tesspy

Release 0.0.1

tesspy developers

Aug 23, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

`tesspy` is a python library for geographical tessellation.

The process of discretization of space into subspaces without overlaps and gaps is called tessellation and is of interest to researchers in the field of spatial analysis. Tessellation is essential in understanding geographical space and provides a framework for analyzing geospatial data. Different tessellation methods are implemented in *tesspy*. They can be divided into two groups. The first group is regular tessellation methods: square grid and hexagon grid. The second group is irregular tessellation methods based on geospatial data. These methods are adaptive squares, Voronoi diagrams, and city blocks. The geospatial data used for tessellation is retrieved from the OpenStreetMap database.

The package is currently maintained by @siavash-saki and @JoHamann.

https://github.com/siavash-saki/tesspy

CHAPTER

TWO

INSTALL

You can install `tesspy` from PyPI using pip (**Not Recommended**):

```
pip install tesspy
```

and from conda (**Recommended**):

```
conda install tesspy
```

CHAPTER

THREE

CREATING A NEW ENVIRONMENT FOR TESSPY

`tesspy` depends on `geopandas`, which could make the installation sometimes tricky because of the conflicts with the current packages. Therefore, we recommend creating a new clean environment and installing the dependencies from the conda-forge channel.

Create a new environment:

```
conda create -n tesspy_env -c conda-forge
```

Activate this environment:

```
conda activate tesspy_env
```

Install tesspy from conda-forge channel:

```
conda install -c conda-forge tesspy
```

CHAPTER FOUR

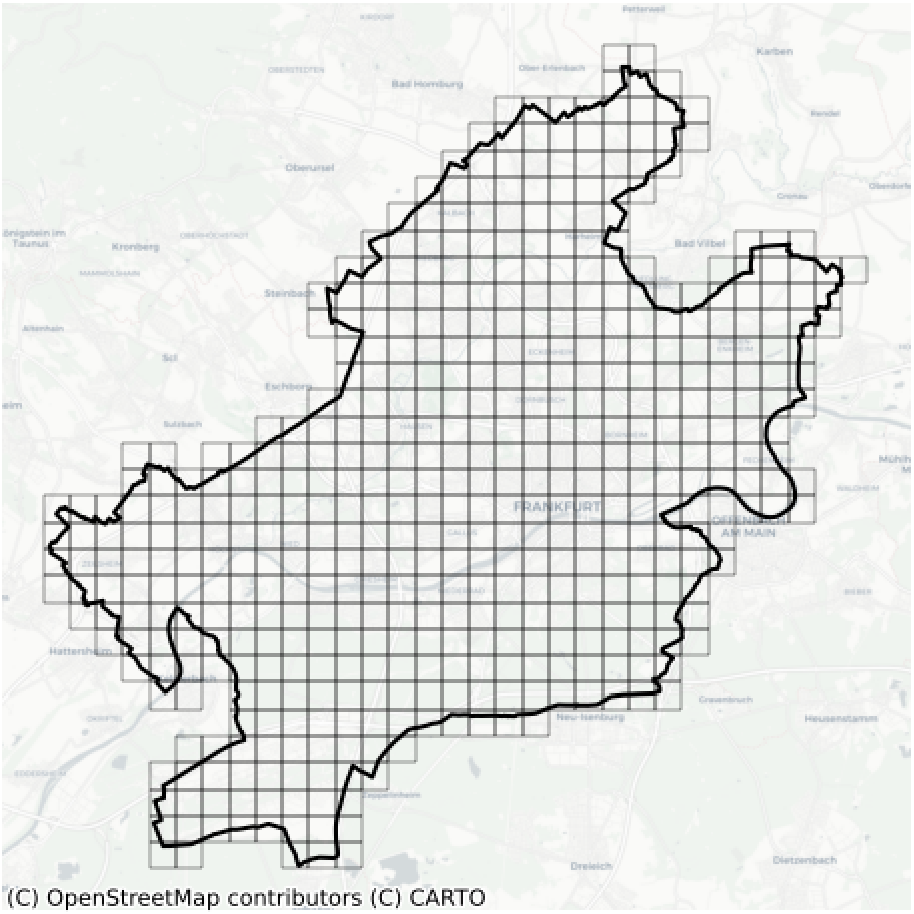
EXAMPLES

The city of "Frankfurt am Main" in Germany is used to showcase different tessellation methods. This is how a tessellation object is built, and different methods are called. For the tessellation methods based on Points of Interests (adaptive squares, Voronoi polygons, and City Blocks), we use `amenity` data from the OpenStreetMap.:

```
from tesspy import Tessellation
ffm= Tessellation('Frankfurt am Main')
```

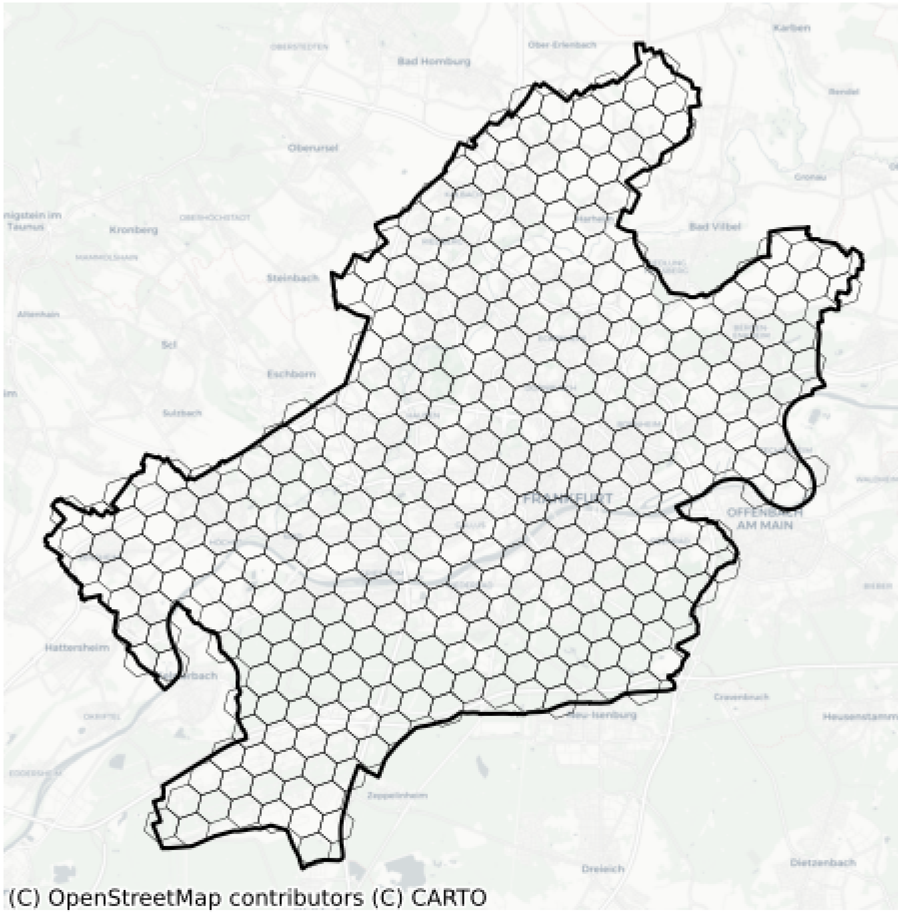
4.1 Squares

```
ffm_sqruares = ffm.squares(resolution=15)
```



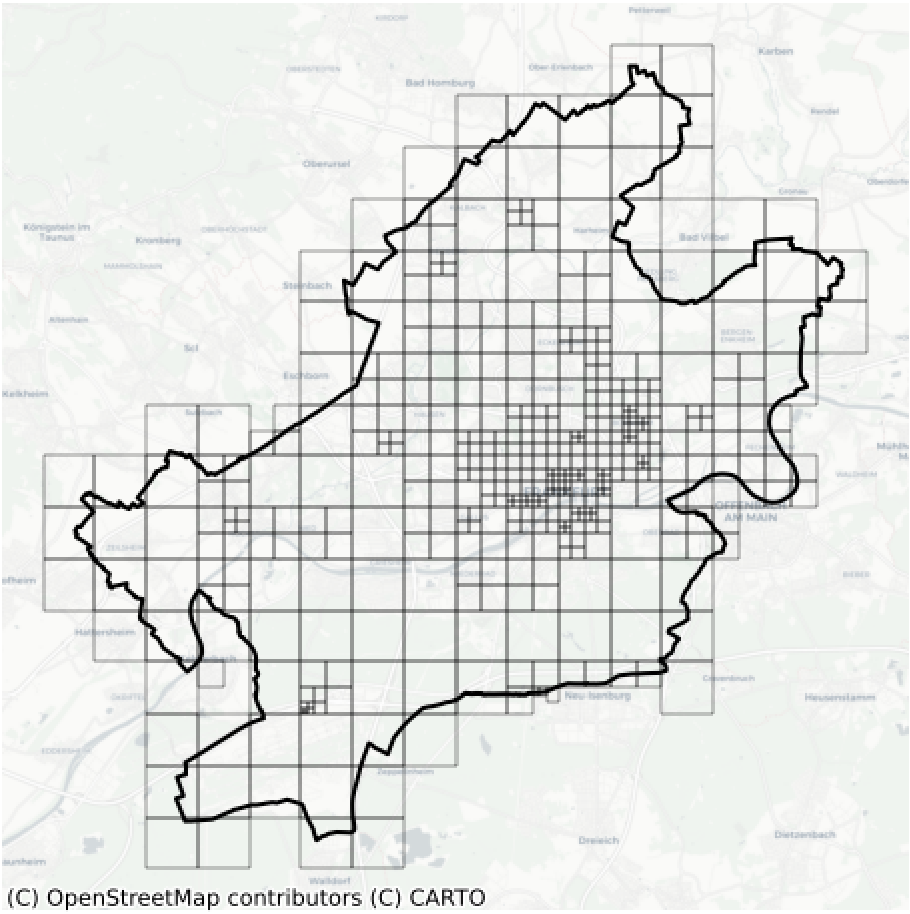
4.2 Hexagons

```
ffm_sqruares = ffm.squares(resolution=15)
```



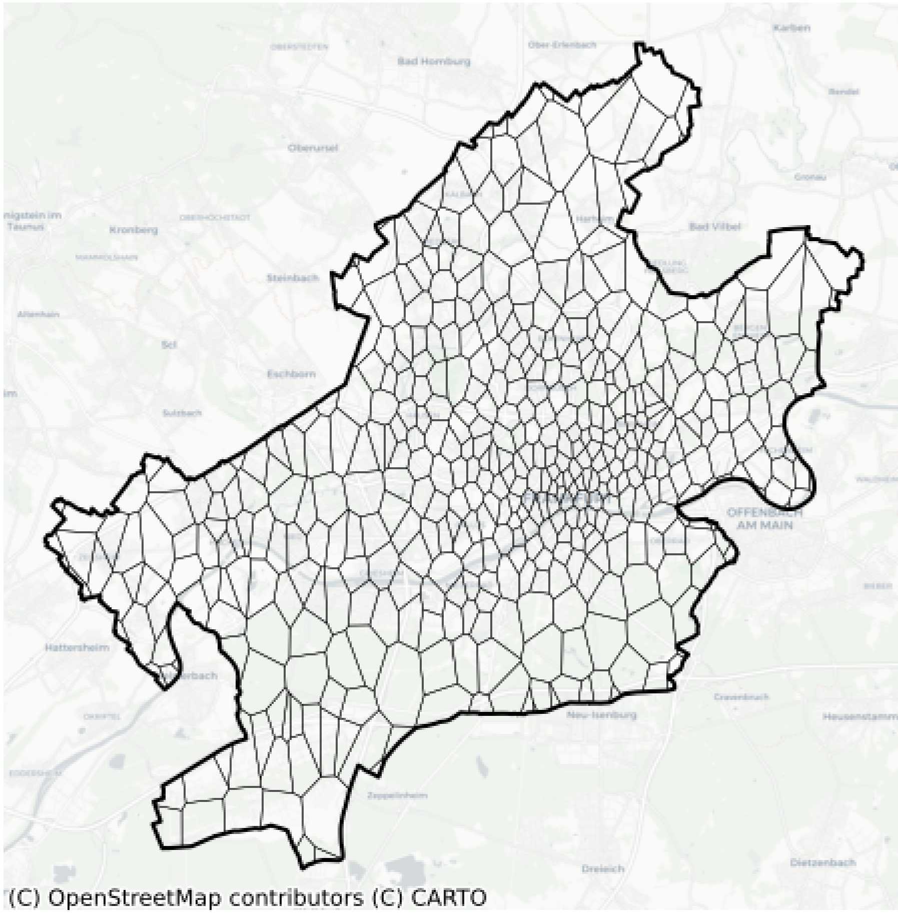
4.3 Adaptive Squares

```
ffm_asq = ffm.adaptive_squares(start_resolution=14, threshold=100, poi_categories=[
↪'amenity'])
```



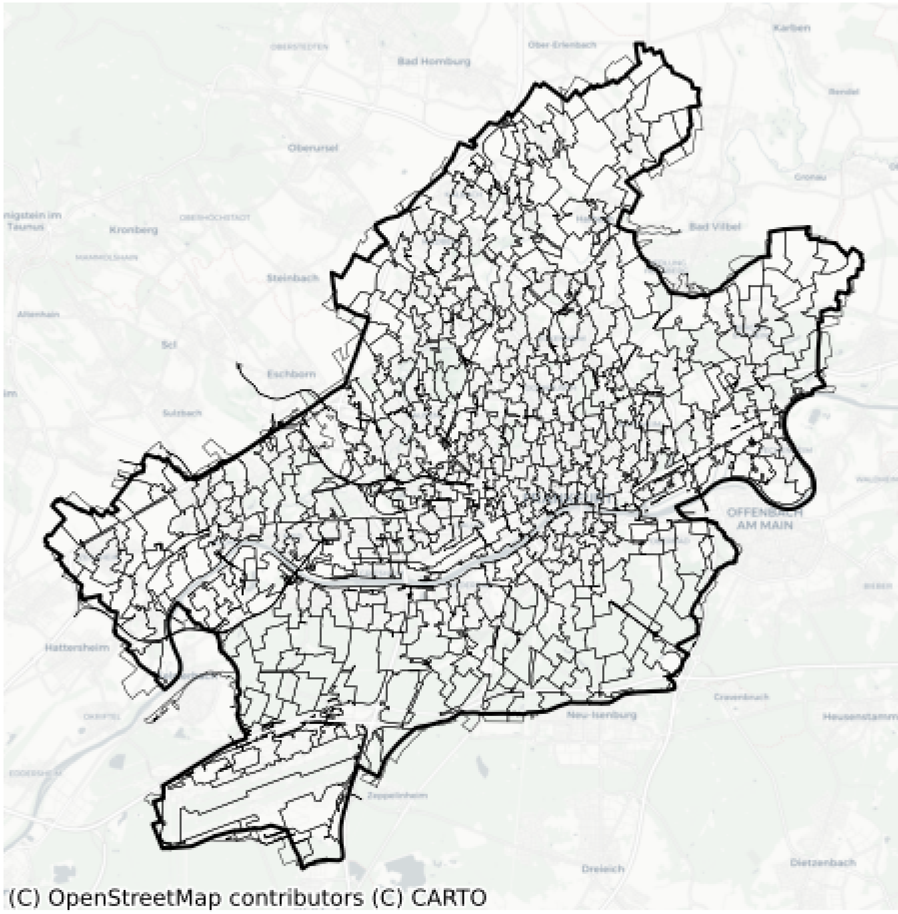
4.4 Voronoi Polygons

```
ffm_sqruares = ffm.squares(resolution=15)
```



4.5 City Blocks

```
ffm_sqruares = ffm.squares(resolution=15)
```



4.5.1 Examples

Getting Started

This example shows the basic functions of `tesspy`, which are five types of tessellations: square, hexagon, adaptive squares, voronoi diagrams and city blocks. We have selected the city of **Frankfurt am Main** in Germany to demonstrate tessellation.

