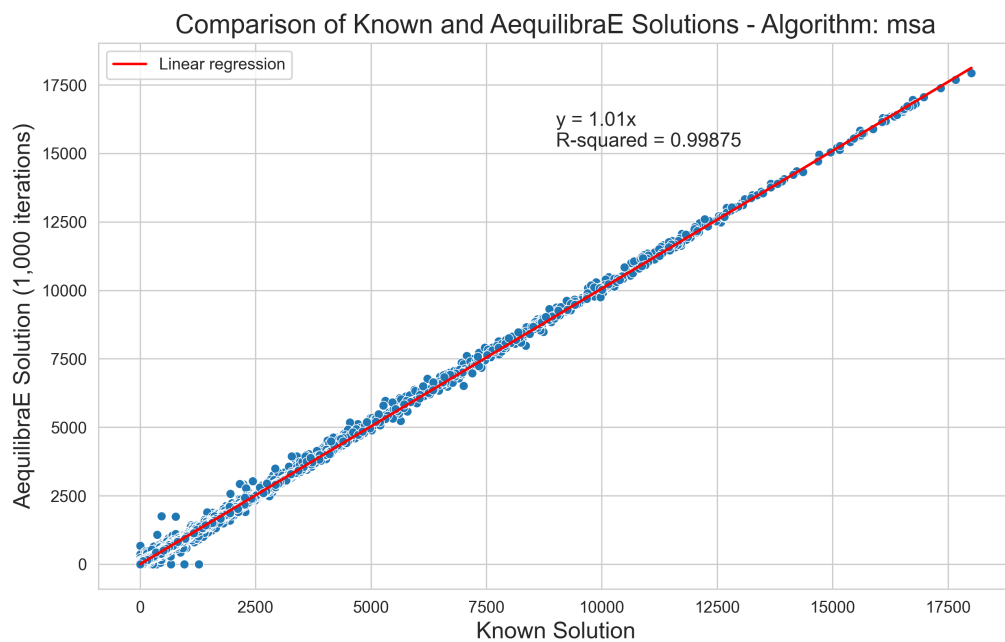
### Conjugate Frank-Wolfe



### Frank-Wolfe



MSA

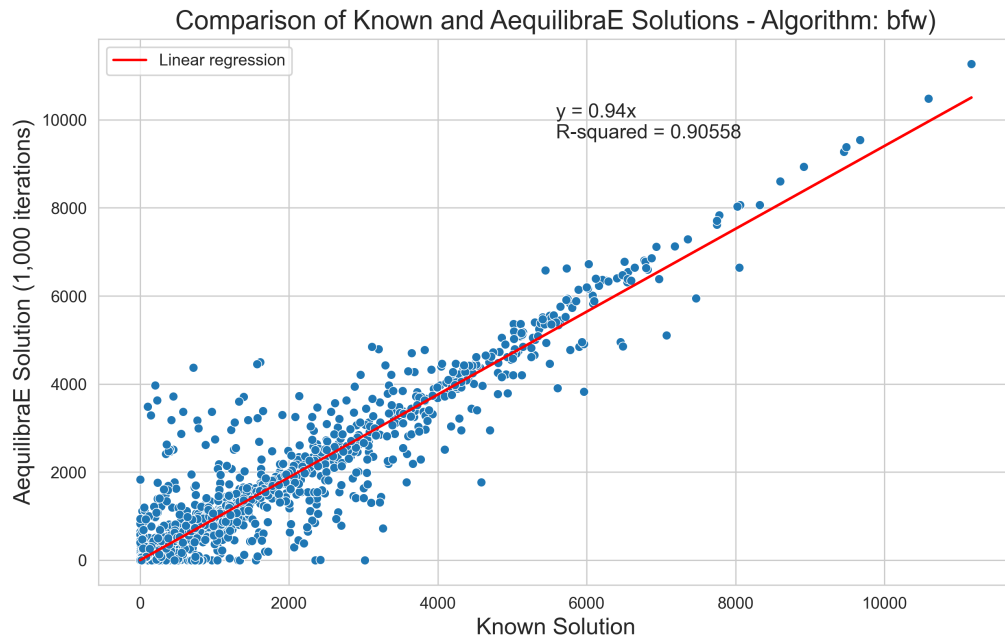


Barcelona

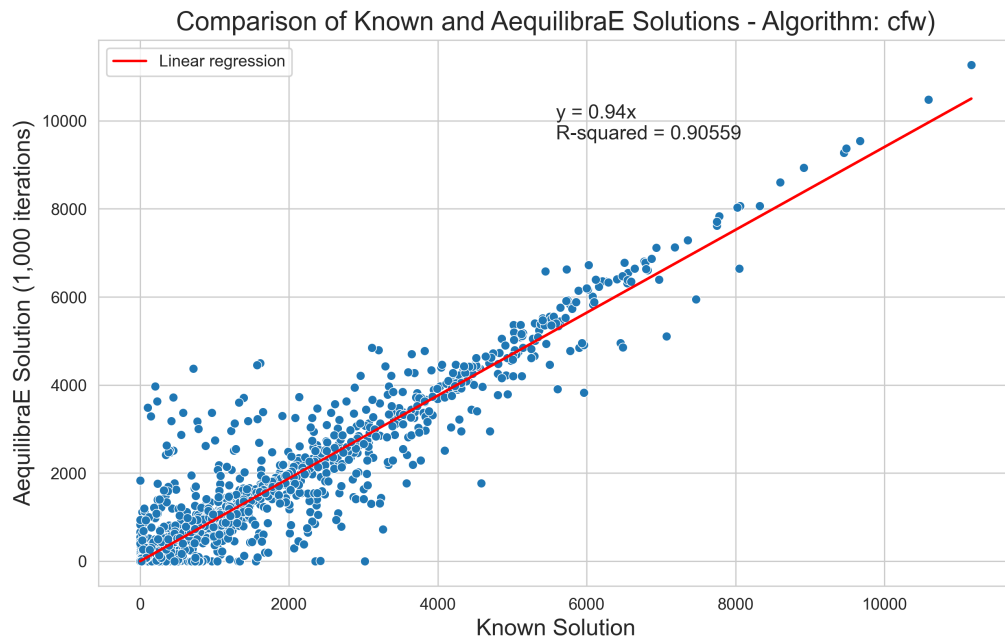
Network stats

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

## Bi-conjugate Frank-Wolfe

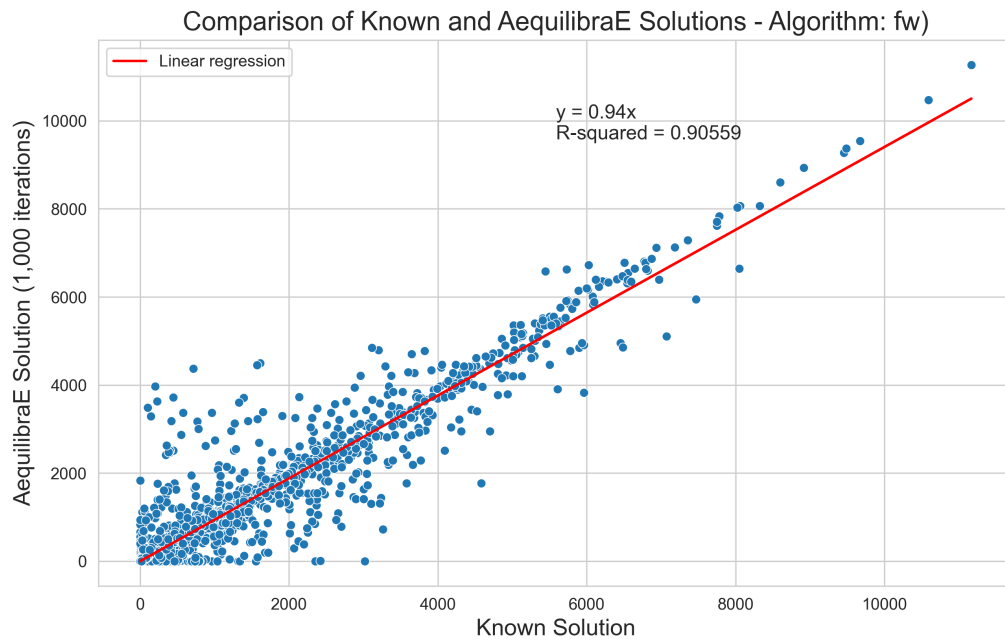


## Conjugate Frank-Wolfe

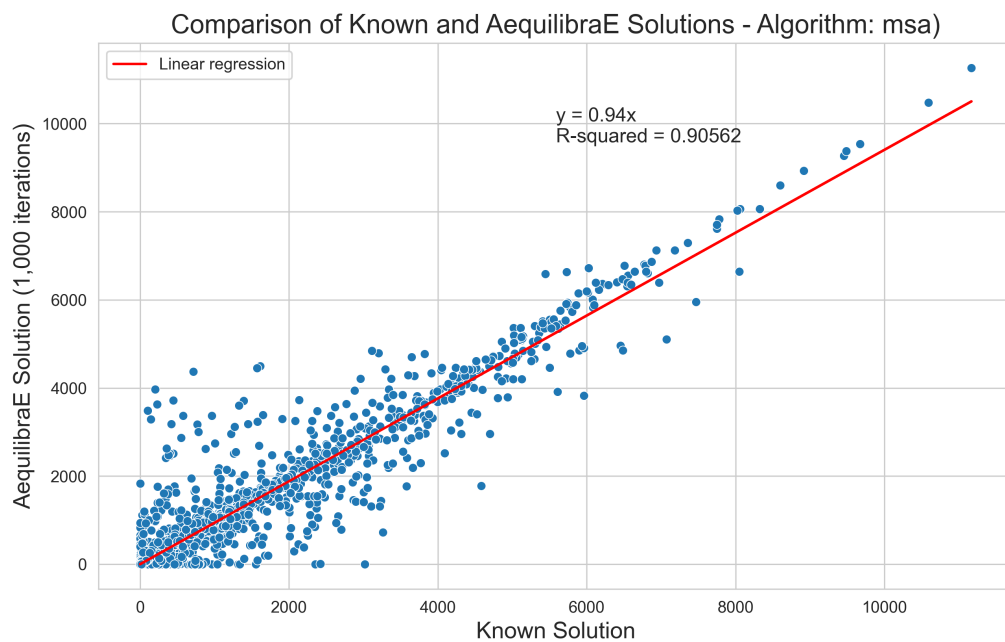


## Frank-Wolfe





MSA

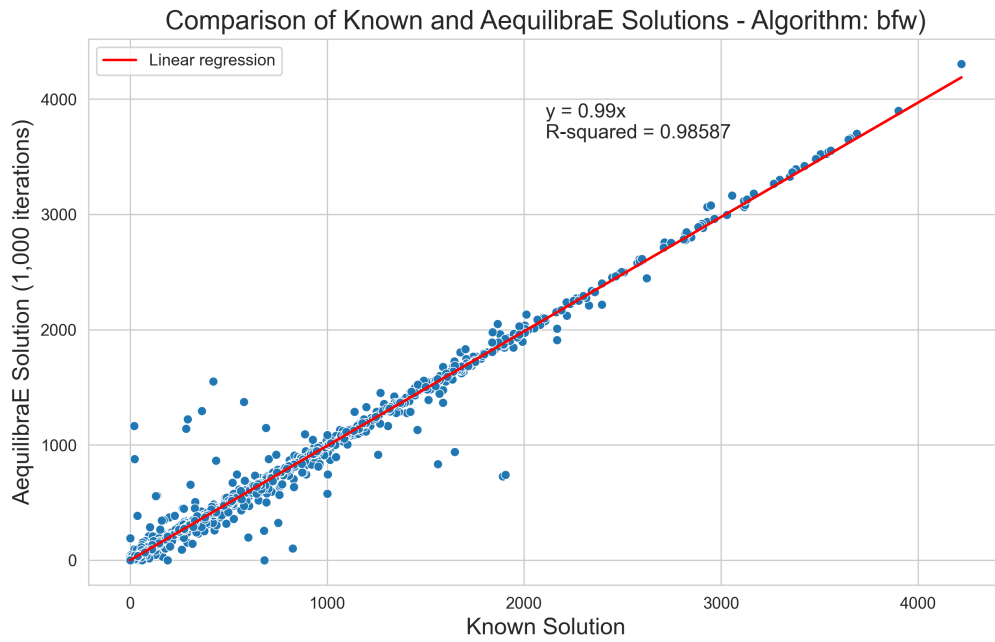


Winnipeg

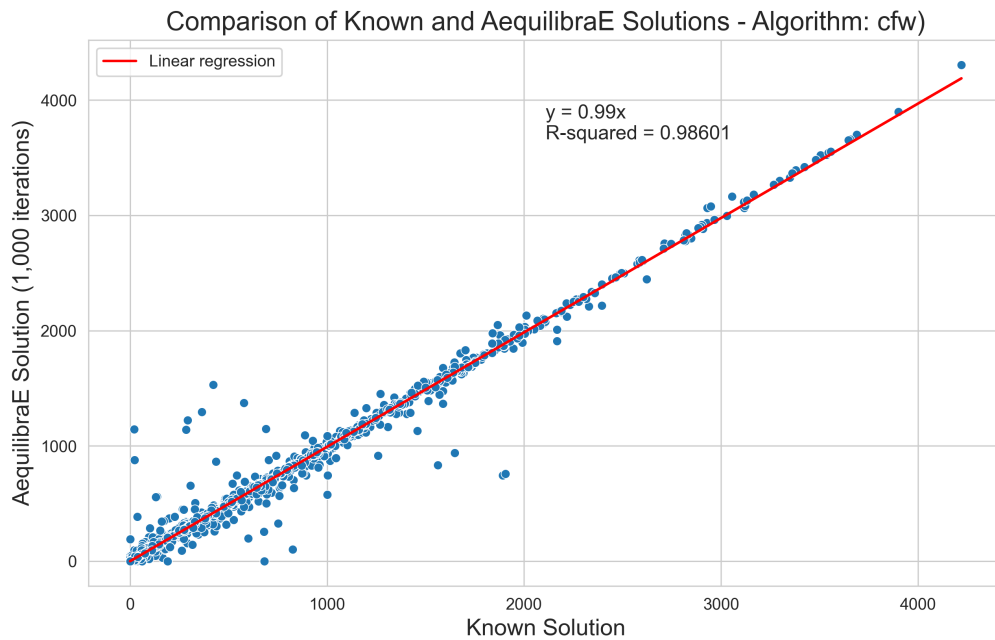
Network stats

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

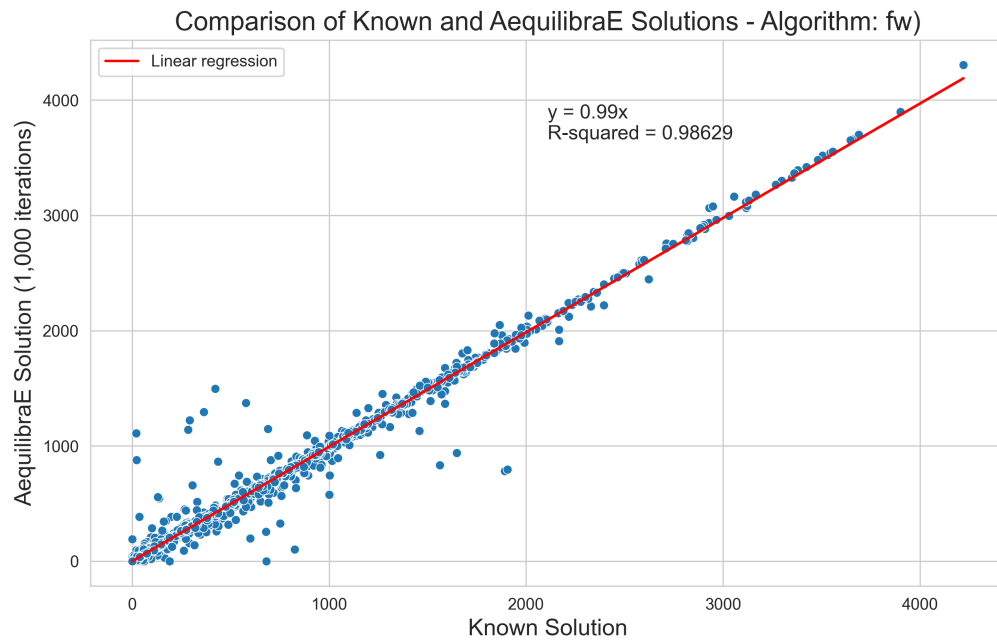
## Bi-conjugate Frank-Wolfe



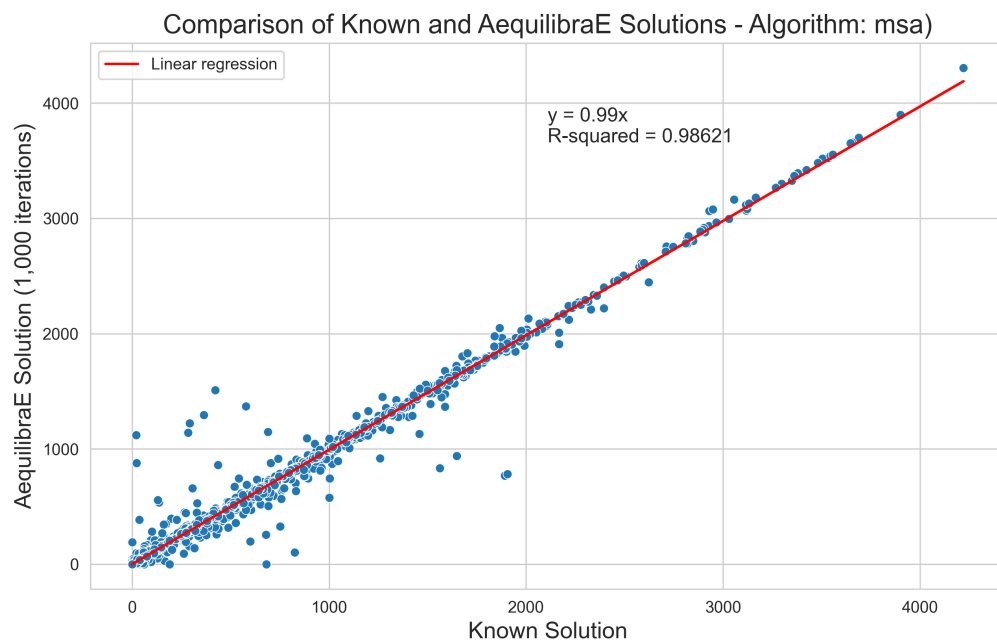
## Conjugate Frank-Wolfe



## Frank-Wolfe



MSA

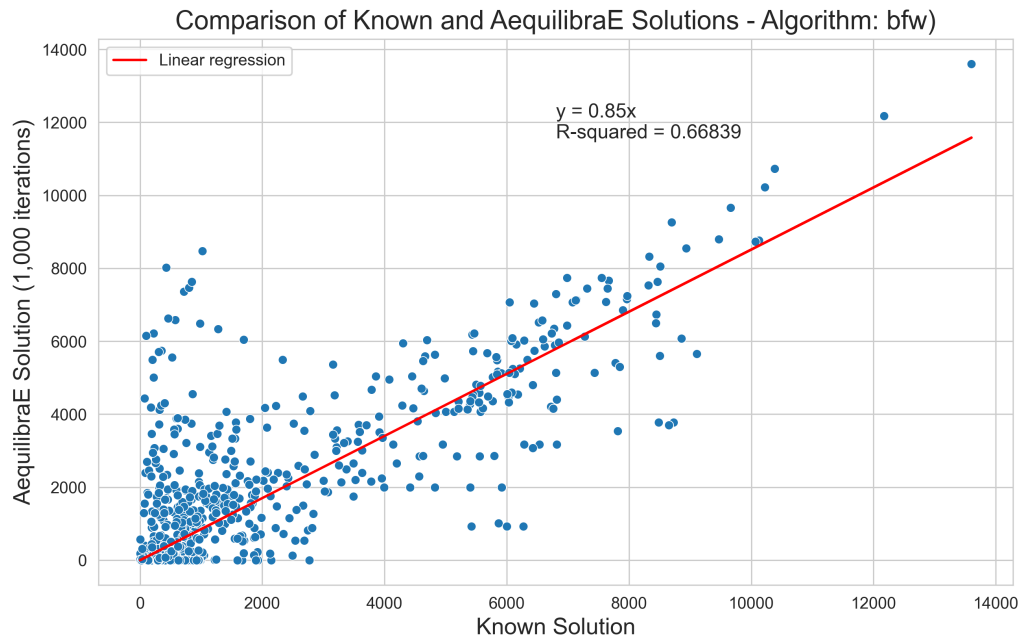


Anaheim

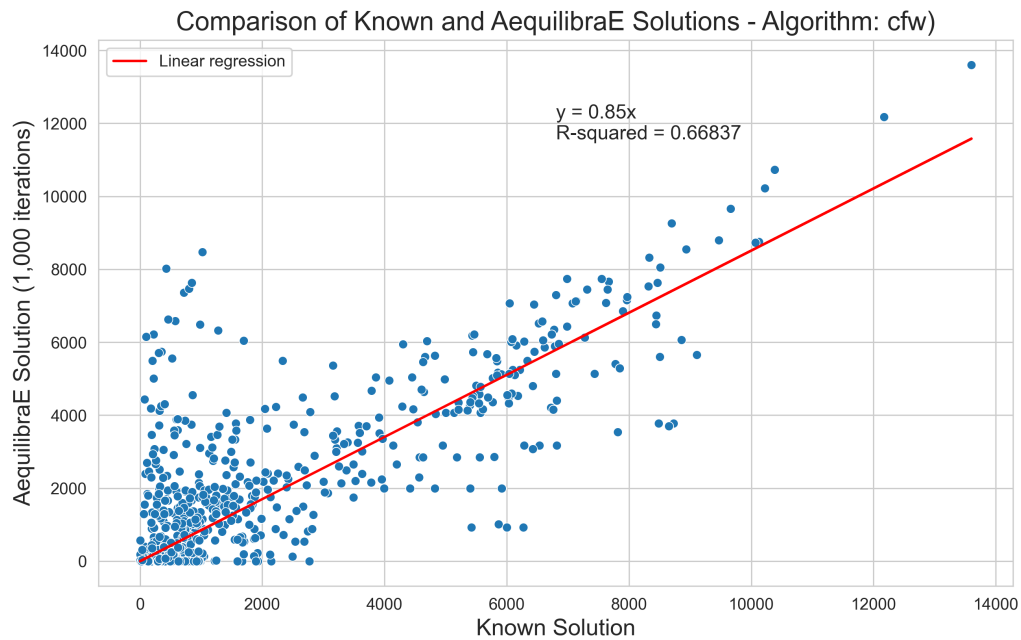
Network stats