Installation

Creating a new environment for tesspy

`tesspy` depends on `geopandas`, which could make the installation sometimes tricky because of the conflicts with the current packages. Therefore, we recommend creating a new clean environment and installing the dependencies from the conda-forge channel.

Create a new environment:

```
conda create -n tesspy_env -c conda-forge
```

Activate this environment:

```
conda activate tesspy_env
```

Install tesspy from conda-forge channel:

```
conda install -c conda-forge tesspy
```

Extra dependencies for tutorials

There are several extra python packages necessary for running the tutorials: `contextlily`, `esda`, `libpysal`, `matplotlib`, `seaborn`, and `statsmodels`. While some are already installed during the installation of tesspy as dependencies, the others should be manually installed in the same conda environment, `tesspy_env`. These packages are all available on the conda-forge channel. A list of required dependencies can be found in `Examples\requirements_tutorials.txt` and `Examples\tutiruals_env.yaml`.

If you want to follow the tutorials, you can simply use one of these files to create an environment for running the jupyter notebooks.

There are some different ways to do this.

- If you already installed `tesspy` in the new environment `tesspy_env` as explained above, you can install the rest of the dependencies using:

  ```
  conda install -c conda-forge seaborn contextily esda libpysal statsmodels
  ```

- If you want to do it from scratch, you can create an environment. Then install the dependencies using one of the files provided and then install `tesspy`:

1. First, create an environment, add `conda-forge`, and set channel priority to strict. This way, you make sure all the dependencies are installed from `conda-forge` channel.

   ```
   conda create -n tesspy_env
   conda activate tesspy_env
   conda config --env --add channels conda-forge
   conda config --env --set channel_priority strict
   ```

2. Install dependencies using either:

   ```
   conda install --file requirements_tutorials.txt
   ```

3. Install `tesspy`:

   ```
   conda install tesspy
   ```

Now, your environment is ready for running all the example jupyter notebooks. Please note that one of the dependencies of the `tesspy` has different names in pypi and conda. This dependency is called `h3` in pypi and `h3-py` in conda. Therefore, using `pip` to install the dependencies raise an error.

Tessellation of Frankfurt am Main

We start by importing the `Tessellation` object from `tesspy` module:

[2]:
```
from tesspy import Tessellation
```

[3]:
```
# Shapely 1.8.1 makes pandas to produce many warnings; this is to get rid of these␣
↪warnings
import warnings

warnings.simplefilter("ignore")
```

Defining the area

There are different ways to define the area. The most straightforward way is passing an address.

[4]:
```
ffm = Tessellation("Frankfurt am Main")
```

With `.get_polygon()` method, we can retrieve the polygon. This is in the form of a GeoPandas GeoDataFrame. We visualize the polygon to make sure data collection was successful and correct:

[5]:
```
ffm.get_polygon().plot(figsize=(10, 10)).set_axis_off();
```

nbsphinx-code-borderwhite

The polygon shows the city of Frankfurt. Note that the CRS of the GeoPandas is EPSG:4326. We can double-check that:

[6]:
```
print(ffm.get_polygon().crs)
```

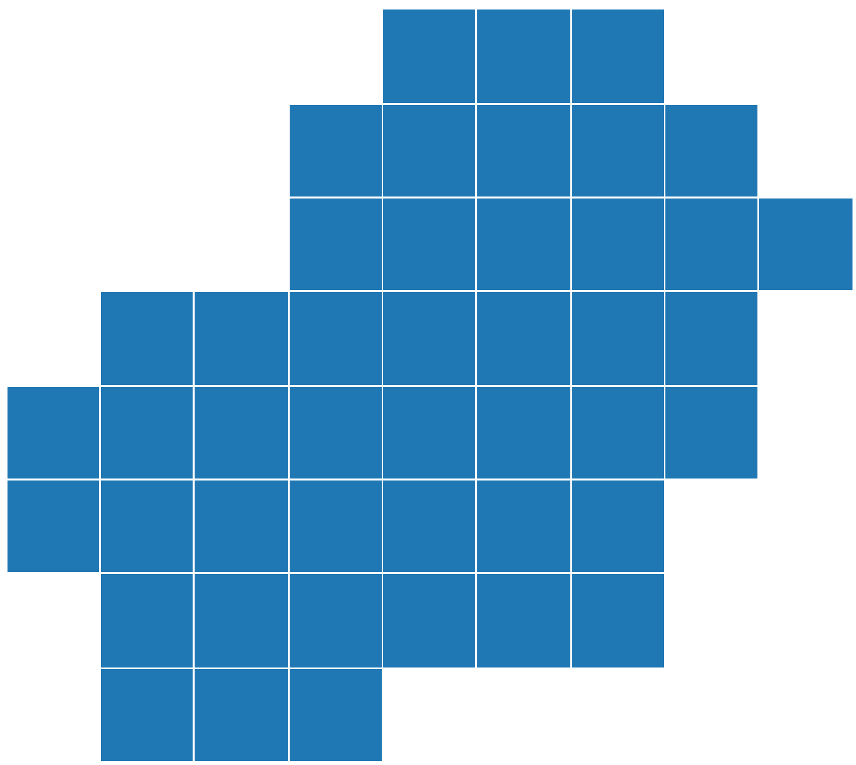
```
epsg:4326
```

It is already ready for tessellation.

Squares

The first tessellation method is the square grid. It creates (almost) equal squares that cover the whole area surface. In order to set the size, we need to pass a `resolution`. This value is usually between 1 and 21. Larger values mean smaller squares and consequently a finer tessellation. For example, with a `resolution=13`, we have:

[7]:
```
ffm_sqr_13 = ffm.squares(13)
ffm_sqr_13.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

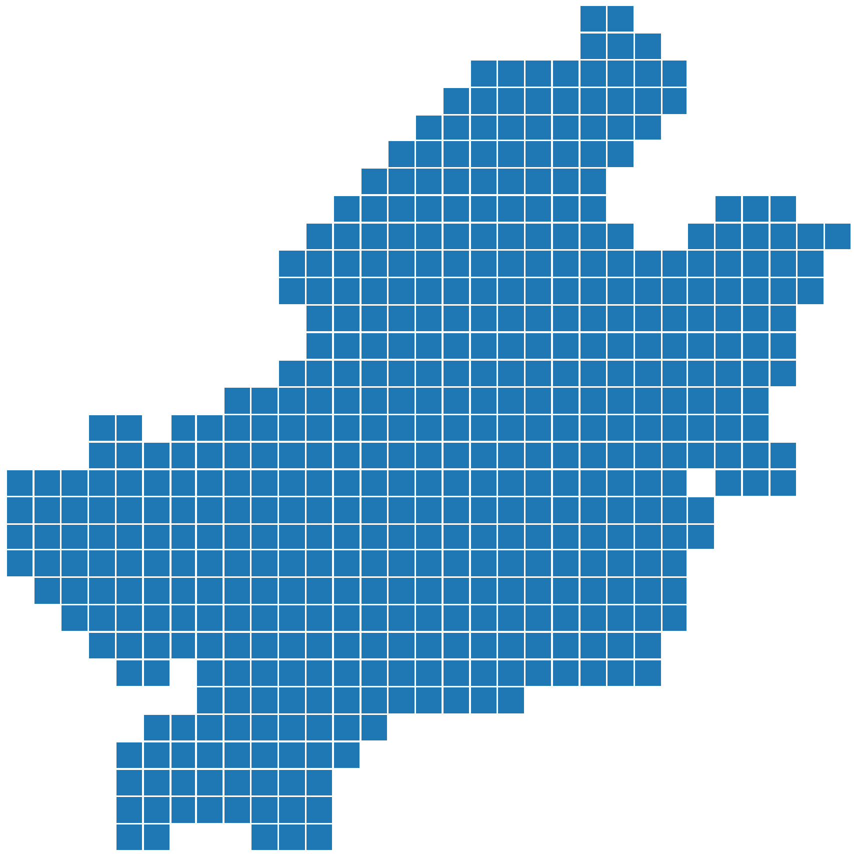
[8]:
```
ffm_sqr_13.head()
```

[8]:

	geometry	quadkey
0	POLYGON ((8.48145 50.09239, 8.48145 50.12058, ...	1202033020220
1	POLYGON ((8.48145 50.06419, 8.48145 50.09239, ...	1202033020222
2	POLYGON ((8.52539 50.12058, 8.52539 50.14875, ...	1202033020203
3	POLYGON ((8.52539 50.09239, 8.52539 50.12058, ...	1202033020221
4	POLYGON ((8.52539 50.06419, 8.52539 50.09239, ...	1202033020223

The square grid with a `resolution=15` look like this:

```
[9]: ffm_sqr_15 = ffm.squares(15)
     ffm_sqr_15.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

As it can be seen the number of square are relatively higher when resolution is higher. We can check the number of squares for each resolution:

```
[10]: print("Resolution=13  ==>  Number of squares:  ", ffm_sqr_13.shape[0])
      print("Resolution=15  ==>  Number of squares:  ", ffm_sqr_15.shape[0])
```

```
Resolution=13  ==>  Number of squares:   45
Resolution=15  ==>  Number of squares:   488
```

hexagons

The next method is hexagons. Similar to squares, this method creates (almost) equal shapes to cover the area. A slight difference is that the algorithm finds the optimal number of hexagons, and not all the surface is covered. Therefore, there may be some areas on the borders which are left behind.

Similarly, we need to pass a resolution to set the hexagon sizes. The larger numbers mean smaller hexagons. A suitable value is usually between 5 and 15. Here is the hexagon tessellation with `resolution=7`:

[11]:
```
ffm_hex_7 = ffm.hexagons(7)
ffm_hex_7.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```

nbsphinx-code-borderwhite

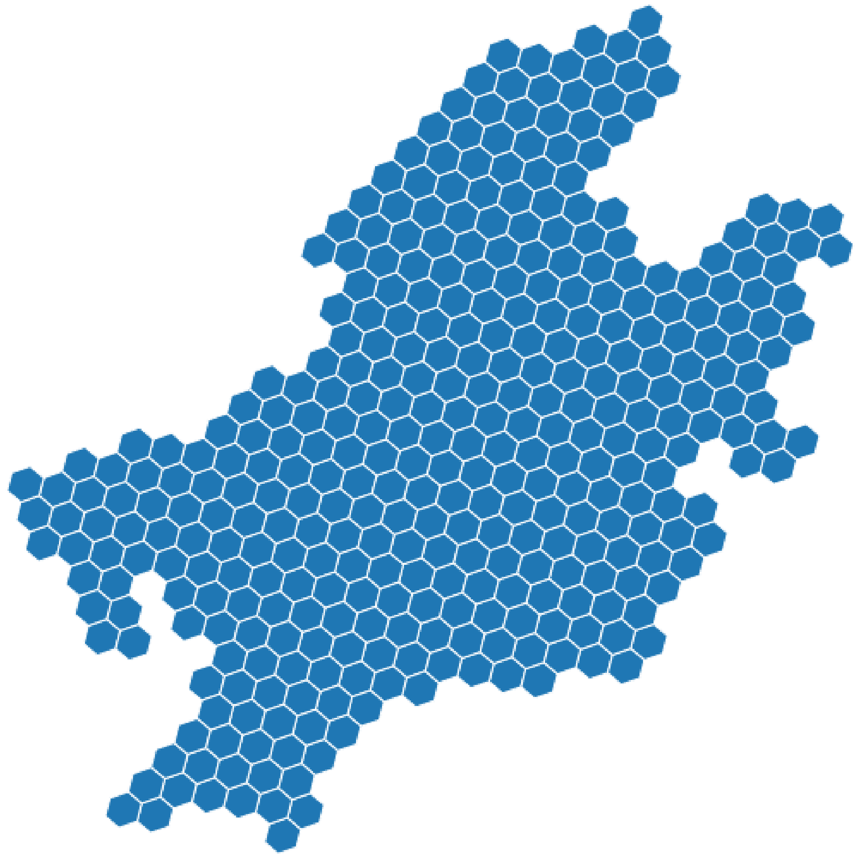
[12]:
```
ffm_hex_7.head()
```

[12]:

	hex_id	geometry
0	871faeaaeffffff	POLYGON ((8.59585 50.10425, 8.59746 50.09256, ...
1	871faeaa3ffffff	POLYGON ((8.56260 50.10080, 8.56422 50.08911, ...
2	871faeaadffffff	POLYGON ((8.64334 50.12694, 8.64494 50.11525, ...
3	871faea13ffffff	POLYGON ((8.62429 50.14275, 8.62590 50.13107, ...
4	871faea02ffffff	POLYGON ((8.65276 50.18123, 8.65437 50.16955, ...

At the next resolution, in this case, `resolution=8`, each hexagon is divided into 7 sub-hexagons. This means the total number of hexagons is (almost) 7 times more when resolution increments one unit. Following plot shows the hexagons of Frankfurt with `resolution=8`:

[13]:
```
ffm_hex_8 = ffm.hexagons(8)
ffm_hex_8.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

[14]:
```
print("Resolution=7  ==>  Number of hexagons:  ", ffm_hex_7.shape[0])
print("Resolution=8  ==>  Number of hexagons:  ", ffm_hex_8.shape[0])
```

```
Resolution=7  ==>  Number of hexagons:   53
Resolution=8  ==>  Number of hexagons:   366
```

Adaptive squares

Adaptive squares are an extension of regular squares. This method starts with relatively large squares and uses the spatial data, i.e., Points of Interests (POI), to divide the high-density squares into 4 subsquares. The division of squares is done until a specific threshold for POI count per square is reached.

The spatial data are retrieved from OpenStreetMap (OSM). We can use which POI categories we want to use. The selected POI categories should be passed as a list. These categories are the OSM primary categories, which represent physical objects on the map. More information can be found at this link.