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

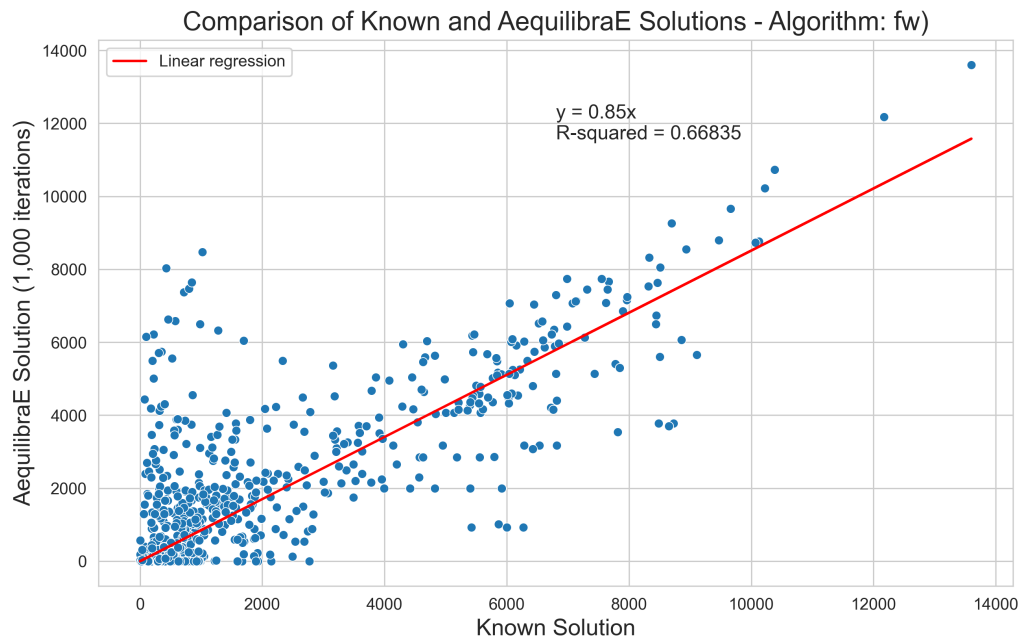
## Bi-conjugate Frank-Wolfe



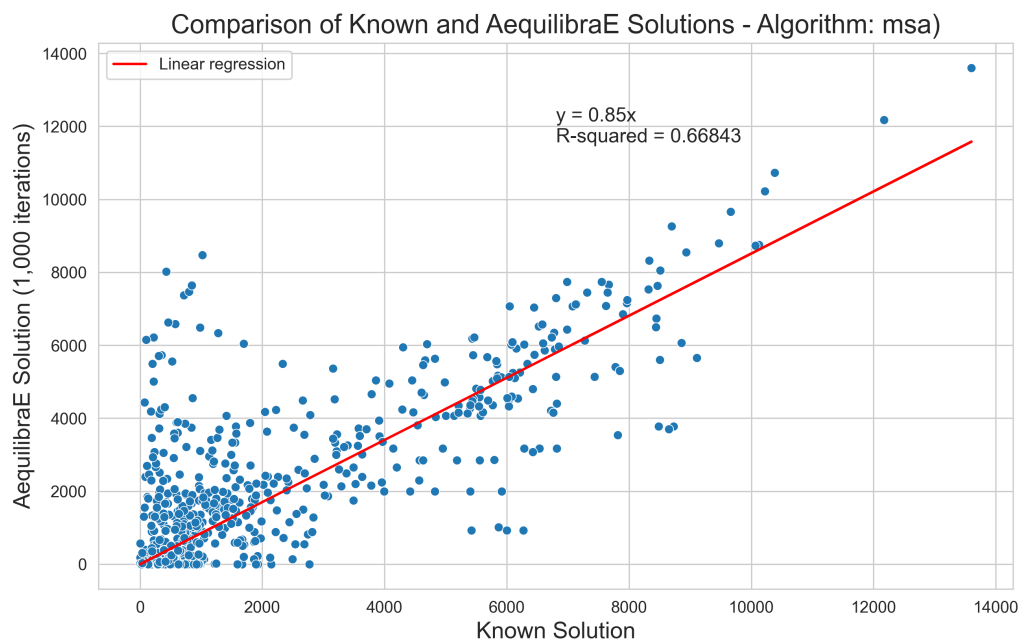
## Conjugate Frank-Wolfe



## Frank-Wolfe



MSA

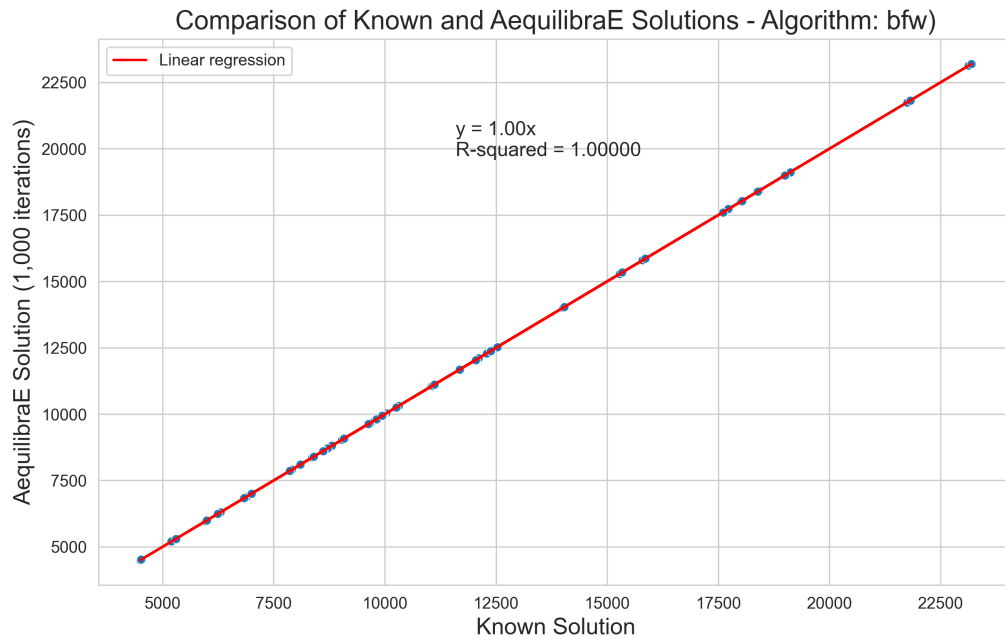


Sioux Falls

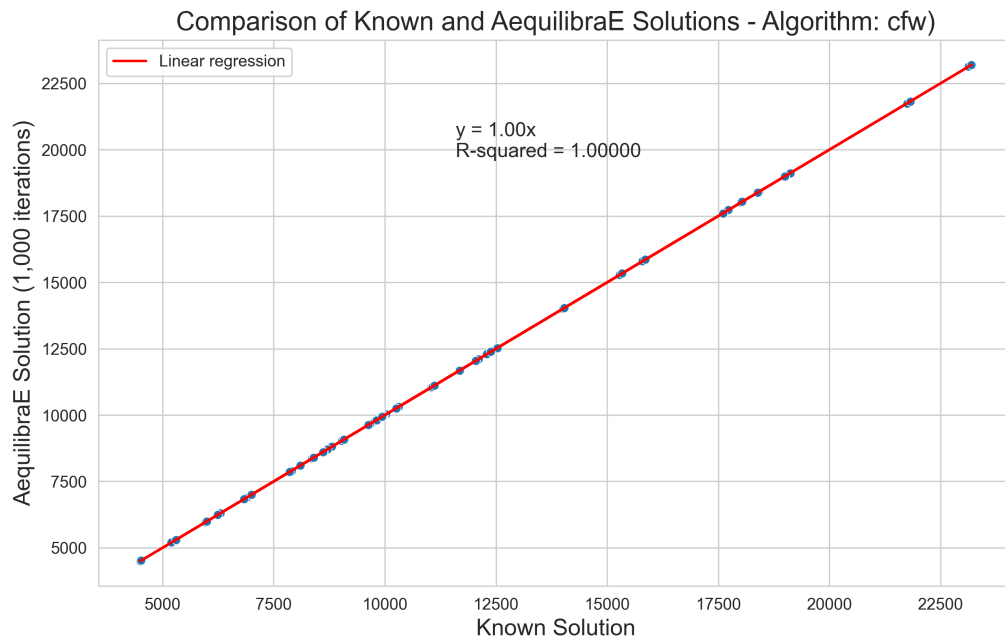
Network stats

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

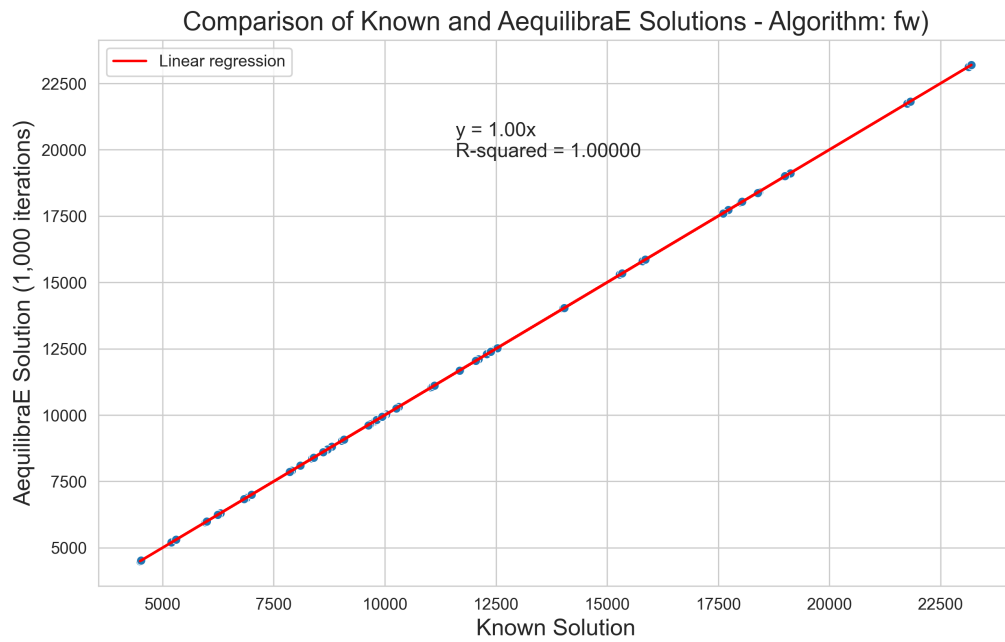
## Bi-conjugate Frank-Wolfe



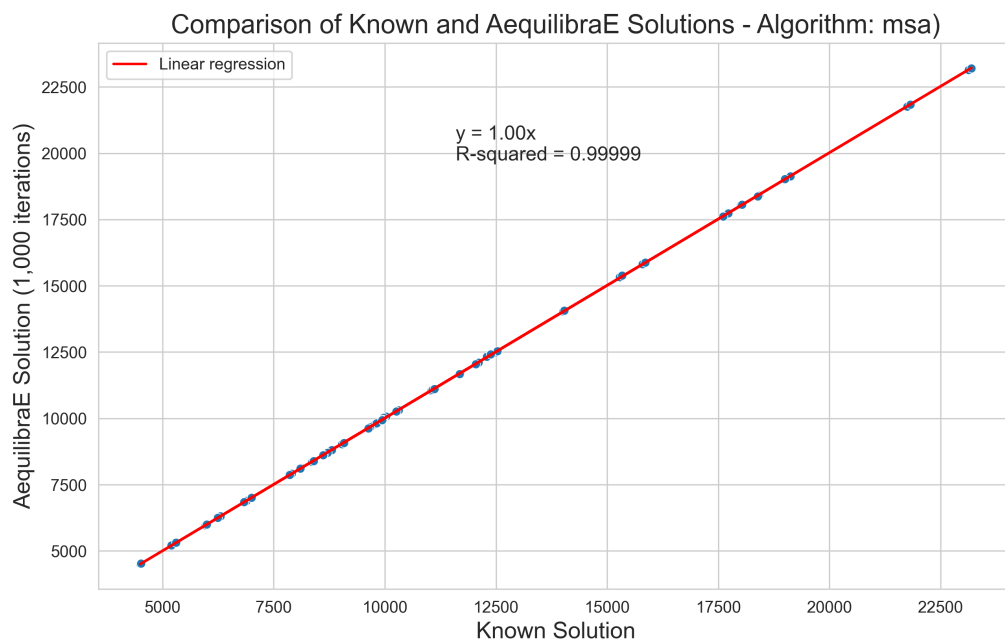
## Conjugate Frank-Wolfe



## Frank-Wolfe



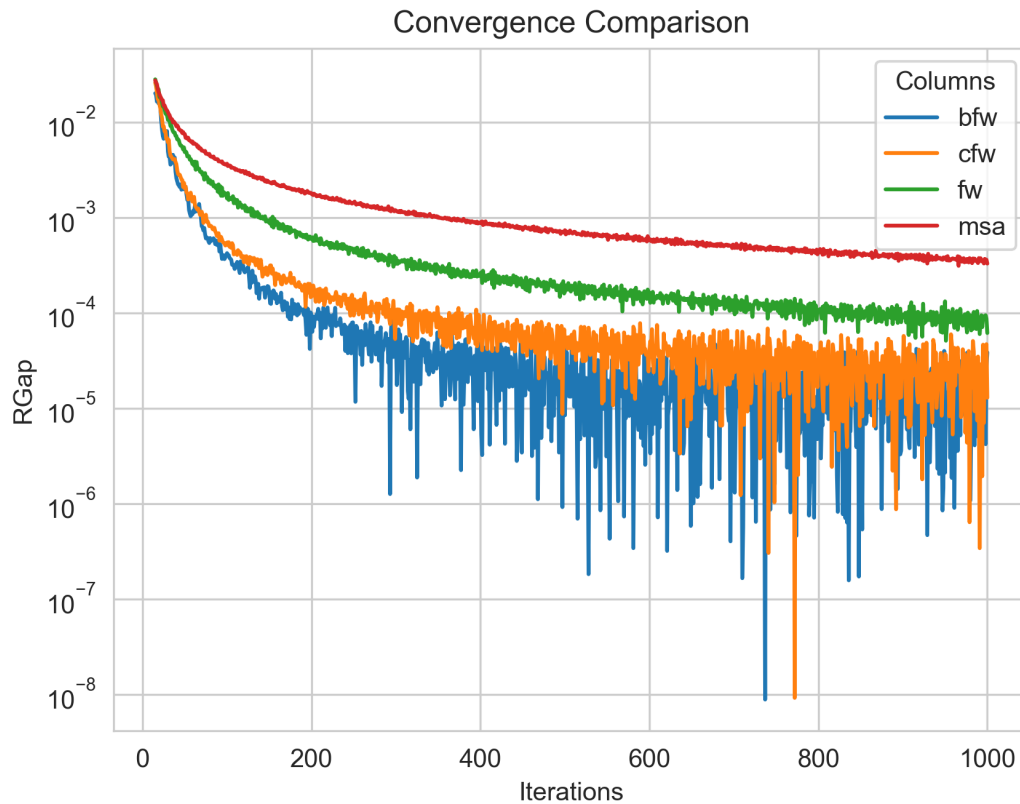
MSA



## C.2 Convergence Study

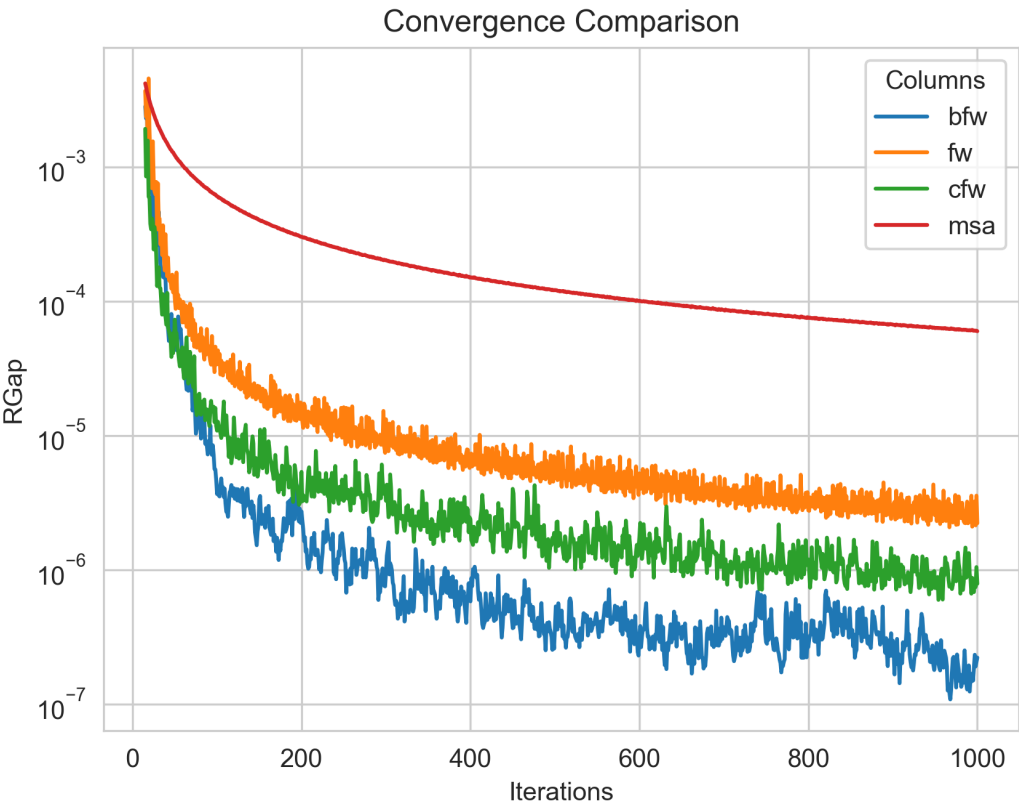
Besides validating the final results from the algorithms, we have also compared how well they converge for the largest instance we have tested (Chicago Regional), as that instance has a comparable size to real-world models.