You can see the top-level categories by:

[15]:
```
print(Tessellation.osm_primary_features())
```

```
['aerialway', 'aeroway', 'amenity', 'barrier', 'boundary', 'building', 'craft',
↪'emergency', 'geological', 'healthcare', 'highway', 'historic', 'landuse', 'leisure',
↪'man_made', 'military', 'natural', 'office', 'place', 'power', 'public_transport',
↪'railway', 'route', 'shop', 'sport', 'telecom', 'tourism', 'water', 'waterway']
```

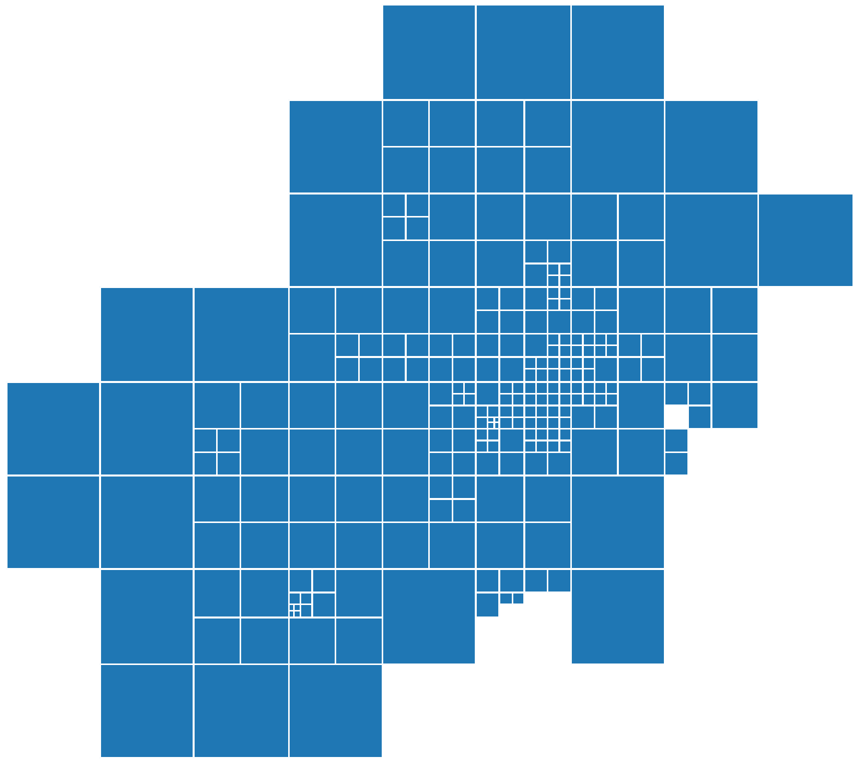
So we build the adaptive squares for

[16]:
```python
# Adaptive Squares using only amenity data
ffm_asq = ffm.adaptive_squares(
    start_resolution=13,
    poi_categories=["amenity"],
    threshold=None,
    timeout=60,
    verbose=True,
)
```

```
Getting data from OSM...
Creating POI DataFrame...
Cleaning POI DataFrame...
Threshold=198  ==> set as the median POI-count per square at the initial level
Threshold exceeded! Squares are subdivided into resolution 14
Threshold exceeded! Squares are subdivided into resolution 15
Threshold exceeded! Squares are subdivided into resolution 16
Threshold exceeded! Squares are subdivided into resolution 17
```

We visualize the adaptive squares:

[17]:
```python
ffm_asq.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

[18]:
```
ffm_asq.head()
```

[18]:
```
          quadkey  count                                           geometry
0   1202033020033     83  POLYGON ((8.61328 50.17690, 8.61328 50.20503, ...
1   1202033020120     32  POLYGON ((8.65723 50.20503, 8.65723 50.23315, ...
2   1202033020121     39  POLYGON ((8.70117 50.20503, 8.70117 50.23315, ...
3  12020330201220      5  POLYGON ((8.63525 50.19097, 8.63525 50.20503, ...
4  12020330201221     49  POLYGON ((8.65723 50.19097, 8.65723 50.20503, ...
```

Voronoi-Diagrams

Voronoi-Diagram is a method for tessellation that uses irregular shapes to cover the area. In this method, we have a given set of points, called generators. For each generator, there is a polygon that contains the area that is closer to this generator than other generators.

In this context, the generators are the POI, coming from OSM as explained for adaptive squares. Usually, the number of POI is larger than the required number of polygons. So, we can use a clustering method to cluster POI in the first step and then use the cluster centroids as the generator points.

In the example below, we use `shop` and `public_transport` as POI. We use the `k-means` clustering algorithm to cluster the POI and build the generators. The number of polygons is set to 100.

[19]:
```
ffm_voronoi = ffm.voronoi(
    cluster_algo="k-means",
    poi_categories=["shop", "public_transport"],
```

(continues on next page)

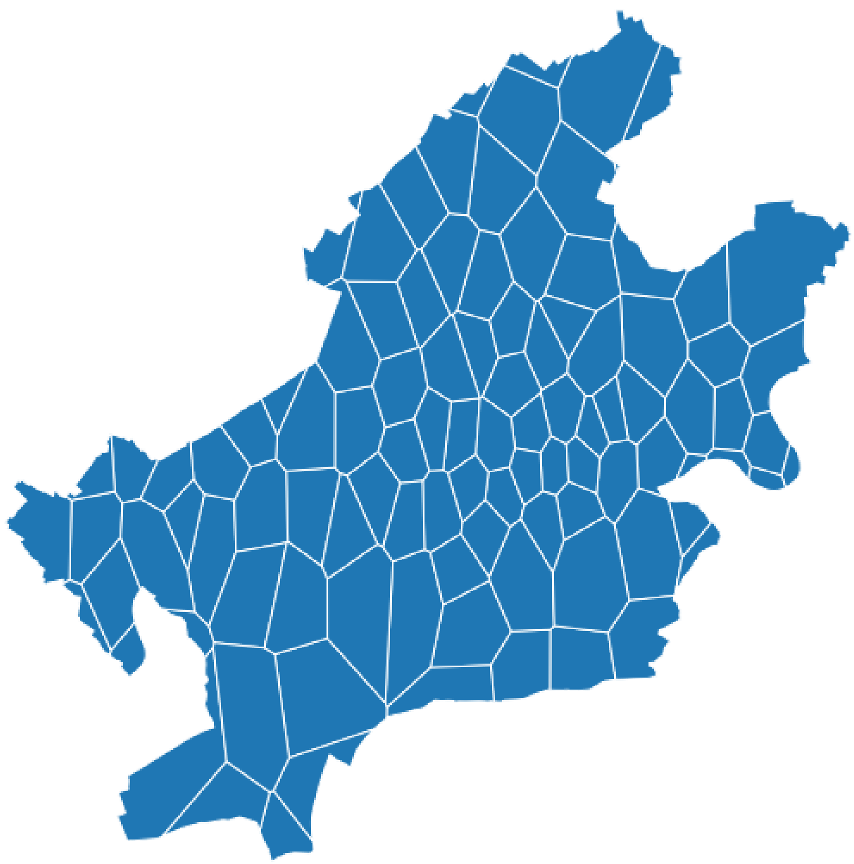
(continued from previous page)

```
    timeout=60,
    n_polygons=100,
    verbose=True,
)
```

```
Getting data from OSM...
Creating POI DataFrame...
Cleaning POI DataFrame...
K-Means Clustering...
Creating Voronoi polygons...
```

We can now plot and visualize the voronoi polygons:

[20]:
```
ffm_voronoi.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

[21]:
```
ffm_voronoi.head()
```

[21]:
```
                                            geometry
0  POLYGON ((8.59543 50.08904, 8.58224 50.09516, ...
1  POLYGON ((8.72848 50.13084, 8.71238 50.14025, ...
2  POLYGON ((8.65669 50.13082, 8.64616 50.15157, ...
3  POLYGON ((8.65870 50.10552, 8.66003 50.11245, ...
4  POLYGON ((8.51751 50.09623, 8.50234 50.08454, ...
```

```
[22]: print("Number of polygons ==> ", len(ffm_voronoi))
```

```
Number of polygons ==>  100
```

City blocks

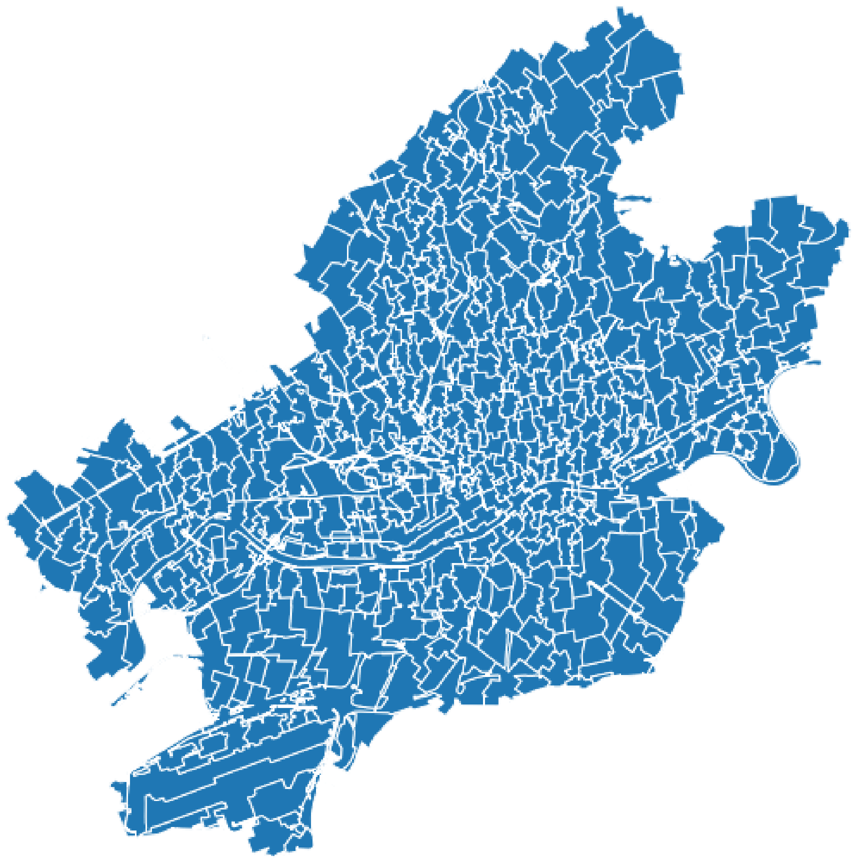
The last tessellation method is city blocks. We define city blocks as the smallest area surrounded by street segments. This method gets the road network data from OSM and generates polygons based on the roads. We use hierarchical clustering to merge a group of contiguous polygons. This guarantees that tiny polygons like road islands are not identified as single polygons. The number of desired polygons can be passed. This would be an approximation. The final number of city blocks could slightly vary.

For example, the following is the example of city blocks for Frankfurt:

```
[23]: ffm_cb = ffm.city_blocks(
          n_polygons=500, detail_deg=None, split_roads=True, verbose=True
      )
```

```
Selected highway type(s) are ['highway'~
↪'motorway|trunk|primary|secondary|tertiary|residential|unclassified|motorway_
↪link|trunk_link|primary_link|secondary_link|living_street|pedestrian|track|bus_
↪guideway|footway|path|service|cycleway']
Collecting road network data...
Splitting the linestring, such that each linestring has exactly 2 points.
Collected data has 272749 street segments.
Creating initial city blocks using the road network data...
Merging small city blocks...
```

```
[24]: ffm_cb.plot(lw=1, edgecolor="w", figsize=(10, 10)).set_axis_off();
```



nbsphinx-code-borderwhite

[26]:
```
ffm_cb.head()
```

[26]:
```
                                            geometry
0  POLYGON ((8.68232 50.10887, 8.68220 50.10886, ...
1  POLYGON ((8.67485 50.11179, 8.67490 50.11181, ...
2  POLYGON ((8.68545 50.10926, 8.68522 50.10925, ...
3  POLYGON ((8.69298 50.11307, 8.69298 50.11309, ...
4  POLYGON ((8.69076 50.10175, 8.69068 50.10173, ...
```

[27]:
```
print("Number of polygons ==> ", len(ffm_cb))
```

```
Number of polygons ==>  760
```

Tesslating Different Cities

This notebook demonstrates how the implemented tessellation methods function for different cities. We select a wide variety of cities on different continents. For example, cities that consist of multipolygons and islands are also considered.

Selected cities are `Barcelona`, `Key West`, `Nairobi`, `Tehran`.

To run this notebook, in addition to `tesspy`, you need `contextily` for basemap visualization. This package is only used to enhance visualization and has no effect on tessellation.

[3]:
```
from tesspy import Tessellation
import matplotlib.pyplot as plt
import contextily as ctx

from time import sleep
```

[2]:
```
# Shapely 1.8.1 makes pandas to produce many warnings; this is to get rid of these␣
↪warnings
import warnings

warnings.simplefilter("ignore")
```

Getting city polygons

[5]:
```
# Create a tessellation object for each city
barcelona = Tessellation("Barcelona, Spain")
key_west = Tessellation("Key West, United States")
nairobi = Tessellation("Nairobi, Kenia")
tehran = Tessellation("Tehran, Iran")

# get polygone of the investigated area
barcelona_polygon = barcelona.get_polygon()
key_west_polygon = key_west.get_polygon()
nairobi_polygon = nairobi.get_polygon()
tehran_polygon = tehran.get_polygon()
```

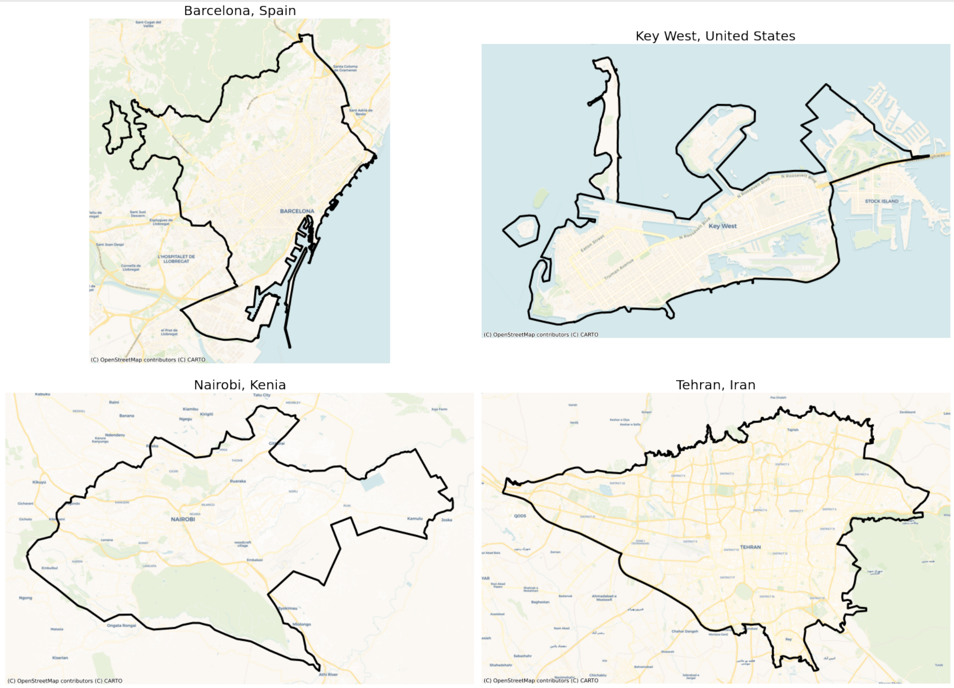
[4]:
```
# visualization of areas
cities_polygons = [barcelona_polygon, key_west_polygon, nairobi_polygon, tehran_polygon]
cities_names = [
    "Barcelona, Spain",
    "Key West, United States",
    "Nairobi, Kenia",
    "Tehran, Iran",
]

fig, axs = plt.subplots(2, 2, figsize=(20, 15))

for ax, city_polygon, city_name in zip(axs.flatten(), cities_polygons, cities_names):
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
    ax.set_title(city_name, fontsize=20)
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
```