Chicago

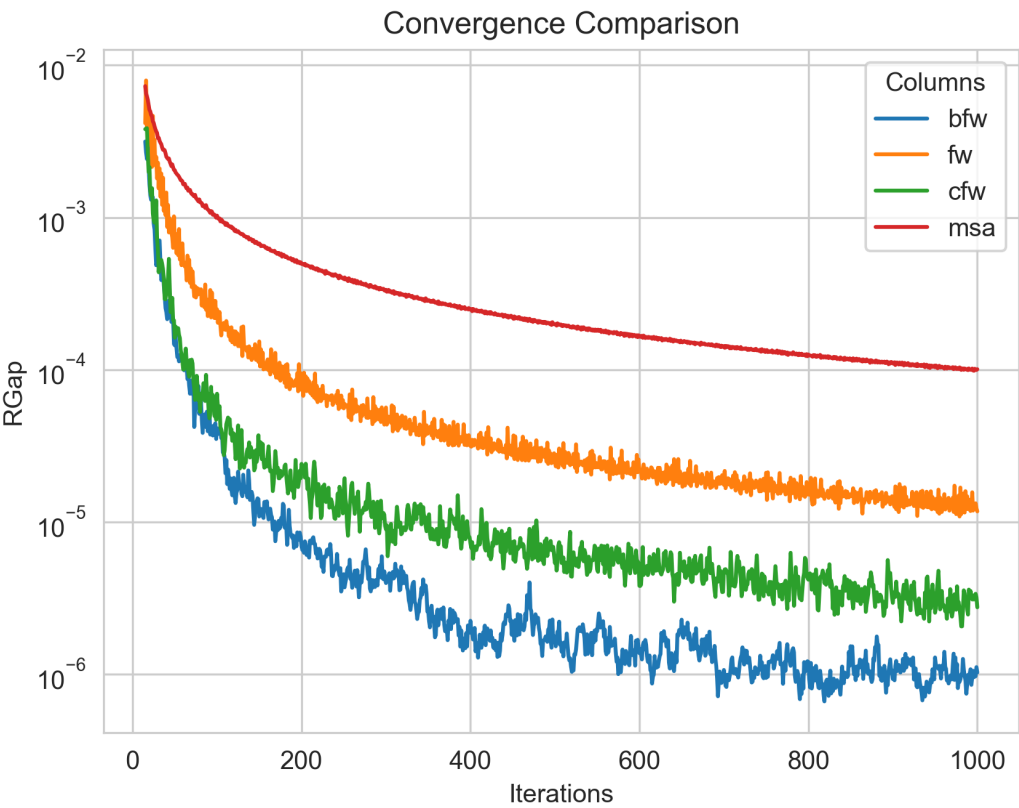


Barcelona

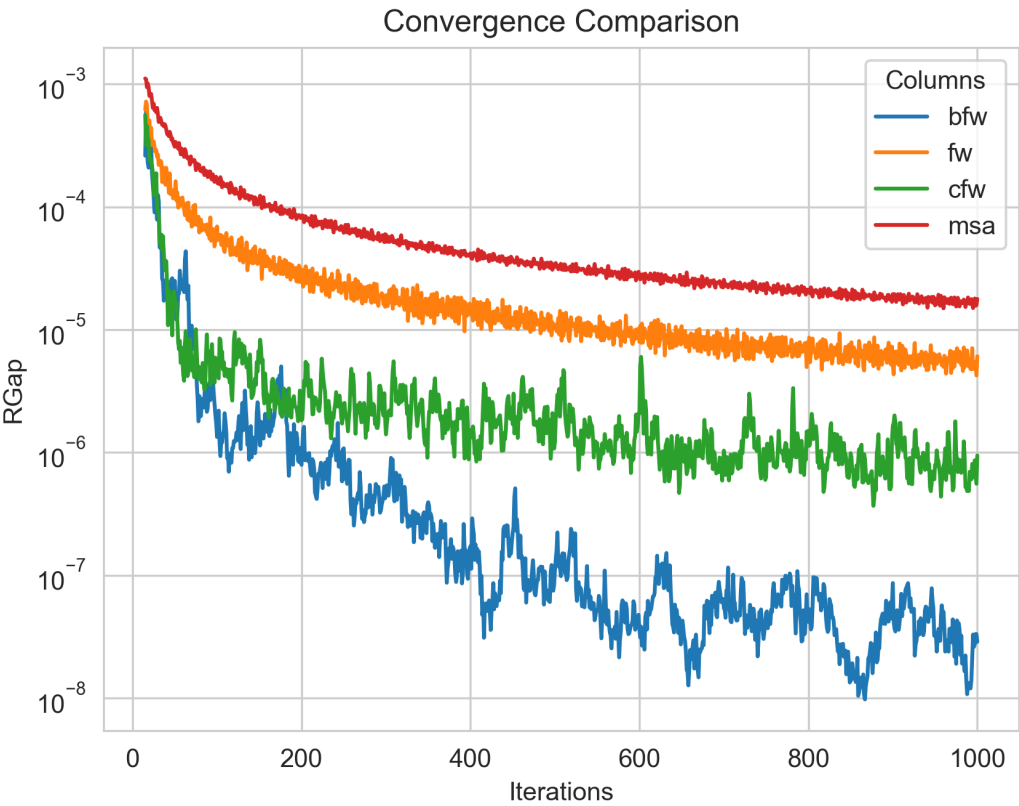




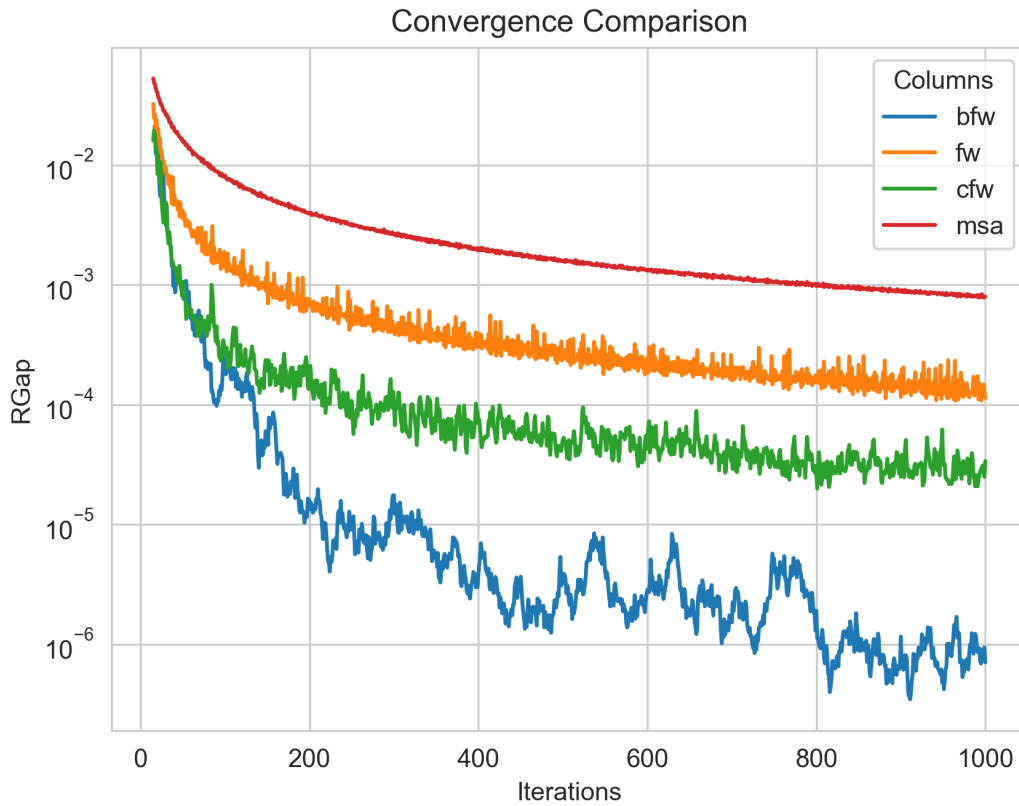
Winnipeg



Anaheim



Sioux-Falls



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 Bi-conjugate 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.

### C.3 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 Bi-conjugate 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

## C.4 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.

```
# 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))
```

(continues on next page)

(continued from previous page)

```

    for i in range(zones):
        for j in range(zones):
            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):
    assignclass = TrafficClass("car", g, mat)
    if "toll" in g.network.columns:
        assignclass.set_fixed_cost("toll")

```

(continues on next page)

(continued from previous page)

```

    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.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):
```

(continues on next page)

(continued from previous page)

```

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

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

```



## IMPORTING FROM OPEN STREET MAPS

Please review the information *Open Street Maps*

### 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**

### D.1 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.

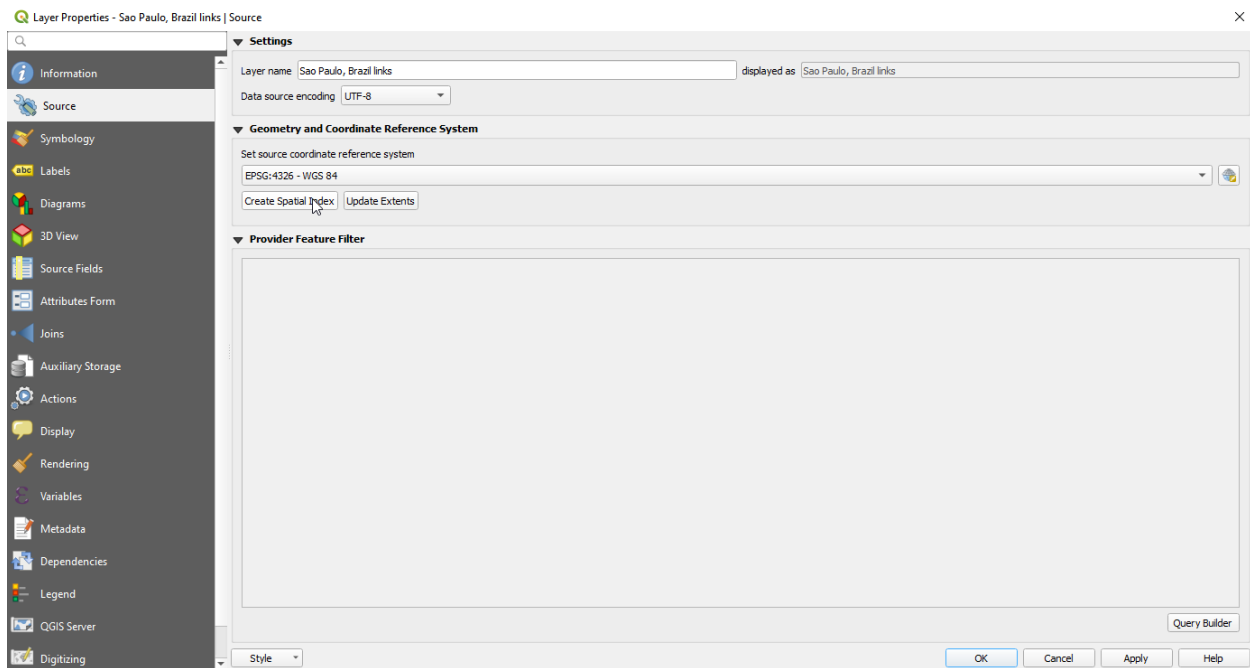
For this reason, AequilibraE's default option when importing a network from OSM is to **NOT create spatial indices**, which renders the network consistency triggers useless.

If you are using a vanilla Python installation (your case if you are not sure), you can import the network without creating indices, as shown below.

```
from aequilibrae.project import Project

p = Project()
p.new('path/to/project/new/folder')
p.network.create_from_osm(place_name='my favorite place')
p.conn.close()
```

And then manually add the spatial index on QGIS by adding both links and nodes layers to the canvas, and selecting properties and clicking on *create spatial index* for each layer at a time. This action automatically saves the spatial indices to the sqlite database.