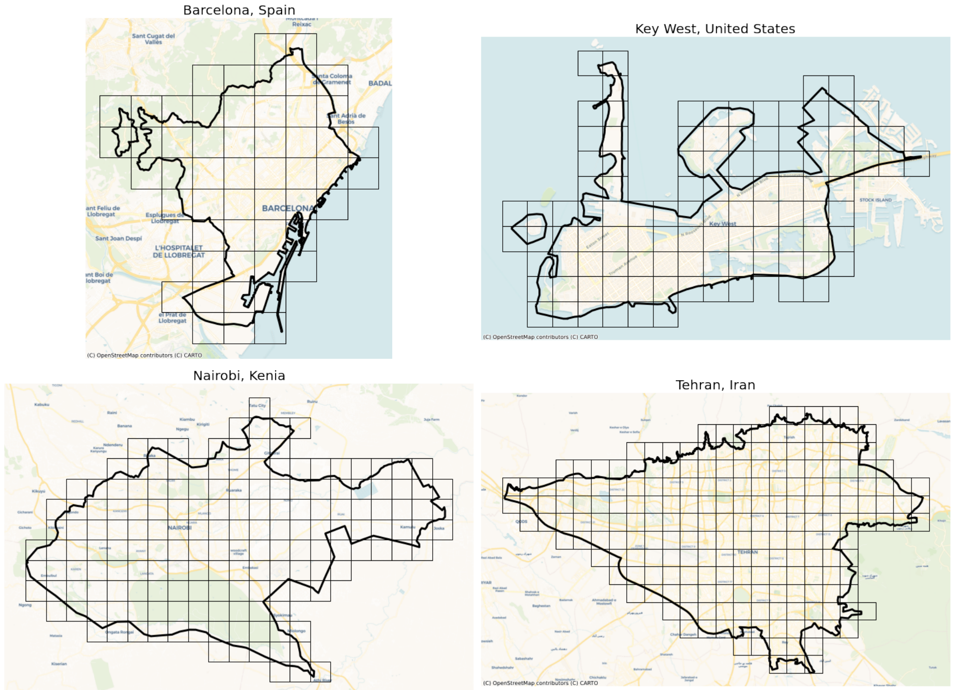
nbsphinx-code-borderwhite



Squares

[5]:
```
# Creating square tessellation

barcelona_squares = barcelona.squares(14)
key_west_squares = key_west.squares(16)
nairobi_squares = nairobi.squares(14)
tehran_squares = tehran.squares(14)
```

[6]:
```
# visualization of square tessellation

cities_squares = [barcelona_squares, key_west_squares, nairobi_squares, tehran_squares]

fig, axs = plt.subplots(2, 2, figsize=(20, 15))

for ax, city_polygon, city_name, city_square in zip(
    axs.flatten(), cities_polygons, cities_names, cities_squares
):
    city_square.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=1)
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
    ax.set_title(city_name, fontsize=20)

plt.tight_layout()
```



nbsphinx-code-borderwhite

Hexagons

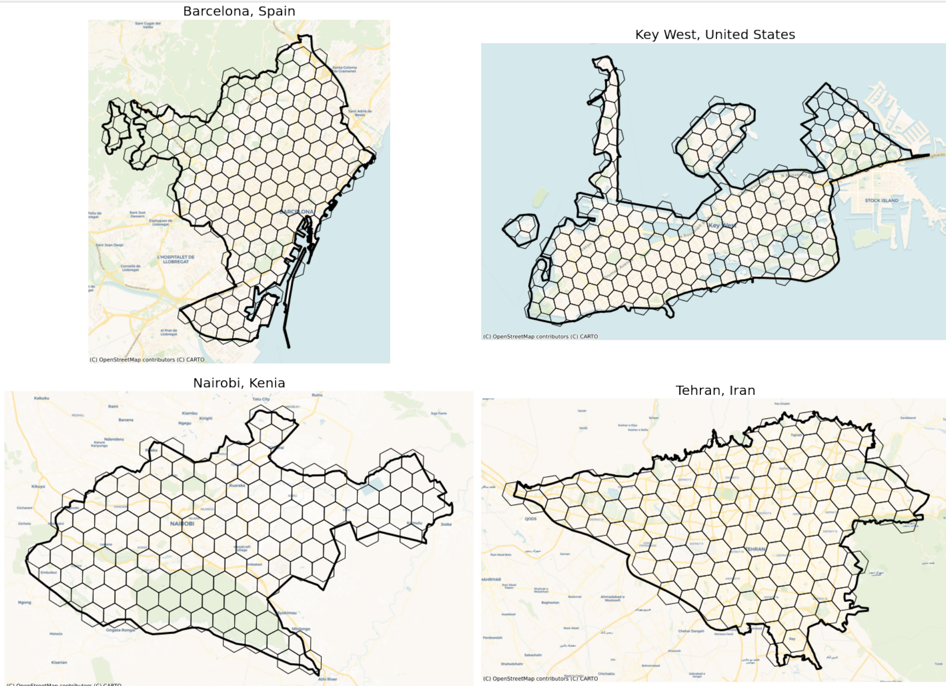
[7]:
```
# Creating hexagons tessellation

barcelona_hexagons = barcelona.hexagons(8)
key_west_hexagons = key_west.hexagons(9)
nairobi_hexagons = nairobi.hexagons(7)
tehran_hexagons = tehran.hexagons(7)
```

[8]:
```
# visualization of hexagons tessellation

cities_hexagons = [
    barcelona_hexagons,
    key_west_hexagons,
    nairobi_hexagons,
    tehran_hexagons,
]

fig, axs = plt.subplots(2, 2, figsize=(20, 15))

for ax, city_polygon, city_name, city_hexagon in zip(
    axs.flatten(), cities_polygons, cities_names, cities_hexagons
):
    city_hexagon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=1)
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
```

(continues on next page)

(continued from previous page)

```
    ax.set_title(city_name, fontsize=20)

plt.tight_layout()
```



nbsphinx-code-borderwhite

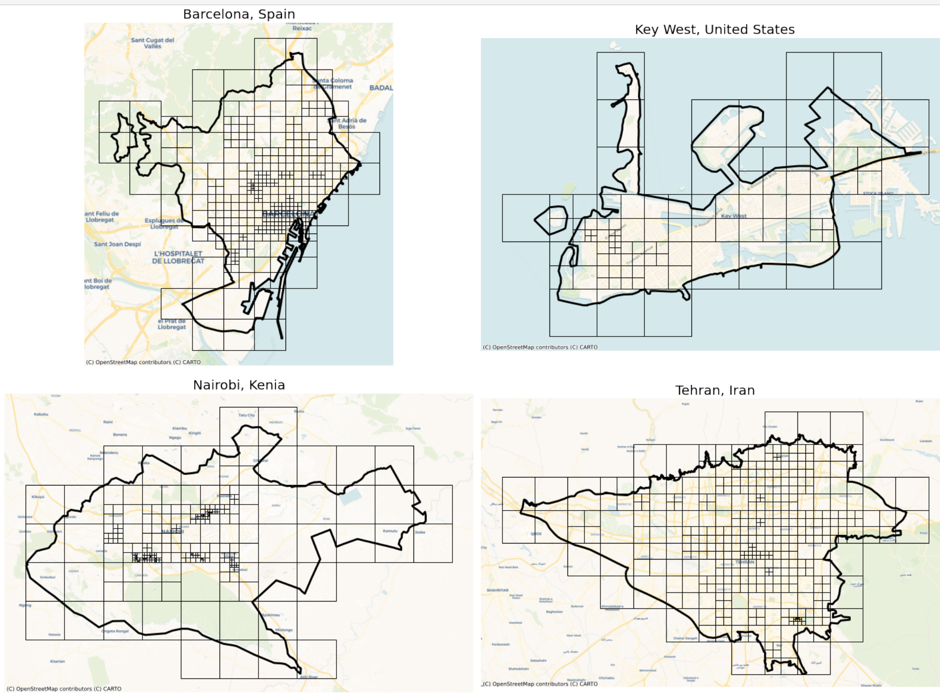
Adaptive squares

In order to retrieve data from OpenStreetMap, we need to follow the fair use policy of overpass to avoid using up all server resources. If we send several requests consequently with the same IP, we get an error. Therefore, after each request, we wait 60 seconds to send the next request.

```
[9]: # Creating adaptive squares tessellation

barcelona_asq = barcelona.adaptive_squares(
    start_resolution=14, poi_categories=["amenity"]
)
sleep(60)
key_west_asq = key_west.adaptive_squares(
    start_resolution=15, poi_categories=["amenity"], threshold=40
)
sleep(60)
nairobi_asq = nairobi.adaptive_squares(
    start_resolution=13, poi_categories=["amenity"], threshold=150
)
sleep(60)
tehran_asq = tehran.adaptive_squares(
    start_resolution=13, poi_categories=["amenity"], threshold=100
)
```

[10]:
```
# visualization of adaptive squares tessellation

cities_asq = [barcelona_asq, key_west_asq, nairobi_asq, tehran_asq]

fig, axs = plt.subplots(2, 2, figsize=(20, 15))

for ax, city_polygon, city_name, city_asq in zip(
    axs.flatten(), cities_polygons, cities_names, cities_asq
):
    city_asq.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=1)
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
    ax.set_title(city_name, fontsize=20)

plt.tight_layout()
```

nbsphinx-code-borderwhite

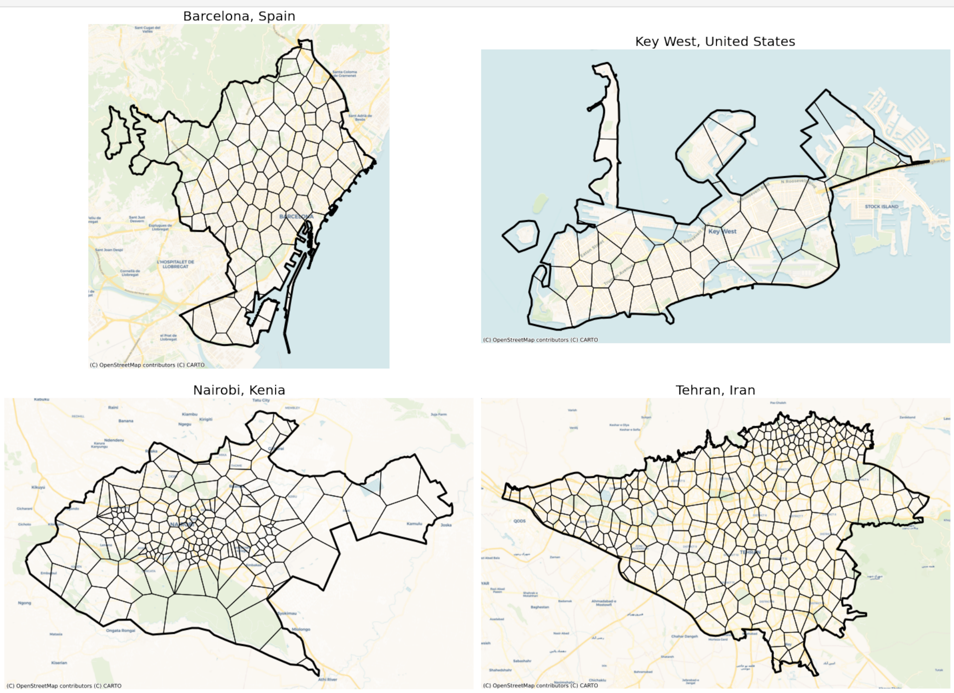


Voronoi Diagrams

[11]:
```
# Creating voronoi tessellation

barcelona_vor = barcelona.voronoi(n_polygons=100, poi_categories=["building"])
sleep(60)
key_west_vor = key_west.voronoi(n_polygons=50, poi_categories=["building"])
sleep(90)
nairobi_vor = nairobi.voronoi(n_polygons=200, poi_categories=["building"])
sleep(120)
tehran_vor = tehran.voronoi(n_polygons=300, poi_categories=["building"])
```

[12]:
```
# visualization of voronoi tessellation

cities_vor = [barcelona_vor, key_west_vor, nairobi_vor, tehran_vor]

fig, axs = plt.subplots(2, 2, figsize=(20, 15))

for ax, city_polygon, city_name, city_vor in zip(
    axs.flatten(), cities_polygons, cities_names, cities_vor
):
    city_vor.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=1)
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
    ax.set_title(city_name, fontsize=20)

plt.tight_layout()
```

nbsphinx-code-borderwhite



City Blocks

[7]:
```
# Creating city blocks tessellation
# Since it takes a lot of time, we only create the city blocks for two cities.

sleep(60)
barcelona_blks = barcelona.city_blocks(n_polygons=500, detail_deg=11)
sleep(60)
nairobi_blks = nairobi.city_blocks(n_polygons=500)
# sleep(60)
# key_west_blks = key_west.city_blocks()
# sleep(60)
```

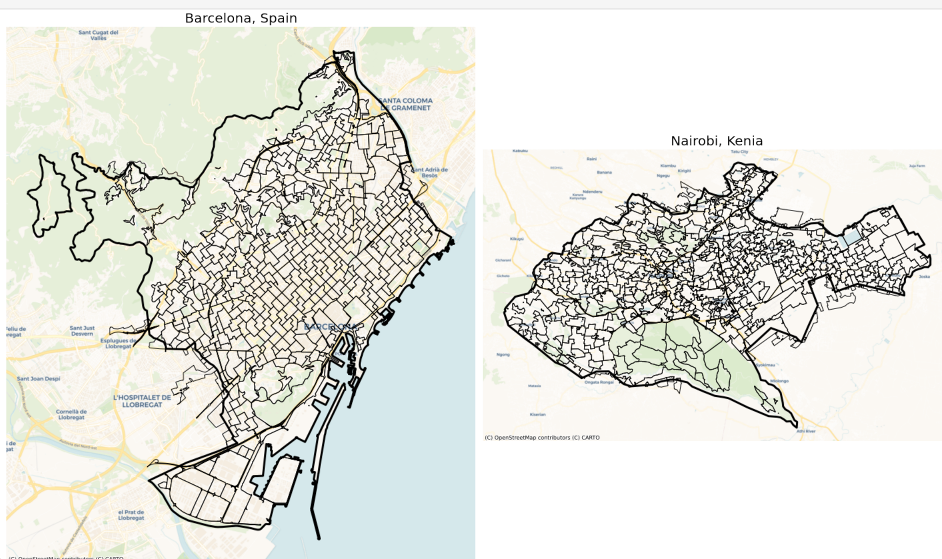
(continues on next page)

(continued from previous page)

```
# tehran_blks = tehran.city_blocks(n_polygons=500, detail_deg=11)
```

[8]:
```
# visualization of city blocks tessellation

cities_blks = [barcelona_blks, nairobi_blks]

cities_polygons = [barcelona_polygon, nairobi_polygon]

cities_names = [
    "Barcelona, Spain",
    "Nairobi, Kenia",
]

fig, axs = plt.subplots(1, 2, figsize=(20, 15))

for ax, city_polygon, city_name, city_blks in zip(
    axs.flatten(), cities_polygons, cities_names, cities_blks
):
    city_blks.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=1)
    city_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="k", lw=3)
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    ax.set_axis_off()
    ax.set_title(city_name, fontsize=20)

plt.tight_layout()
```

nbsphinx-code-borderwhite



Voronoi Diagrams

This notebook shows how Voronoi diagrams can be built based on the points of interest. Points of interest (POI) are physical objects on the map, e.g., restaurants, bars, offices, buildings, etc. The POI can be directly used as generators for Voronoi diagrams. Alternatively, if there are many POI, they can be clustered, and the cluster centroids can be used as generators for Voronoi diagrams. The latter is a more typical case.

The POI are retrieved from the richest open-source spatial database in the world, i.e., the OpenStreetMap database. We use top-level POI categories of OSM for retrieving data. A complete list of these categories can be found at OSM wiki.

To run this notebook, in addition to `tesspy`, you need `contextily` for basemap visualization. This package is only used to enhance visualization and does not affect tessellation.