If you are an expert user and made sure your Python installation was compiled against a complete SQLite set of extensions, then go ahead and import the network with the option for creating such indices.

```
from aequilibrae.project import Project

p = Project()
p.new('path/to/project/new/folder/')
p.network.create_from_osm(place_name='my favorite place', spatial_index=True)
p.conn.close()
```

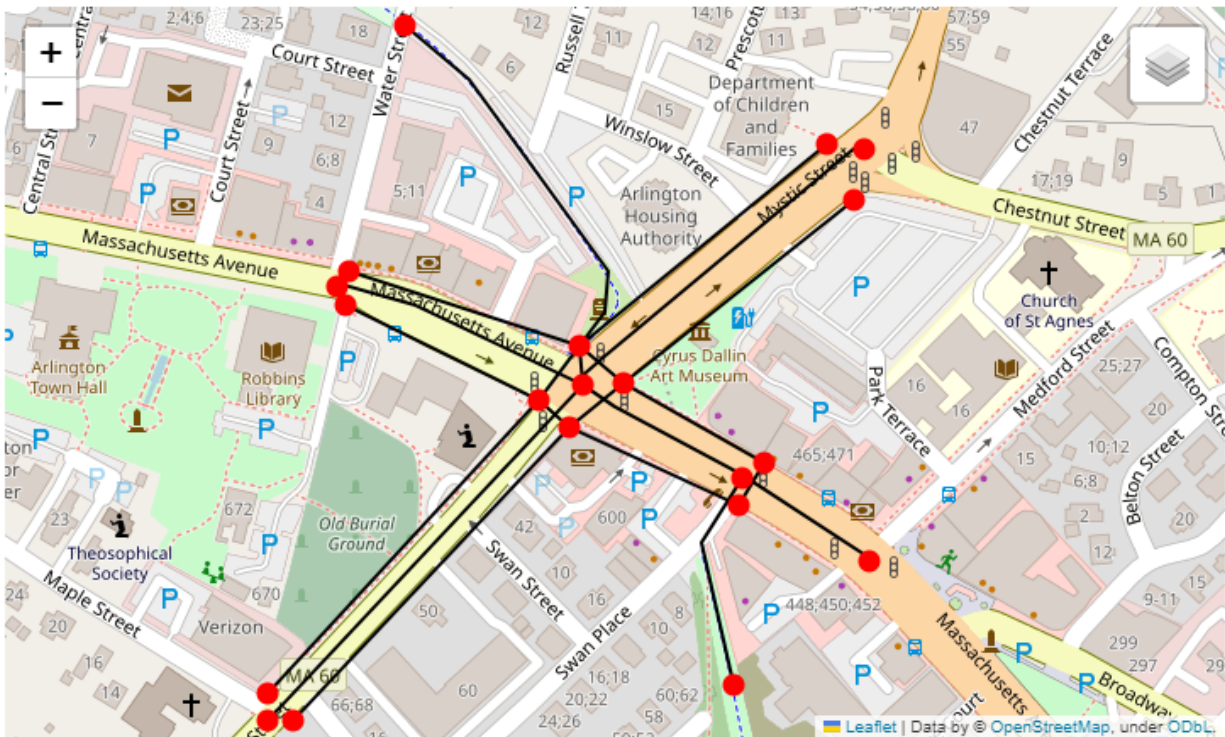
If you want to learn a little more about this topic, you can access this [blog post](#) or check out the SQLite page on [R-Tree](#).

If you want to take a stab at solving your SQLite/Spatialite problem permanently, take a look at this [other blog post](#).

Please also note that the 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 RTree and Spatialite.

## 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'.



As of July 2022, 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](#).

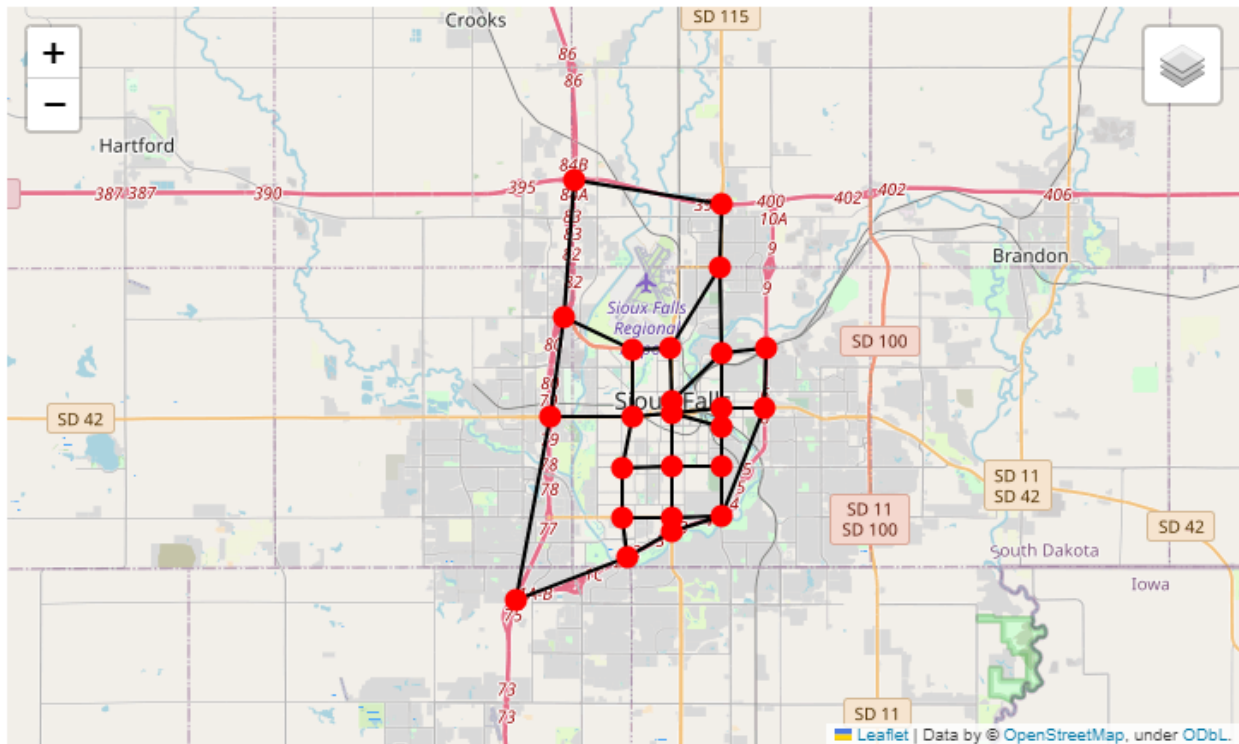
When adding a new field to be read in the `parameters.yml` file, it is important to keep the “required” key set to `False`, since you will always be adding a non-required field. Required fields for a specific table are only those defined in the GMNS specification.

### 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’.**

## EXPORTING AEQUILIBRAE MODEL TO GMNS FORMAT

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



As of July 2022, 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 `create_from_gmns()` function, because mode groups are not used in the AequilibraE modes table.

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

### Note

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

You can find the GMNS specification [here](#).

## PYTHON MODULE INDEX

### a

`aequilibrae`, [177](#)





## Symbols

- `__init__()` (*aequilibrae.Parameters* method), 206
- `__init__()` (*aequilibrae.distribution.GravityApplication* method), 210
- `__init__()` (*aequilibrae.distribution.GravityCalibration* method), 212
- `__init__()` (*aequilibrae.distribution.Ipf* method), 208
- `__init__()` (*aequilibrae.distribution.SyntheticGravityModel* method), 213
- `__init__()` (*aequilibrae.matrix.AequilibraeData* method), 213
- `__init__()` (*aequilibrae.matrix.AequilibraeMatrix* method), 215
- `__init__()` (*aequilibrae.paths.AssignmentResults* method), 227
- `__init__()` (*aequilibrae.paths.Graph* method), 223
- `__init__()` (*aequilibrae.paths.HyperpathGenerating* method), 243
- `__init__()` (*aequilibrae.paths.OptimalStrategies* method), 244
- `__init__()` (*aequilibrae.paths.PathResults* method), 231
- `__init__()` (*aequilibrae.paths.SkimResults* method), 230
- `__init__()` (*aequilibrae.paths.TrafficAssignment* method), 236
- `__init__()` (*aequilibrae.paths.TrafficClass* method), 233
- `__init__()` (*aequilibrae.paths.TransitAssignment* method), 240
- `__init__()` (*aequilibrae.paths.TransitAssignmentResults* method), 228
- `__init__()` (*aequilibrae.paths.TransitClass* method), 234
- `__init__()` (*aequilibrae.paths.TransitGraph* method), 225
- `__init__()` (*aequilibrae.paths.VDF* method), 232
- `__init__()` (*aequilibrae.project.About* method), 179
- `__init__()` (*aequilibrae.project.FieldEditor* method), 181
- `__init__()` (*aequilibrae.project.Log* method), 182
- `__init__()` (*aequilibrae.project.Matrices* method), 182
- `__init__()` (*aequilibrae.project.Network* method), 184
- `__init__()` (*aequilibrae.project.Project* method), 177
- `__init__()` (*aequilibrae.project.Zone* method), 190
- `__init__()` (*aequilibrae.project.Zoning* method), 188
- `__init__()` (*aequilibrae.project.network.Link* method), 202
- `__init__()` (*aequilibrae.project.network.LinkType* method), 201
- `__init__()` (*aequilibrae.project.network.LinkTypes* method), 194
- `__init__()` (*aequilibrae.project.network.Links* method), 195
- `__init__()` (*aequilibrae.project.network.Mode* method), 200
- `__init__()` (*aequilibrae.project.network.Modes* method), 192
- `__init__()` (*aequilibrae.project.network.Node* method), 203
- `__init__()` (*aequilibrae.project.network.Nodes* method), 197
- `__init__()` (*aequilibrae.project.network.Period* method), 205
- `__init__()` (*aequilibrae.project.network.Periods* method), 198
- `__init__()` (*aequilibrae.transit.Transit* method), 245
- `__init__()` (*aequilibrae.transit.TransitGraphBuilder* method), 247

## A

- About (class in *aequilibrae.project*), 179
- activate() (*aequilibrae.project.Project* method), 179
- add() (*aequilibrae.project.FieldEditor* method), 181
- add() (*aequilibrae.project.network.Modes* method), 192
- add\_centroid() (*aequilibrae.project.Zone* method), 190
- add\_class() (*aequilibrae.paths.TrafficAssignment* method), 238
- add\_class() (*aequilibrae.paths.TransitAssignment* method), 242
- add\_info\_field() (*aequilibrae.project.About* method), 180