Area

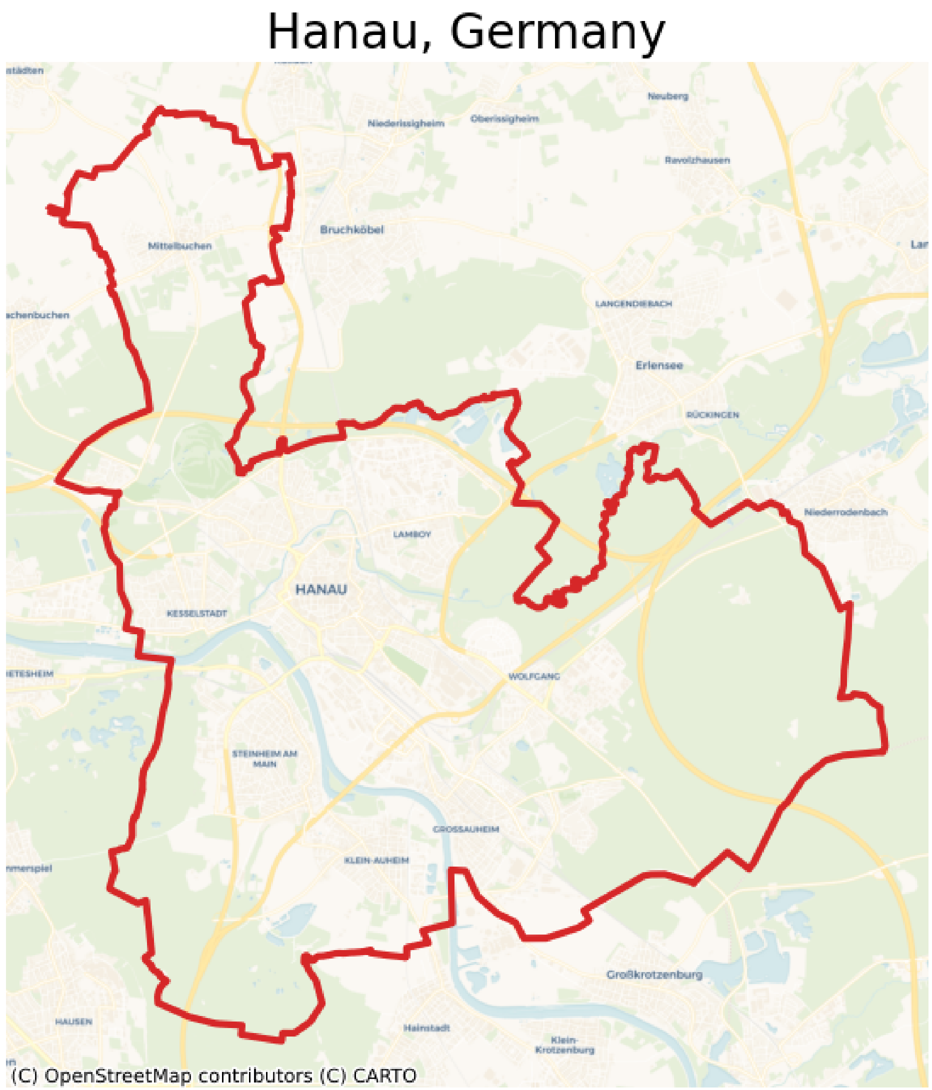
As a case study, we use **Hanau**, a city in Hesse, Germany.

[1]:
```
from tesspy import Tessellation
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100
plt.rcParams["figure.figsize"] = (8, 8)
import contextily as ctx
from time import sleep
```

[2]:
```
import warnings

warnings.simplefilter("ignore")
```

[3]:
```
# Create a tessellation object
hanau = Tessellation("Hanau, Germany")

# get polygon of the investigated area
hanau_polygon = hanau.get_polygon()
```

[4]:
```
# visualization of area
ax = hanau_polygon.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="tab:red", lw=3)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Hanau, Germany", fontsize=20)
plt.show()
```



nbsphinx-code-borderwhite

POI Categories

We can specify the desired POI categories by creating a list of them and passing it to the `poi_categories` keyword. We can investigate the possible options by:

[5]:
```
hanau.osm_primary_features()
```

[5]:
```
['aerialway',
 'aeroway',
 'amenity',
 'barrier',
 'boundary',
 'building',
 'craft',
 'emergency',
 'geological',
```

(continues on next page)

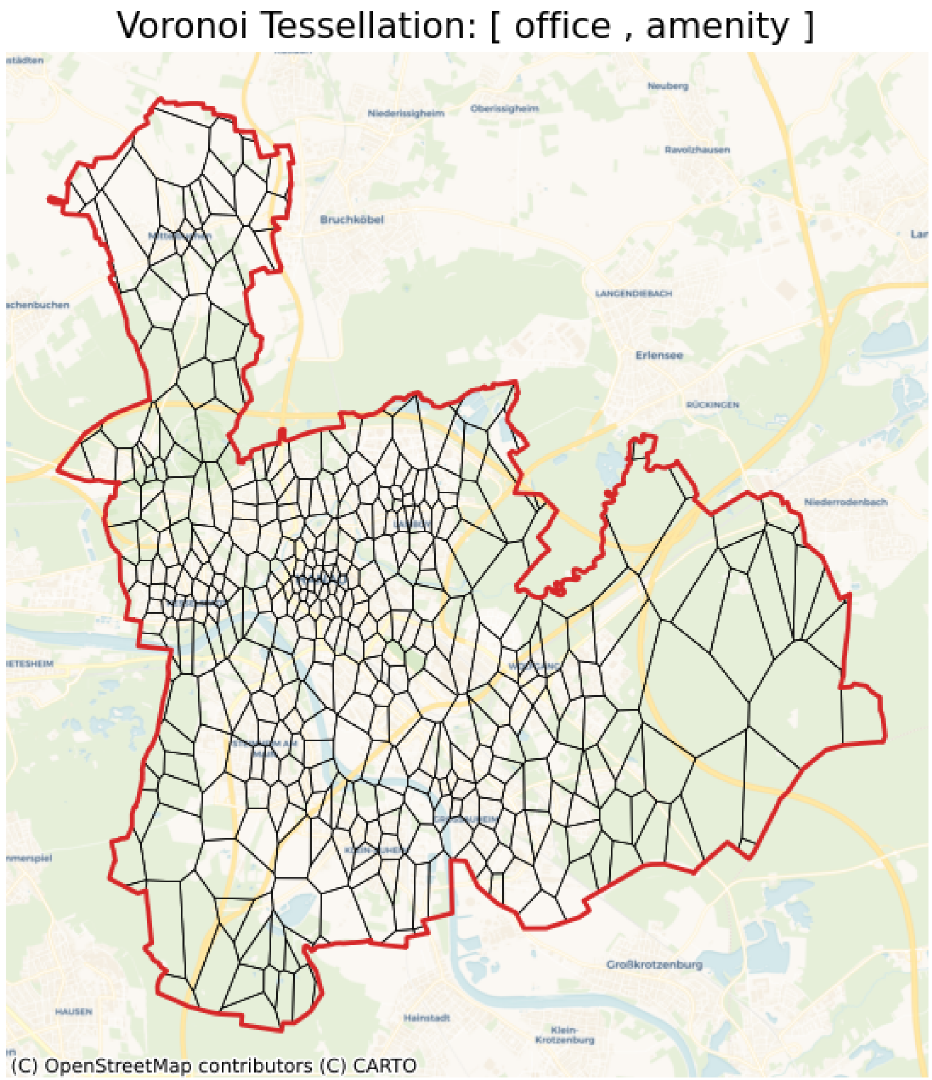
(continued from previous page)

```
'healthcare',
'highway',
'historic',
'landuse',
'leisure',
'man_made',
'military',
'natural',
'office',
'place',
'power',
'public_transport',
'railway',
'route',
'shop',
'sport',
'telecom',
'tourism',
'water',
'waterway']
```

For example, we can select `office`, and `amenity`.

```
[6]: hanau_vor = hanau.voronoi(n_polygons=500, poi_categories=["office", "amenity"])
```

```
[7]: ax = hanau_vor.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
hanau_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Voronoi Tessellation: [ office , amenity ]", fontsize=15)
plt.show()
```

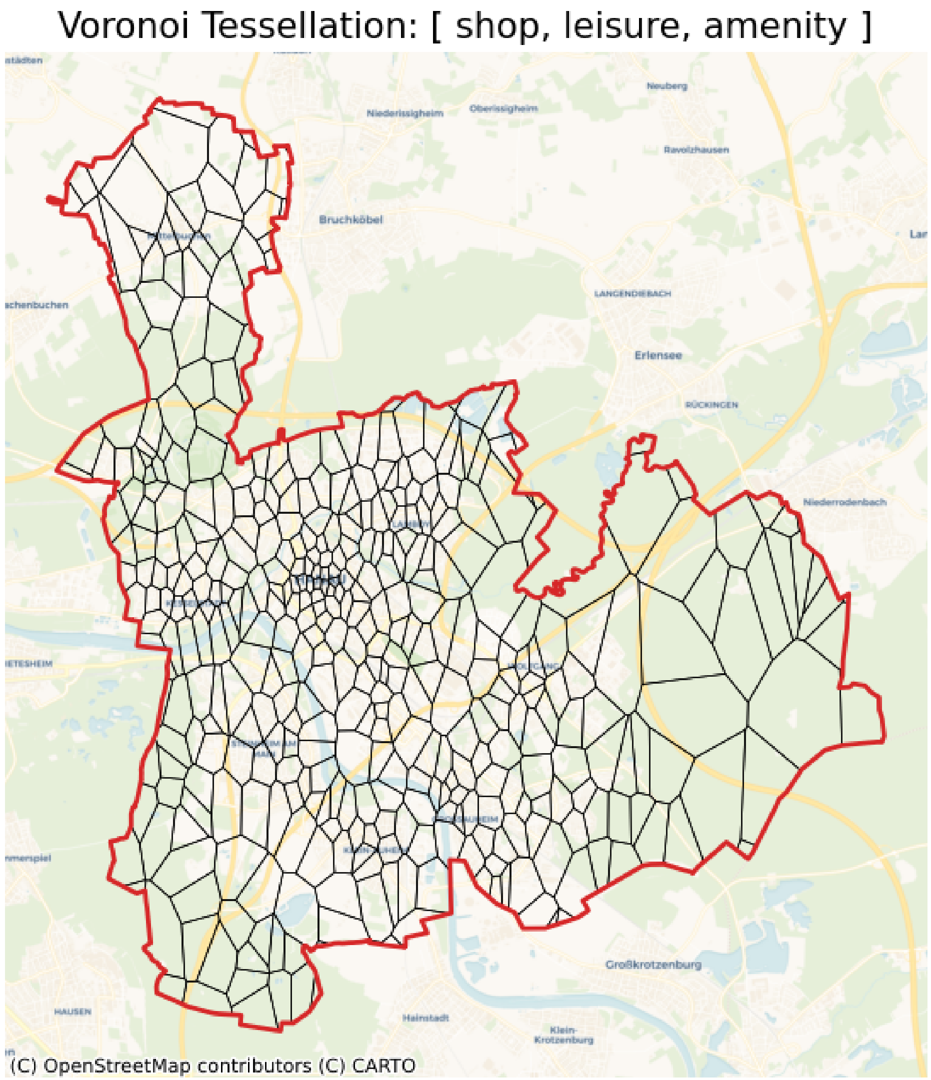


nbsphinx-code-borderwhite

We can build other Voronoi diagrams with different POI categories. For example, with `shop`, `leisure`, and `amenity`. Please note, the `amenity` POI are already retrieved from the last request and saved in the object. This request retrieves only `shop` and `leisure` data from the OSM.

[8]:
```
hanau_vor = hanau.voronoi(n_polygons=500, poi_categories=["shop", "leisure", "amenity"])
```

[9]:
```
ax = hanau_vor.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
hanau_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Voronoi Tessellation: [ shop, leisure, amenity ]", fontsize=15)
plt.show()
```



nbsphinx-code-borderwhite

If we set POI categories as `all`, all the POI are retrieved from the OSM. It is recommended to do this initially so that all POI data are retrieved and saved in the object. After that, we can play around with poi_categories, the number of polygons, and the clustering algorithm without sending a request to OSM. This makes the whole process faster as it avoids multiple requests. This way, we could also prevent consequent requests on OSM that may result in a Runtime error for overloading the server.

Getting all the POI data can take a few minutes. Therefore, we set the verbose to `True` to track the process.

[10]:
```
sleep(180)
hanau_vor = hanau.voronoi(n_polygons=500, poi_categories="all", verbose=True)
```

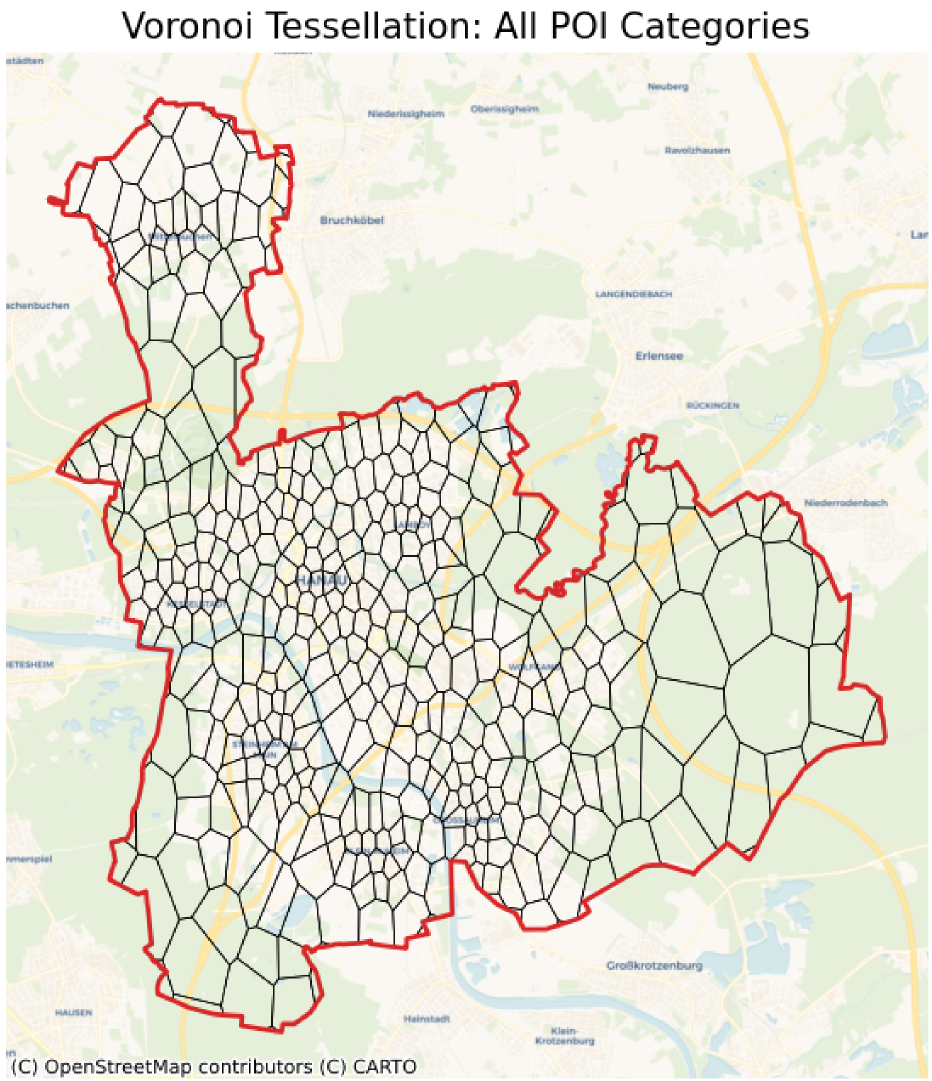
```
Getting data from OSM...
Creating POI DataFrame...
Cleaning POI DataFrame...
K-Means Clustering...
Creating Voronoi polygons...
```

[11]:
```
ax = hanau_vor.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
```

(continues on next page)

(continued from previous page)

```
hanau_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Voronoi Tessellation: All POI Categories", fontsize=15)
plt.show()
```



nbsphinx-code-borderwhite

All the POI are now collected and saved in the Tessellation object. We can call the POI data by:

[12]:
```
hanau_poi_data = hanau.get_poi_data()
hanau_poi_data.head()
```

[12]:
```
  type                                           geometry  \
0  way  [{'lat': 50.1315905, 'lon': 8.9205818}, {'lat'...
1  way  [{'lat': 50.1173541, 'lon': 8.8956382}, {'lat'...
2  way  [{'lat': 50.127607, 'lon': 8.8860676}, {'lat':...
3  way  [{'lat': 50.1498123, 'lon': 8.8867456}, {'lat'...
```

(continues on next page)

(continued from previous page)

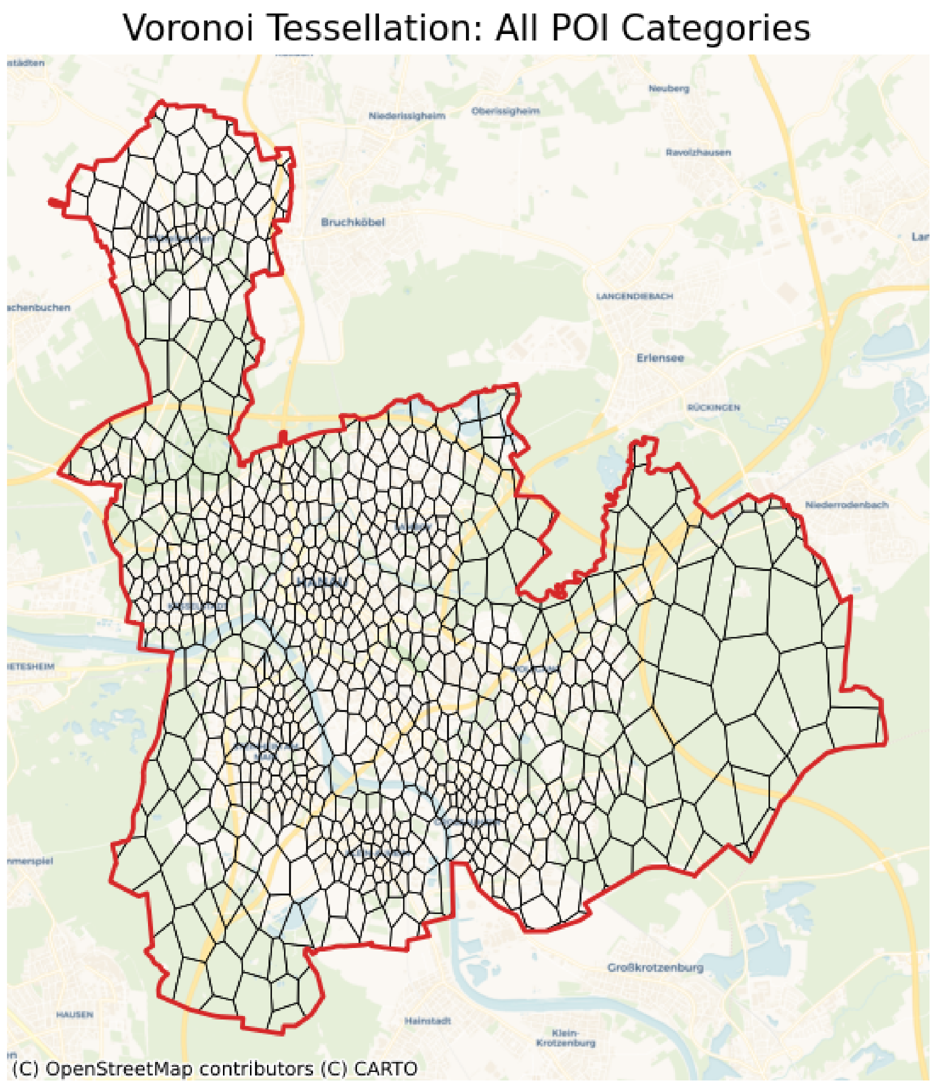
```
4  way  [{'lat': 50.1331451, 'lon': 8.8983281}, {'lat'...

                                                tags  center_latitude  \
0  {'access': 'yes', 'amenity': 'parking', 'fee':...        50.131392
1       {'amenity': 'parking', 'created_by': 'JOSM'}        50.117161
2  {'amenity': 'parking', 'parking': 'street_side...        50.127186
3  {'access': 'yes', 'amenity': 'parking', 'capac...        50.149523
4  {'access': 'yes', 'amenity': 'parking', 'creat...        50.133025

   center_longitude  amenity  office  leisure   shop  aerialway  ...  place  \
0          8.920731     True   False    False  False      False  ...  False
1          8.895813     True   False    False  False      False  ...  False
2          8.886050     True   False    False  False      False  ...  False
3          8.886864     True   False    False  False      False  ...  False
4          8.898218     True   False    False  False      False  ...  False

   power  public_transport  railway  route  sport  telecom  tourism  water  \
0  False             False    False  False  False    False    False  False
1  False             False    False  False  False    False    False  False
2  False             False    False  False  False    False    False  False
3  False             False    False  False  False    False    False  False
4  False             False    False  False  False    False    False  False

   waterway
0     False
1     False
2     False
3     False
4     False

[5 rows x 34 columns]
```

We can now test different Voronoi tessellations with different POI categories without having to get data again from the OSM. We create 1000 Voronoi polygons with all the POI data again. As can be seen by verbose, data collection is skipped.

[13]:
```python
hanau_vor = hanau.voronoi(n_polygons=1000, poi_categories="all", verbose=True)
```

```
K-Means Clustering...
Creating Voronoi polygons...
```

[14]:
```python
ax = hanau_vor.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
hanau_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Voronoi Tessellation: All POI Categories", fontsize=15)
plt.show()
```



nbsphinx-code-borderwhite

Number of Polygons

Voronoi polygons need a list of generators to be built. The number of polygons is the number of generators. Please note that the keyword `n_polygons` is an approximate number. In order to have more accurate results, always a bigger area (buffered area) than the investigated area is analyzed. The reason is to have reasonable polygons on the borders.

The final number of created Voronoi polygons is usually smaller than `n_polygons`. We can play around with the number to get the desired results.

Let's create 100, 500, and 1000 polygons using only `buildings`.

[15]:
```
hanau_vor_100_buildings = hanau.voronoi(n_polygons=100, poi_categories=["building"])
hanau_vor_500_buildings = hanau.voronoi(n_polygons=500, poi_categories=["building"])
hanau_vor_1000_buildings = hanau.voronoi(n_polygons=1000, poi_categories=["building"])
```

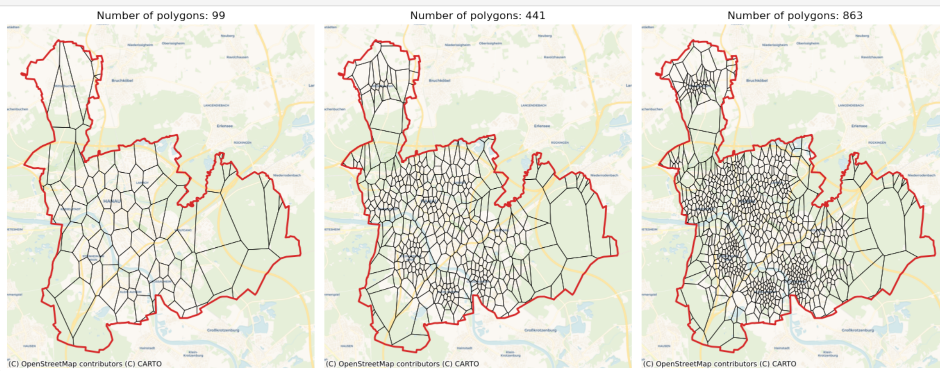
[16]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 6))
```

(continues on next page)

(continued from previous page)

```
hanau_vor_100_buildings.to_crs("EPSG:3857").plot(
    ax=axs[0], facecolor="none", edgecolor="k", lw=0.5
)
hanau_vor_500_buildings.to_crs("EPSG:3857").plot(
    ax=axs[1], facecolor="none", edgecolor="k", lw=0.5
)
hanau_vor_1000_buildings.to_crs("EPSG:3857").plot(
    ax=axs[2], facecolor="none", edgecolor="k", lw=0.5
)

axs[0].set_title(f"Number of polygons: {len(hanau_vor_100_buildings)}")
axs[1].set_title(f"Number of polygons: {len(hanau_vor_500_buildings)}")
axs[2].set_title(f"Number of polygons: {len(hanau_vor_1000_buildings)}")

for ax in axs.flatten():
    ax.set_axis_off()
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    hanau_polygon.to_crs("EPSG:3857").plot(
        ax=ax, facecolor="none", edgecolor="tab:red", lw=2
    )

plt.tight_layout()
```

nbsphinx-code-borderwhite



Clustering Algorithms

The Voronoi generators can be directly set as the POI coordinates. This can be done by setting the `cluster_algo` to `None`. For example, we can create Voronoi polygons with `leisure` as generators.

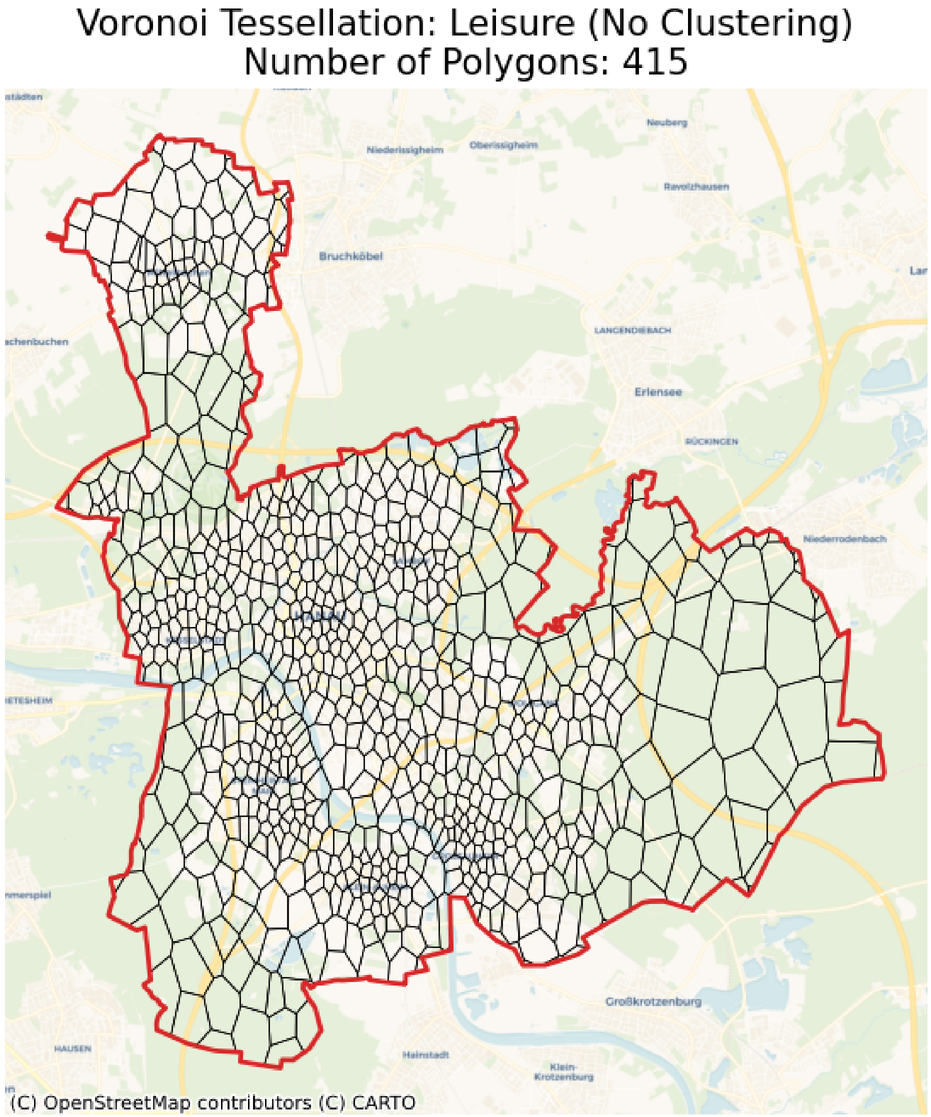
[17]:
```
hanau_vor_leisure = hanau.voronoi(cluster_algo=None, poi_categories=["leisure"])
```

[18]:
```
ax = hanau_vor.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
hanau_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title(
    f"Voronoi Tessellation: Leisure (No Clustering)\nNumber of Polygons: {len(hanau_vor_
↪leisure)}",
```

(continues on next page)

(continued from previous page)

```
    fontsize=15,
)
plt.show()
```



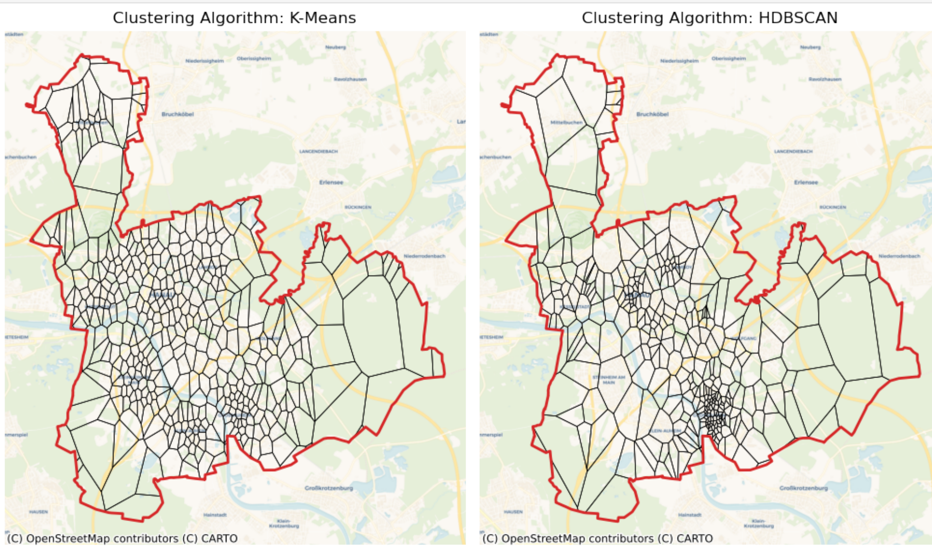
nbsphinx-code-borderwhite

Since there are usually many POI, we cluster the POI and use the cluster centroids as generators. There are currently two clustering algorithms implemented. The first and robust one is `K-Means`. The second one is `hdbscan` which is still under test. For `K-Means` clustering, we set the `n_polygons` (`min_cluster_size` is ignored). For `hdbscan` we set the `min_cluster_size` (`n_polygons` is ignored).

[19]:
```
hanau_vor_kmeans = hanau.voronoi(
    cluster_algo="k-means", n_polygons=500, poi_categories=["building"]
)
hanau_vor_hdbscan = hanau.voronoi(
    cluster_algo="hdbscan", min_cluster_size=7, poi_categories=["building"]
)
```

[20]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 6))

hanau_vor_kmeans.to_crs("EPSG:3857").plot(
    ax=axs[0], facecolor="none", edgecolor="k", lw=0.5
)
hanau_vor_hdbscan.to_crs("EPSG:3857").plot(
    ax=axs[1], facecolor="none", edgecolor="k", lw=0.5
)

axs[0].set_title(f"Clustering Algorithm: K-Means")
axs[1].set_title(f"Clustering Algorithm: HDBSCAN")

for ax in axs.flatten():
    ax.set_axis_off()
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
    hanau_polygon.to_crs("EPSG:3857").plot(
        ax=ax, facecolor="none", edgecolor="tab:red", lw=2
    )

plt.tight_layout()
```

nbsphinx-code-borderwhite



City Blocks

This notebook shows how we can properly build city blocks for urban areas. City blocks are the areas surrounded by streets. We use the OpenStreetMap to get the road network of an area.

To run this notebook, in addition to `tesspy`, you need `contextily` for basemap visualization. This package is only used to enhance visualization and has no effect on tessellation.

Area

We use the city of **Liverpool** in the United Kingdom as a case study.

[1]:
```
from tesspy import Tessellation
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100
plt.rcParams["figure.figsize"] = (8, 8)
import contextily as ctx
from time import sleep
```

[2]:
```
# Create a tessellation object
liverpool = Tessellation("Liverpool, United Kingdom")

# get polygon of the investigated area
liverpool_polygon = liverpool.get_polygon()
```

[14]:
```
# visualization of area
ax = liverpool_polygon.to_crs("EPSG:3857").plot(
    facecolor="none", edgecolor="tab:red", lw=3
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title("Liverpool, United Kingdom", fontsize=15)
plt.show()
```

nbsphinx-code-borderwhite

Street types

There are different types of streets defined by the OSM. We have listed this from top-level to low-level. This list can be seen by using the static method `.osm_highway_types()`. By using the keyword `detail_deg`, we can specify how detailed we want to get road network data from the OSM starting from the top category of the list. `detail_deg` can be between 1 and 19.

[12]:
```
liverpool.osm_highway_types()
```

[12]:
```
['motorway',
 'trunk',
```

(continues on next page)

(continued from previous page)

```
 'primary',
 'secondary',
 'tertiary',
 'residential',
 'unclassified',
 'motorway_link',
 'trunk_link',
 'primary_link',
 'secondary_link',
 'living_street',
 'pedestrian',
 'track',
 'bus_guideway',
 'footway',
 'path',
 'service',
 'cycleway']
```

For example, for a rough tessellation we can use only the top 5 road (highway) types which are: `motorway`, `trunk`, `primary`, `secondary`, and `tertiary`.

[20]:
```
liverpool_cb_deg5 = liverpool.city_blocks(n_polygons=None, detail_deg=5)
```

[21]:
```
ax = liverpool_cb_deg5.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.5)
liverpool_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title(
    "Liverpool, United Kingdom\n City Blocks: Top 5 Highway Types", fontsize=15
)
plt.show()
```

nbsphinx-code-borderwhite

Creating city blocks

If we use most of the road network data, we end up with many small polygons. Some of which are not even meaningful, e.g., traffic islands. Therefore, we cluster the small polygons (using hierarchical clustering) and merge them into larger polygons. This way, we can generate more realistic city blocks. In addition, we can generate an arbitrary number of city blocks by tuning the number of clusters by `n_polygons`.

If we only use the top-level road types, we may not need to cluster the polygons. In this case, as seen above, we can pass `None` to `n_polygons`.

In the first example, we can use all the road data to generate city blocks without clustering or merging them. First, let's see how many polygons it creates and what it looks like:

[22]:
```
liverpool_cb_all = liverpool.city_blocks()
```

[25]:
```
ax = liverpool_cb_all.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="k", lw=0.1)
liverpool_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title(
    "Liverpool, United Kingdom\n City Blocks: Entire Road Network\nAll Polygons",
    fontsize=15,
)
plt.show()
```

nbsphinx-code-borderwhite

[27]:
```
print("Number of polygons: ", len(liverpool_cb_all))
```

```
Number of polygons:  11868
```

This resulted in more than 10,000 polygons. So it makes sense to cluster and merge these polygons. With repeat the same process. This time we set `n_polygons` to be 500:

[29]:
```
liverpool_cb_all_500 = liverpool.city_blocks(n_polygons=500)
```

[35]:
```
ax = liverpool_cb_all_500.to_crs("EPSG:3857").plot(
```

(continues on next page)

(continued from previous page)

```
    facecolor="none", edgecolor="k", lw=0.2
)
liverpool_polygon.to_crs("EPSG:3857").plot(
    ax=ax, facecolor="none", edgecolor="tab:red", lw=2
)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Voyager)
ax.set_axis_off()
ax.set_title(
    f"Liverpool, United Kingdom"
    "City Blocks: Entire Road Network"
    f"nClustered: {len(liverpool_cb_all_500)} Polygons",
    fontsize=15,
)
plt.show()
```

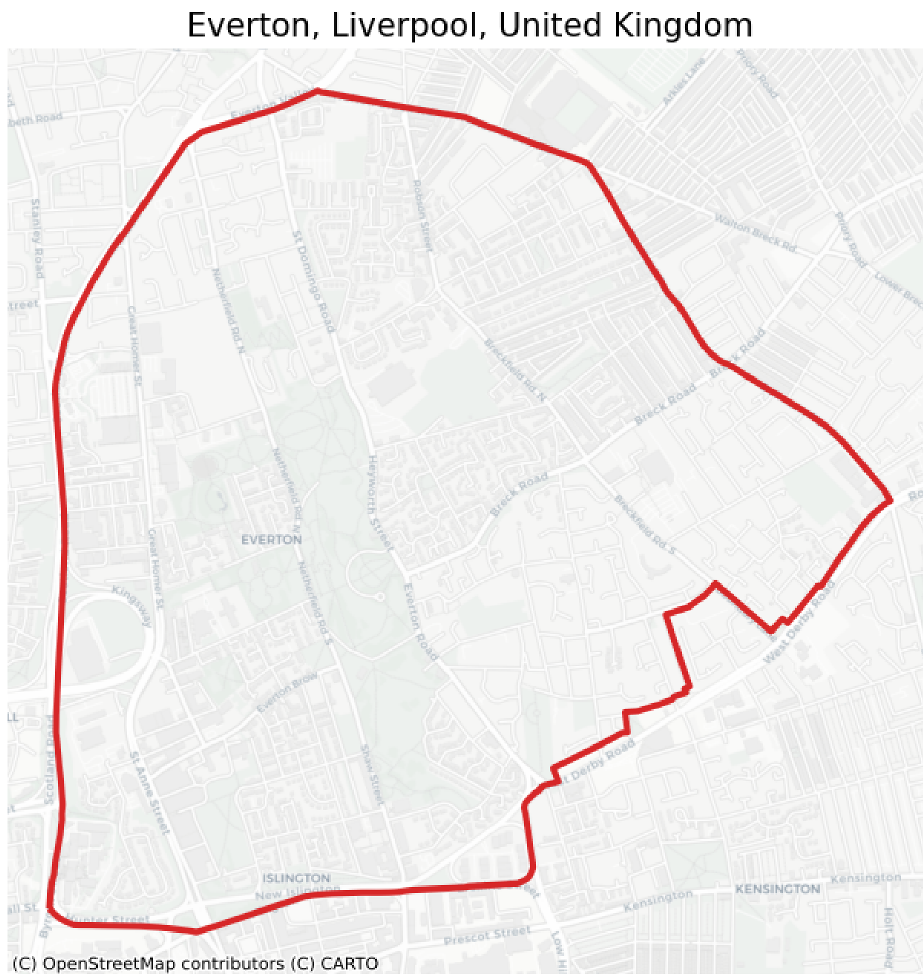
nbsphinx-code-borderwhite

These polygons look much better. We can use these in further spatial analyses.

To see the difference more clearly, we investigate a small cross-section of Liverpool:

[54]:
```
# Create a tessellation object
ev = Tessellation("Everton, Liverpool, UK")

# get polygon of the investigated area
ev_polygon = ev.get_polygon()
```

[62]:
```
# visualization of area
ax = ev_polygon.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="tab:red", lw=3)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.set_title("Everton, Liverpool, United Kingdom", fontsize=15)
plt.show()
```



nbsphinx-code-borderwhite

[60]:
```
ev_cb_all = ev.city_blocks()
ev_cb_all_100 = ev.city_blocks(n_polygons=100)
```

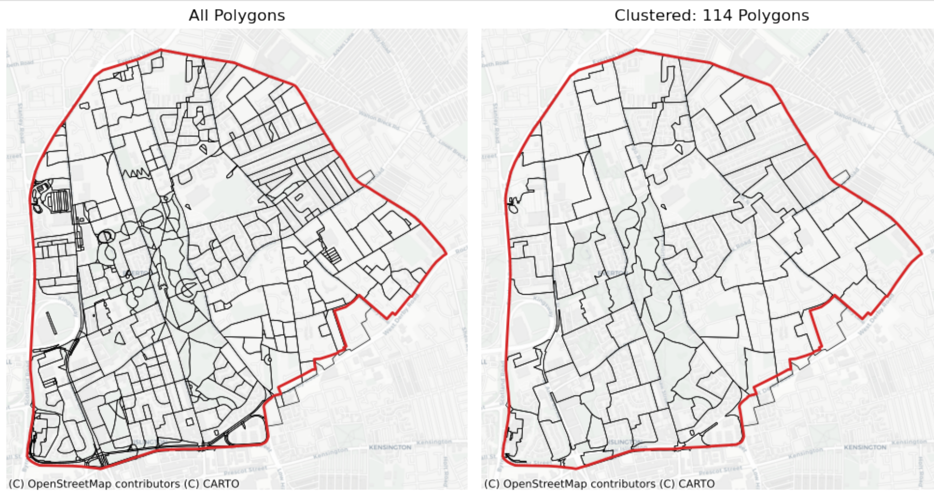
[61]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 6))

ev_cb_all.to_crs("EPSG:3857").plot(ax=axs[0], facecolor="none", edgecolor="k", lw=0.5)
ev_cb_all_100.to_crs("EPSG:3857").plot(
    ax=axs[1], facecolor="none", edgecolor="k", lw=0.5
)

axs[0].set_title(f"All Polygons")
axs[1].set_title(f"Clustered: {len(ev_cb_all_100)} Polygons")

for ax in axs.flatten():
```

(continues on next page)

(continued from previous page)

```
    ax.set_axis_off()
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
    ev_polygon.to_crs("EPSG:3857").plot(
        ax=ax, facecolor="none", edgecolor="tab:red", lw=2
    )

plt.tight_layout()
```

nbsphinx-code-borderwhite



Analyzing Urban Areas

This notebook shows how we can perform POI-based analysis for urban areas. Discretization of urban areas allows spatial analysis. For example, a very typical use case is to create heatmaps. You can read any spatial data that you have (e.g., real estate prices, air pollution, etc.) and start to analyze them based on the created tiles. Here we use Open Street Map to show several example use cases.

We create heatmaps of amenities based on different tessellation methods. Moreover, investigate the autocorrelation between polygons by calculating Moran's I. In addition, different types of amenities can be extracted from POI data. For example, we visualize cafes and restaurants.

To run this notebook, in addition to `tesspy`, you need `contextily` for basemap visualization and `esda`, `statsmodels`, and `libpysal` for statistical and spatial analysis.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100
plt.rcParams["figure.figsize"] = (8, 8)

import geopandas as gpd
from shapely.geometry import Point, LineString, Polygon, MultiPoint
import contextily as ctx

import esda
import libpysal as lp
```

(continues on next page)

(continued from previous page)

```
import statsmodels.api as sm
from scipy.stats import norm

# Shapely 1.8.1 makes pandas to produce many warnings; this is to get rid of these␣
↪warnings
import warnings

warnings.simplefilter("ignore")

from time import sleep
```

[2]:
```
from tesspy import Tessellation
```

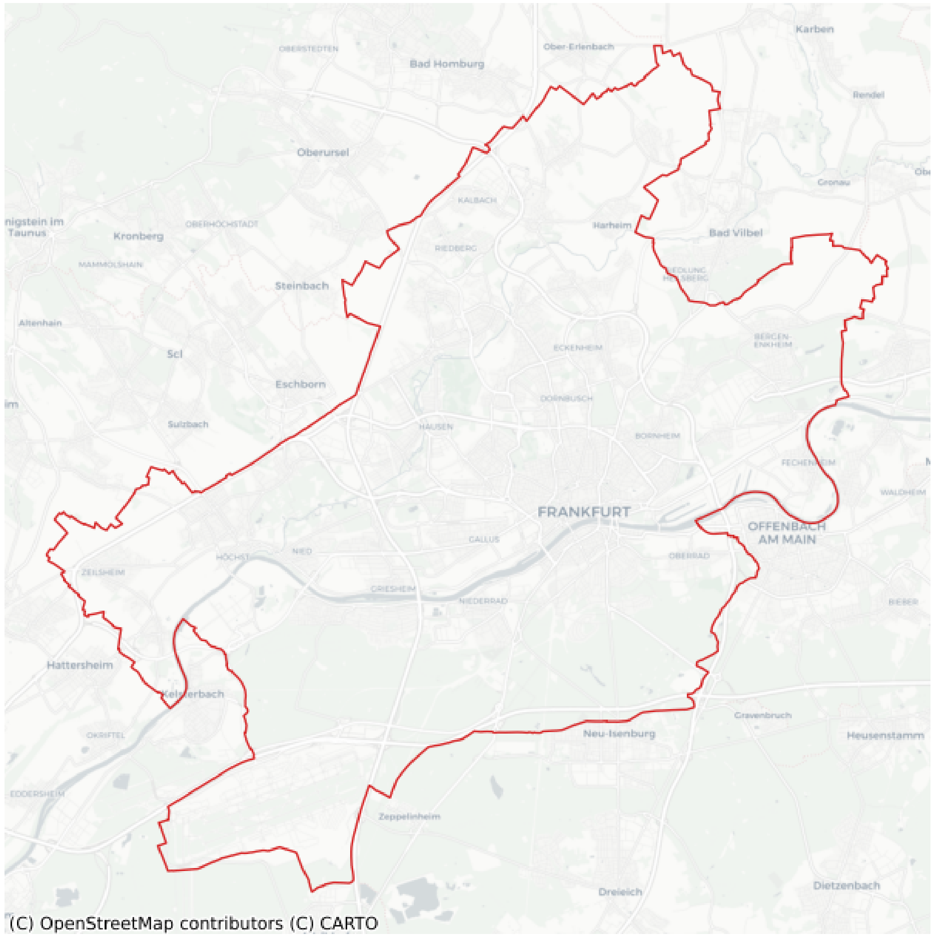
Area

We use **Frankfurt am Main** in Germany as a case study. First, we get the city boundary. Then we generate different tessellations.