- `add_mode()` (*aequilibrae.project.network.Link* method), 202
- `add_preload()` (*aequilibrae.paths.TrafficAssignment* method), 239
- `add_zones()` (*aequilibrae.transit.TransitGraphBuilder* method), 248
- `aequilibrae` module, 177
- `AequilibraeData` (class in *aequilibrae.matrix*), 213
- `AequilibraeMatrix` (class in *aequilibrae.matrix*), 215
- `algorithms_available()` (*aequilibrae.paths.TrafficAssignment* method), 240
- `algorithms_available()` (*aequilibrae.paths.TransitAssignment* method), 242
- `all_algorithms` (*aequilibrae.paths.TrafficAssignment* attribute), 237
- `all_algorithms` (*aequilibrae.paths.TransitAssignment* attribute), 241
- `all_fields()` (*aequilibrae.project.FieldEditor* method), 181
- `all_modes()` (*aequilibrae.project.network.Modes* method), 193
- `all_types()` (*aequilibrae.project.network.LinkTypes* method), 194
- `all_zones()` (*aequilibrae.project.Zoning* method), 189
- `apply()` (*aequilibrae.distribution.GravityApplication* method), 211
- `assign()` (*aequilibrae.paths.HyperpathGenerating* method), 243
- `AssignmentResults` (class in *aequilibrae.paths*), 227
- `available_skims()` (*aequilibrae.paths.Graph* method), 224
- `available_skims()` (*aequilibrae.paths.TransitGraph* method), 225
- B**
- `bpr_parameters` (*aequilibrae.paths.TrafficAssignment* attribute), 237
- `build_graphs()` (*aequilibrae.project.Network* method), 186
- `build_pt_preload()` (*aequilibrae.transit.Transit* method), 246
- C**
- `calibrate()` (*aequilibrae.distribution.GravityCalibration* method), 212
- `check_exists()` (*aequilibrae.project.Matrices* method), 183
- `check_file_indices()` (*aequilibrae.project.Project* method), 179
- `cleaning()` (in module *aequilibrae*), 177
- `clear()` (*aequilibrae.project.Log* method), 182
- `clear_database()` (*aequilibrae.project.Matrices* method), 183
- `close()` (*aequilibrae.matrix.AequilibraeMatrix* method), 218
- `close()` (*aequilibrae.project.Project* method), 178
- `columns()` (*aequilibrae.matrix.AequilibraeMatrix* method), 221
- `computational_view()` (*aequilibrae.matrix.AequilibraeMatrix* method), 219
- `compute_path()` (*aequilibrae.paths.PathResults* method), 231
- `config` (*aequilibrae.transit.TransitGraphBuilder* property), 249
- `connect()` (*aequilibrae.project.Project* method), 179
- `connect_db()` (*aequilibrae.project.network.Link* method), 203
- `connect_db()` (*aequilibrae.project.network.LinkType* method), 201
- `connect_db()` (*aequilibrae.project.network.Node* method), 204
- `connect_db()` (*aequilibrae.project.network.Period* method), 205
- `connect_db()` (*aequilibrae.project.Zone* method), 191
- `connect_mode()` (*aequilibrae.project.network.Node* method), 204
- `connect_mode()` (*aequilibrae.project.Zone* method), 190
- `contents()` (*aequilibrae.project.Log* method), 182
- `convert_demand_matrix_from_zone_to_node_ids()` (*aequilibrae.transit.TransitGraphBuilder* method), 249
- `convex_hull()` (*aequilibrae.project.Network* method), 187
- `copy()` (*aequilibrae.matrix.AequilibraeMatrix* method), 220
- `copy_link()` (*aequilibrae.project.network.Links* method), 196
- `count_centroids()` (*aequilibrae.project.Network* method), 187
- `count_links()` (*aequilibrae.project.Network* method), 187
- `count_nodes()` (*aequilibrae.project.Network* method), 187
- `coverage()` (*aequilibrae.project.Zoning* method), 188
- `create()` (*aequilibrae.project.About* method), 180
- `create_additional_db_fields()` (*aequilibrae.transit.TransitGraphBuilder* method), 249
- `create_compressed_link_network_mapping()` (*aequilibrae.paths.Graph* method), 224
- `create_compressed_link_network_mapping()` (*aequilibrae.paths.TransitGraph* method), 226
- `create_empty()` (*aequilibrae.matrix.AequilibraeData*

- method), 213
- create\_empty() (aequilibrae.matrix.AequilibraeMatrix method), 216
- create\_from\_gmns() (aequilibrae.project.Network method), 186
- create\_from\_omx() (aequilibrae.matrix.AequilibraeMatrix method), 217
- create\_from\_osm() (aequilibrae.project.Network method), 185
- create\_from\_trip\_list() (aequilibrae.matrix.AequilibraeMatrix method), 218
- create\_graph() (aequilibrae.transit.Transit method), 246
- create\_graph() (aequilibrae.transit.TransitGraphBuilder method), 248
- create\_line\_geometry() (aequilibrae.transit.TransitGraphBuilder method), 248
- create\_od\_node\_mapping() (aequilibrae.transit.TransitGraphBuilder method), 248
- create\_transit\_database() (aequilibrae.transit.Transit method), 246
- create\_zoning\_layer() (aequilibrae.project.Zoning method), 188
- ## D
- data (aequilibrae.project.network.Links property), 196
- data (aequilibrae.project.network.Nodes property), 198
- data (aequilibrae.project.network.Periods property), 200
- data (aequilibrae.project.Zoning property), 189
- data\_fields() (aequilibrae.project.network.Link method), 203
- data\_fields() (aequilibrae.project.network.Node method), 204
- data\_fields() (aequilibrae.project.network.Period method), 205
- deactivate() (aequilibrae.project.Project method), 179
- default\_capacities (aequilibrae.transit.Transit attribute), 245
- default\_pces (aequilibrae.transit.Transit attribute), 245
- default\_period (aequilibrae.project.network.Periods property), 200
- default\_types() (aequilibrae.paths.Graph method), 224
- default\_types() (aequilibrae.paths.TransitGraph method), 226
- delete() (aequilibrae.project.network.Link method), 202
- delete() (aequilibrae.project.network.Links method), 196
- delete() (aequilibrae.project.network.LinkType method), 201
- delete() (aequilibrae.project.network.LinkTypes method), 194
- delete() (aequilibrae.project.network.Modes method), 192
- delete() (aequilibrae.project.Zone method), 190
- delete\_record() (aequilibrae.project.Matrices method), 183
- disconnect\_mode() (aequilibrae.project.Zone method), 191
- drop\_mode() (aequilibrae.project.network.Link method), 203
- ## E
- empty() (aequilibrae.matrix.AequilibraeData class method), 213
- exclude\_links() (aequilibrae.paths.Graph method), 224
- exclude\_links() (aequilibrae.paths.TransitGraph method), 226
- execute() (aequilibrae.paths.OptimalStrategies method), 244
- execute() (aequilibrae.paths.TrafficAssignment method), 240
- execute() (aequilibrae.paths.TransitAssignment method), 242
- export() (aequilibrae.matrix.AequilibraeData method), 214
- export() (aequilibrae.matrix.AequilibraeMatrix method), 218
- export\_to\_gmns() (aequilibrae.project.Network method), 186
- extent() (aequilibrae.project.Network method), 187
- extent() (aequilibrae.project.network.Links method), 196
- extent() (aequilibrae.project.network.Nodes method), 198
- extent() (aequilibrae.project.network.Periods method), 199
- extent() (aequilibrae.project.Zoning method), 189
- ## F
- FieldEditor (class in aequilibrae.project), 181
- fields (aequilibrae.project.network.Links property), 196
- fields (aequilibrae.project.network.Modes property), 192
- fields (aequilibrae.project.network.Nodes property), 198
- fields (aequilibrae.project.network.Periods property), 200

fields (*aequilibrae.project.Zoning* property), 189  
 fields() (*aequilibrae.project.network.LinkTypes* method), 194  
 file\_default (*aequilibrae.Parameters* attribute), 207  
 fit() (*aequilibrae.distribution.Ipf* method), 208  
 from\_db() (*aequilibrae.transit.TransitGraphBuilder* class method), 249  
 from\_path() (*aequilibrae.project.Project* class method), 178  
 functions\_available() (*aequilibrae.paths.VDF* method), 232

## G

get() (*aequilibrae.project.network.Links* method), 195  
 get() (*aequilibrae.project.network.LinkTypes* method), 194  
 get() (*aequilibrae.project.network.Modes* method), 192  
 get() (*aequilibrae.project.network.Nodes* method), 197  
 get() (*aequilibrae.project.network.Periods* method), 199  
 get() (*aequilibrae.project.Zoning* method), 189  
 get\_by\_name() (*aequilibrae.project.network.LinkTypes* method), 194  
 get\_by\_name() (*aequilibrae.project.network.Modes* method), 193  
 get\_closest\_zone() (*aequilibrae.project.Zoning* method), 189  
 get\_graph\_to\_network\_mapping() (*aequilibrae.paths.AssignmentResults* method), 228  
 get\_heuristics() (*aequilibrae.paths.PathResults* method), 232  
 get\_load\_results() (*aequilibrae.paths.AssignmentResults* method), 228  
 get\_load\_results() (*aequilibrae.paths.TransitAssignmentResults* method), 229  
 get\_matrix() (*aequilibrae.matrix.AequilibraeMatrix* method), 217  
 get\_matrix() (*aequilibrae.project.Matrices* method), 183  
 get\_record() (*aequilibrae.project.Matrices* method), 183  
 get\_sl\_results() (*aequilibrae.paths.AssignmentResults* method), 228  
 Graph (class in *aequilibrae.paths*), 223  
 graphs (*aequilibrae.transit.Transit* attribute), 245  
 GravityApplication (class in *aequilibrae.distribution*), 209  
 GravityCalibration (class in *aequilibrae.distribution*), 211

## H

HyperpathGenerating (class in *aequilibrae.paths*), 243

info (*aequilibrae.paths.TrafficClass* property), 234  
 info (*aequilibrae.paths.TransitClass* property), 235  
 info() (*aequilibrae.paths.HyperpathGenerating* method), 244  
 info() (*aequilibrae.paths.TrafficAssignment* method), 239  
 info() (*aequilibrae.paths.TransitAssignment* method), 241  
 installation, 251  
 Ipf (class in *aequilibrae.distribution*), 207  
 is\_omx() (*aequilibrae.matrix.AequilibraeMatrix* method), 219

## L

Link (class in *aequilibrae.project.network*), 201  
 link\_types (*aequilibrae.project.Network* attribute), 185  
 Links (class in *aequilibrae.project.network*), 195  
 LinkType (class in *aequilibrae.project.network*), 201  
 LinkTypes (class in *aequilibrae.project.network*), 193  
 list() (*aequilibrae.project.Matrices* method), 183  
 list\_fields() (*aequilibrae.project.About* method), 180  
 list\_modes() (*aequilibrae.project.Network* method), 185  
 load() (*aequilibrae.distribution.SyntheticGravityModel* method), 213  
 load() (*aequilibrae.matrix.AequilibraeData* method), 214  
 load() (*aequilibrae.matrix.AequilibraeMatrix* method), 219  
 load() (*aequilibrae.project.Project* method), 178  
 load() (*aequilibrae.transit.Transit* method), 246  
 load\_from\_disk() (*aequilibrae.paths.Graph* method), 224  
 load\_from\_disk() (*aequilibrae.paths.TransitGraph* method), 226  
 Log (class in *aequilibrae.project*), 182  
 log() (*aequilibrae.project.Project* method), 179  
 log\_specification() (*aequilibrae.paths.TrafficAssignment* method), 239  
 log\_specification() (*aequilibrae.paths.TransitAssignment* method), 242  
 lonlat (*aequilibrae.project.network.Nodes* property), 198

## M

Matrices (class in *aequilibrae.project*), 182  
 Mode (class in *aequilibrae.project.network*), 200  
 Modes (class in *aequilibrae.project.network*), 191  
 module  
     aequilibrae, 177



## N

nan\_to\_num() (*aequilibrae.matrix.AequilibraeMatrix* method), 221

netsignal (*aequilibrae.project.Network* attribute), 185

Network (class in *aequilibrae.project*), 184

new() (*aequilibrae.project.network.Links* method), 196

new() (*aequilibrae.project.network.LinkTypes* method), 194

new() (*aequilibrae.project.network.Modes* method), 193

new() (*aequilibrae.project.Project* method), 178

new() (*aequilibrae.project.Zoning* method), 188

new\_centroid() (*aequilibrae.project.network.Nodes* method), 198

new\_gtfs\_builder() (*aequilibrae.transit.Transit* method), 245

new\_period() (*aequilibrae.project.network.Periods* method), 199

new\_record() (*aequilibrae.project.Matrices* method), 184

Node (class in *aequilibrae.project.network*), 203

Nodes (class in *aequilibrae.project.network*), 197

## O

open() (*aequilibrae.project.Project* method), 178

OptimalStrategies (class in *aequilibrae.paths*), 244

## P

parameters (*aequilibrae.project.Project* property), 179

Parameters (class in *aequilibrae*), 206

PathResults (class in *aequilibrae.paths*), 230

Period (class in *aequilibrae.project.network*), 205

Periods (class in *aequilibrae.project.network*), 198

prepare() (*aequilibrae.paths.AssignmentResults* method), 227

prepare() (*aequilibrae.paths.PathResults* method), 231

prepare() (*aequilibrae.paths.SkimResults* method), 230

prepare() (*aequilibrae.paths.TransitAssignmentResults* method), 229

prepare\_graph() (*aequilibrae.paths.Graph* method), 224

prepare\_graph() (*aequilibrae.paths.TransitGraph* method), 226

Project (class in *aequilibrae.project*), 177

project\_parameters (*aequilibrae.project.Project* property), 179

protected\_fields (*aequilibrae.project.Network* attribute), 185

pt\_con (*aequilibrae.transit.Transit* attribute), 245

## R

random\_name() (*aequilibrae.matrix.AequilibraeData* static method), 215

random\_name() (*aequilibrae.matrix.AequilibraeMatrix* static method), 222

refresh() (*aequilibrae.project.network.Links* method), 196

refresh() (*aequilibrae.project.network.Nodes* method), 198

refresh() (*aequilibrae.project.network.Periods* method), 199

refresh\_fields() (*aequilibrae.project.network.Links* method), 196

refresh\_fields() (*aequilibrae.project.network.Nodes* method), 198

refresh\_fields() (*aequilibrae.project.network.Periods* method), 199

refresh\_geo\_index() (*aequilibrae.project.Zoning* method), 189

reload() (*aequilibrae.project.Matrices* method), 183

remove() (*aequilibrae.project.FieldEditor* method), 181

renumber() (*aequilibrae.project.network.Node* method), 204

renumber() (*aequilibrae.project.network.Period* method), 205

report() (*aequilibrae.paths.TrafficAssignment* method), 240

report() (*aequilibrae.paths.TransitAssignment* method), 242

req\_link\_flds (*aequilibrae.project.Network* attribute), 185

req\_node\_flds (*aequilibrae.project.Network* attribute), 185

reset() (*aequilibrae.paths.AssignmentResults* method), 228

reset() (*aequilibrae.paths.PathResults* method), 232

reset() (*aequilibrae.paths.TransitAssignmentResults* method), 229

restore\_default() (*aequilibrae.Parameters* method), 207

results() (*aequilibrae.paths.TrafficAssignment* method), 239

results() (*aequilibrae.paths.TransitAssignment* method), 242

rows() (*aequilibrae.matrix.AequilibraeMatrix* method), 220

run() (*aequilibrae.paths.HyperpathGenerating* method), 244

## S

save() (*aequilibrae.distribution.SyntheticGravityModel* method), 213

save() (*aequilibrae.matrix.AequilibraeMatrix* method), 216

save() (*aequilibrae.project.FieldEditor* method), 181

save() (*aequilibrae.project.network.Link* method), 202

save() (*aequilibrae.project.network.Links* method), 196

`save()` (*aequilibrae.project.network.LinkType* method), 201  
`save()` (*aequilibrae.project.network.LinkTypes* method), 194  
`save()` (*aequilibrae.project.network.Mode* method), 201  
`save()` (*aequilibrae.project.network.Node* method), 204  
`save()` (*aequilibrae.project.network.Nodes* method), 198  
`save()` (*aequilibrae.project.network.Period* method), 205  
`save()` (*aequilibrae.project.network.Periods* method), 200  
`save()` (*aequilibrae.project.Zone* method), 190  
`save()` (*aequilibrae.project.Zoning* method), 189  
`save()` (*aequilibrae.transit.TransitGraphBuilder* method), 249  
`save_compressed_correspondence()` (*aequilibrae.paths.Graph* method), 224  
`save_compressed_correspondence()` (*aequilibrae.paths.TransitGraph* method), 226  
`save_config()` (*aequilibrae.transit.TransitGraphBuilder* method), 249  
`save_edges()` (*aequilibrae.transit.TransitGraphBuilder* method), 249  
`save_graphs()` (*aequilibrae.transit.Transit* method), 246  
`save_results()` (*aequilibrae.paths.HyperpathGenerating* method), 244  
`save_results()` (*aequilibrae.paths.TrafficAssignment* method), 239  
`save_results()` (*aequilibrae.paths.TransitAssignment* method), 242  
`save_select_link_flows()` (*aequilibrae.paths.TrafficAssignment* method), 240  
`save_select_link_matrices()` (*aequilibrae.paths.TrafficAssignment* method), 240  
`save_select_link_results()` (*aequilibrae.paths.TrafficAssignment* method), 240  
`save_skims()` (*aequilibrae.paths.TrafficAssignment* method), 239  
`save_to_disk()` (*aequilibrae.paths.AssignmentResults* method), 228  
`save_to_disk()` (*aequilibrae.paths.Graph* method), 224  
`save_to_disk()` (*aequilibrae.paths.TransitGraph* method), 226  
`save_to_project()` (*aequilibrae.distribution.GravityApplication* method), 211  
`save_to_project()` (*aequilibrae.distribution.Ipf* method), 208  
`save_vertices()` (*aequilibrae.transit.TransitGraphBuilder* method), 249  
`select_link_flows()` (*aequilibrae.paths.TrafficAssignment* method), 239  
`set_algorithm()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_algorithm()` (*aequilibrae.paths.TransitAssignment* method), 241  
`set_blocked_centroid_flows()` (*aequilibrae.paths.Graph* method), 224  
`set_blocked_centroid_flows()` (*aequilibrae.paths.TransitGraph* method), 226  
`set_capacity_field()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_classes()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_classes()` (*aequilibrae.paths.TransitAssignment* method), 242  
`set_cores()` (*aequilibrae.paths.AssignmentResults* method), 228  
`set_cores()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_cores()` (*aequilibrae.paths.TransitAssignment* method), 241  
`set_cores()` (*aequilibrae.paths.TransitAssignmentResults* method), 229  
`set_demand_matrix_core()` (*aequilibrae.paths.TransitClass* method), 235  
`set_fixed_cost()` (*aequilibrae.paths.TrafficClass* method), 234  
`set_frequency_field()` (*aequilibrae.paths.TransitAssignment* method), 242  
`set_graph()` (*aequilibrae.paths.Graph* method), 225  
`set_graph()` (*aequilibrae.paths.TransitGraph* method), 227  
`set_heuristic()` (*aequilibrae.paths.PathResults* method), 232  
`set_index()` (*aequilibrae.matrix.AequilibraeMatrix* method), 218  
`set_modes()` (*aequilibrae.project.network.Link* method), 202  
`set_path_file_format()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_pce()` (*aequilibrae.paths.TrafficClass* method), 234  
`set_save_path_files()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_select_links()` (*aequilibrae.paths.TrafficClass* method), 234  
`set_skimming()` (*aequilibrae.paths.Graph* method), 225  
`set_skimming()` (*aequilibrae.paths.TransitGraph* method), 227  
`set_time_field()` (*aequilibrae.paths.TrafficAssignment* method), 238

`set_time_field()` (*aequilibrae.paths.TransitAssignment* method), 242  
`set_time_field()` (*aequilibrae.project.Network* method), 187  
`set_vdf()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_vdf_parameters()` (*aequilibrae.paths.TrafficAssignment* method), 238  
`set_vot()` (*aequilibrae.paths.TrafficClass* method), 234  
`setDescription()` (*aequilibrae.matrix.AequilibraeMatrix* method), 222  
`setName()` (*aequilibrae.matrix.AequilibraeMatrix* method), 221  
`setup()` (in module *aequilibrae*), 177  
`skimmable_fields()` (*aequilibrae.project.Network* method), 185  
`SkimResults` (class in *aequilibrae.paths*), 229  
`sql` (*aequilibrae.project.network.Links* attribute), 195  
`sql` (*aequilibrae.project.network.Nodes* attribute), 197  
`sql` (*aequilibrae.project.network.Periods* attribute), 199  
`SyntheticGravityModel` (class in *aequilibrae.distribution*), 213

## T

`to_transit_graph()` (*aequilibrae.transit.TransitGraphBuilder* method), 249  
`total_flows()` (*aequilibrae.paths.AssignmentResults* method), 228  
`TrafficAssignment` (class in *aequilibrae.paths*), 235  
`TrafficClass` (class in *aequilibrae.paths*), 233  
`Transit` (class in *aequilibrae.transit*), 245  
`TransitAssignment` (class in *aequilibrae.paths*), 240  
`TransitAssignmentResults` (class in *aequilibrae.paths*), 228  
`TransitClass` (class in *aequilibrae.paths*), 234  
`TransitGraph` (class in *aequilibrae.paths*), 225  
`TransitGraphBuilder` (class in *aequilibrae.transit*), 247

## U

`update_database()` (*aequilibrae.project.Matrices* method), 183  
`update_trace()` (*aequilibrae.paths.PathResults* method), 232

## V

`VDF` (class in *aequilibrae.paths*), 232

## W

`write_back()` (*aequilibrae.Parameters* method), 207  
`write_back()` (*aequilibrae.project.About* method), 180

## Z

`Zone` (class in *aequilibrae.project*), 190  
`zoning` (*aequilibrae.project.Project* property), 179  
`Zoning` (class in *aequilibrae.project*), 187