[3]:
```
ffm = Tessellation("Frankfurt am Main")
ffm_polygon = ffm.get_polygon()
```

[4]:
```
# visualization of area
ax = ffm_polygon.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="tab:red", lw=1)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.set_title("Frankfurt am Main", fontsize=10)
plt.show()
```

Frankfurt am Main



nbsphinx-code-borderwhite

Tessellation

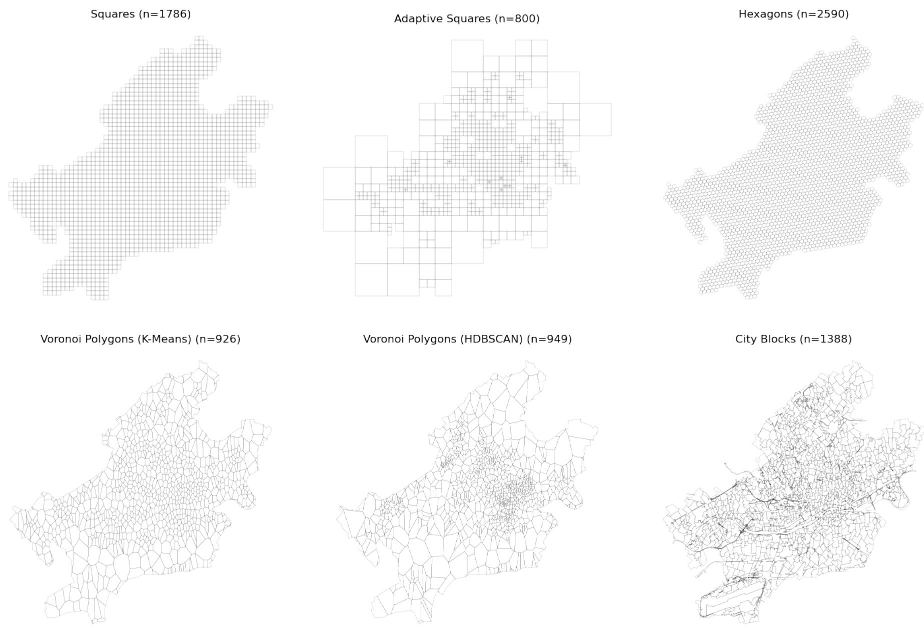
[5]:
```
# squares
ffm_sqr_16 = ffm.squares(16)
# hexagons
ffm_hex_9 = ffm.hexagons(9)
# adaptive squares
ffm_asq = ffm.adaptive_squares(
    start_resolution=13,
    threshold=500,
    timeout=60,
    poi_categories=["shop", "building", "amenity", "office", "public_transport"],
)
# voronoi Diagrams with K-Means
ffm_voronoi_kmeans = ffm.voronoi(
    poi_categories=["shop", "building", "amenity", "office", "public_transport"],
    n_polygons=1000,
)
# voronoi Diagrams with hdbscan
```

(continues on next page)

(continued from previous page)

```
ffm_voronoi_hdbscan = ffm.voronoi(
    cluster_algo="hdbscan",
    min_cluster_size=10,
    poi_categories=["shop", "building", "amenity", "office", "public_transport"],
)
sleep(120)
# city blocks
ffm_cb = ffm.city_blocks(n_polygons=1000, detail_deg=None)
```

[6]:
```
ffm_dfs = [
    ffm_sqr_16,
    ffm_asq,
    ffm_hex_9,
    ffm_voronoi_kmeans,
    ffm_voronoi_hdbscan,
    ffm_cb,
]
titles = [
    "Squares",
    "Adaptive Squares",
    "Hexagons",
    "Voronoi Polygons (K-Means)",
    "Voronoi Polygons (HDBSCAN)",
    "City Blocks",
]

fig, axs = plt.subplots(2, 3, figsize=(15, 10))
for ax, df, title in zip(axs.flatten(), ffm_dfs, titles):
    ax.set_axis_off()
    df.plot(ax=ax, facecolor="none", edgecolor="k", lw=0.1)
    ax.set_title(f"\n{title} (n={len(df)})")

plt.tight_layout()
plt.show()
```



nbsphinx-code-borderwhite

We can calculate the areas of polygons (tiles) and see how they differ between different tessellation methods. In square and hexagon methods, the area of polygons is (almost) constant. By Voronoi and city blocks, it varies. We can take a look at their histograms to investigate them.

We convert the CRS to `EPSG:5243` (for Frankfurt). Using this, the coordinates are in meters, and the calculated area is in square meters.

[7]:
```python
# Calculate Areas

for df in ffm_dfs:
    df["area"] = df.to_crs("EPSG:5243").area
```

Here are the polygon size for squares:

[8]:
```python
ffm_sqr_16["area"].describe()
```

[8]:
```
count      1786.000000
mean     153687.014738
std         306.670013
min      152971.129738
25%      153449.226805
50%      153700.251963
75%      153905.940218
max      154341.071288
Name: area, dtype: float64
```

Here are the polygon size hexagons:

[9]:
```python
ffm_hex_9["area"].describe()
```

[9]:
```
count     2590.000000
mean     95893.716176
std         80.618428
min      95718.110703
```

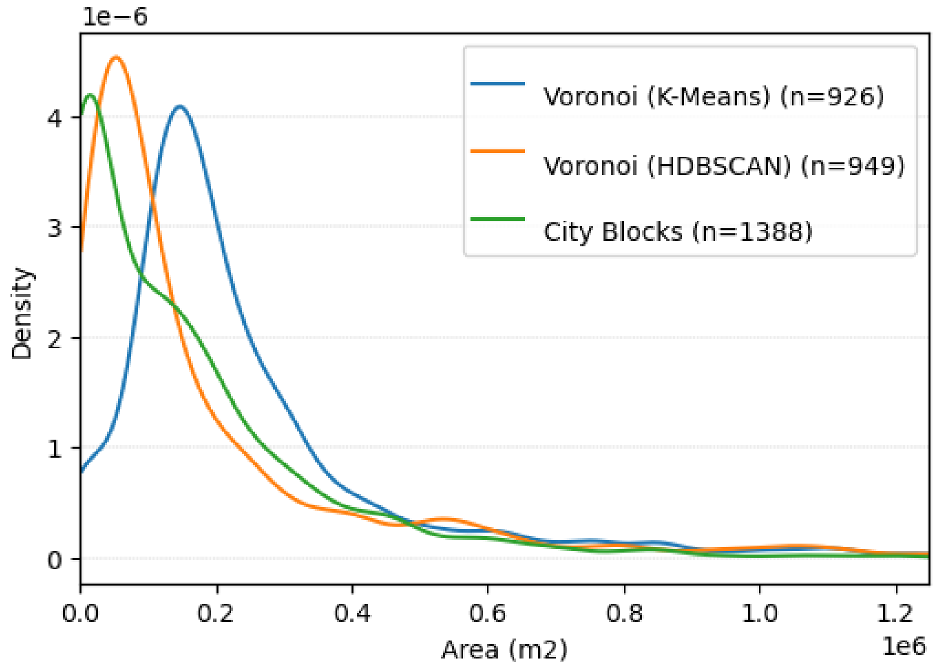
(continues on next page)

(continued from previous page)

```
25%      95830.827327
50%      95896.443375
75%      95960.443788
max      96061.804198
Name: area, dtype: float64
```

For the other Voronoi and city blocks, we create a histogram.

[10]:
```
def trunc_dens(x):
    kde = sm.nonparametric.KDEUnivariate(x)
    kde.fit()
    h = kde.bw
    w = 1 / (1 - norm.cdf(0, loc=x, scale=h))
    d = sm.nonparametric.KDEUnivariate(x)
    d = d.fit(bw=h, weights=w / len(x), fft=False)
    d_support = d.support
    d_dens = d.density
    d_dens[d_support < 0] = 0
    return d_support, d_dens


fig, ax = plt.subplots(1, 1, figsize=(6, 4))

for df, title in zip(
    [ffm_voronoi_kmeans, ffm_voronoi_hdbscan, ffm_cb],
    ["Voronoi (K-Means)", "Voronoi (HDBSCAN)", "City Blocks"],
):

    _x, _y = trunc_dens(df["area"])
    ax.plot(_x[_x > 0], _y[-len(_x[_x > 0]) :], label=f"\n{title} (n={df.shape[0]})")
    ax.set_xlabel("Area (m2)")
    ax.set_ylabel("Density")
    ax.grid(axis="y", lw=0.3, ls="--", zorder=0)

plt.legend()
plt.xlim([0, 1.25e6])
plt.show()
```



nbsphinx-code-borderwhite

POI data: Amenities

Let's continue by taking a look at the retrieved POI data from OSM:

```
[11]: poi_df = ffm.get_poi_data()
      poi_df.head()
```

```
[11]:   type                                           geometry  \
0  way  [{'lat': 50.1074607, 'lon': 8.734269}, {'lat':...
1  way  [{'lat': 50.1670562, 'lon': 8.6760002}, {'lat'...
2  way  [{'lat': 50.1488198, 'lon': 8.6928804}, {'lat'...
3  way  [{'lat': 50.1446936, 'lon': 8.6511394}, {'lat'...
4  way  [{'lat': 50.0518233, 'lon': 8.5654572}, {'lat'...

                                                tags  center_latitude  \
0  {'access': 'customers', 'amenity': 'parking', ...        50.107625
1  {'amenity': 'school', 'contact:email': 'schull...        50.167461
2  {'amenity': 'prison', 'barrier': 'fence', 'nam...        50.149950
3  {'addr:city': 'Frankfurt am Main', 'addr:house...        50.145195
4  {'addr:city': 'Frankfurt am Main', 'addr:postc...        50.052712

   center_longitude  amenity  building  office  public_transport   shop
0          8.734458     True     False   False             False  False
1          8.675039     True     False   False             False  False
2          8.695300     True     False   False             False  False
3          8.650782     True     False   False             False  False
4          8.568809    False      True   False             False  False
```

This dataframe contains geometry and other tags of the selected POI categories. We can take a look at the total number of each POI category in Frankfurt:

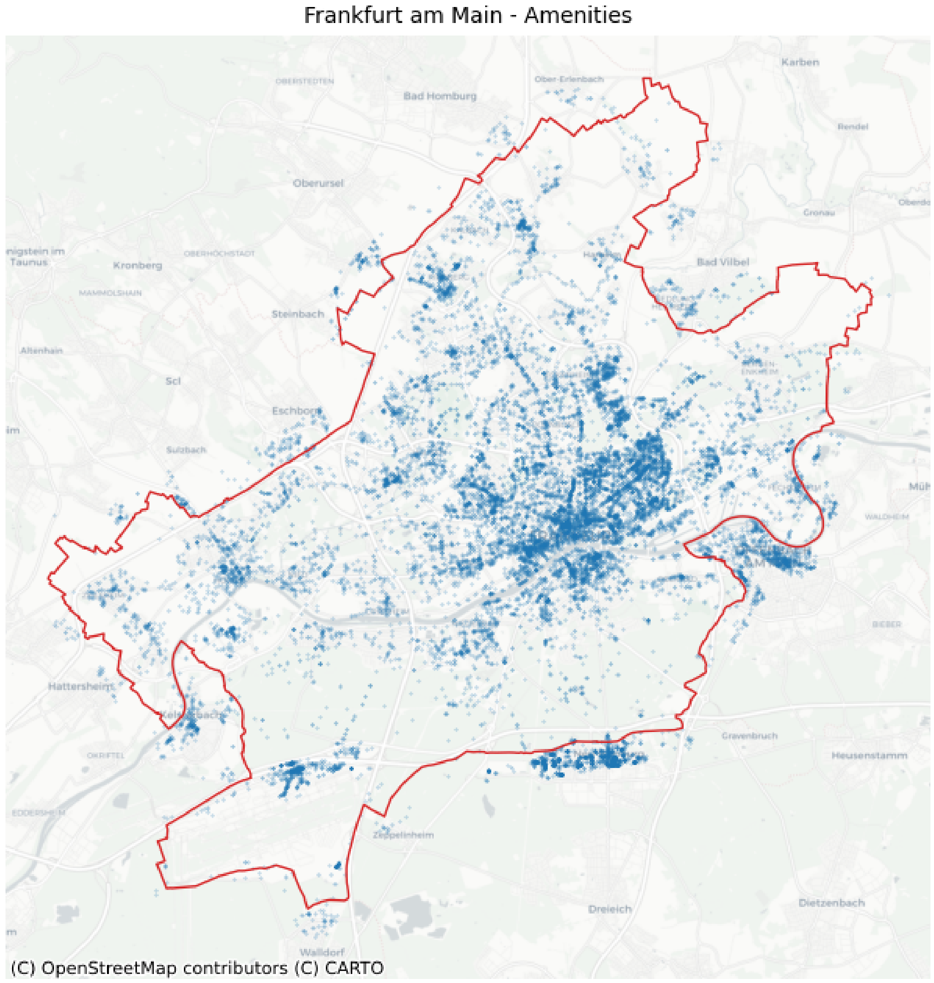
[12]:
```
poi_df.iloc[:, -5:].sum()
```

[12]:
```
amenity              19799
building            144471
office                 689
public_transport      5159
shop                  4357
dtype: int64
```

Let's visualize amenities on the map:

[13]:
```
poi_geodata = gpd.GeoDataFrame(
    data=poi_df.drop(columns=["geometry"]),
    geometry=poi_df[["center_longitude", "center_latitude"]]
    .apply(Point, axis=1)
    .values,
    crs="EPSG:4326",
)

amenity_data = poi_geodata[poi_geodata["amenity"]].copy()
```

[14]:
```
# visualization of area
ax = amenity_data.to_crs("EPSG:3857").plot(color="tab:blue", markersize=0.1, alpha=0.5)
ffm_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="tab:red", lw=1)

ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.set_title("Frankfurt am Main - Amenities", fontsize=10)
plt.show()
```



nbsphinx-code-borderwhite

Using the `tags` in the data, we can analyze the amenities further. For example, the types of amenities can be extracted. Different heatmaps for different amenity types can be provided.

[15]:
```
amenity_data["amenity_type"] = amenity_data["tags"].apply(lambda x: x["amenity"])
```

[16]:
```
amenity_data["amenity_type"].value_counts().head(20)
```

[16]:
```
bicycle_parking     3430
bench               2225
parking             2223
restaurant          1392
parking_space       1272
waste_basket         847
recycling            797
vending_machine      696
kindergarten         580
cafe                 508
post_box             492
fast_food            473
parking_entrance     383
place_of_worship     318
```

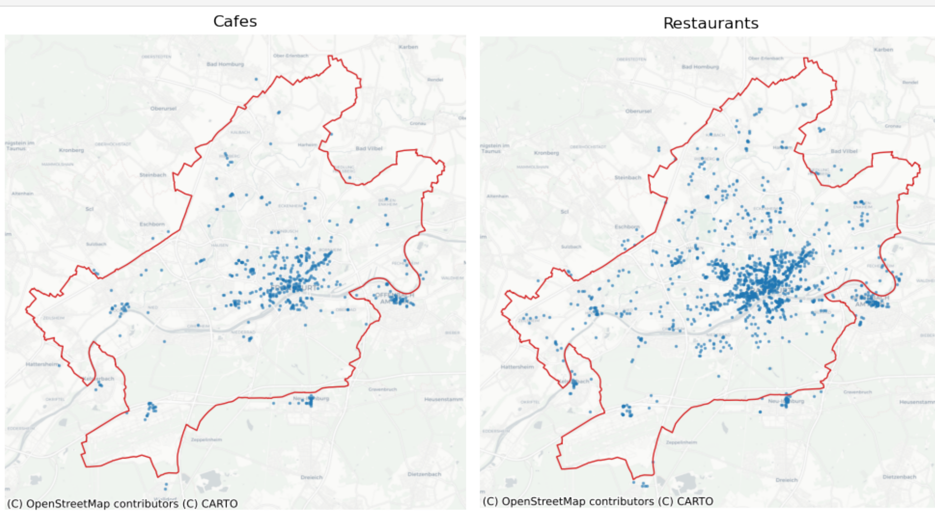
(continues on next page)

(continued from previous page)

```
bar                     246
school                  242
shelter                 214
pub                     201
telephone               189
pharmacy                177
Name: amenity_type, dtype: int64
```

[17]:
```
fig, axs = plt.subplots(1, 2, figsize=(10, 7))
amenity_data[amenity_data["amenity_type"] == "cafe"].to_crs("EPSG:3857").plot(
    ax=axs[0], color="tab:blue", markersize=2, alpha=0.5
)
amenity_data[amenity_data["amenity_type"] == "restaurant"].to_crs("EPSG:3857").plot(
    ax=axs[1], color="tab:blue", markersize=2, alpha=0.5
)

axs[0].set_title("Cafes")
axs[1].set_title("Restaurants")

for ax in axs.flatten():
    ax.set_axis_off()
    ffm_polygon.to_crs("EPSG:3857").plot(
        ax=ax, facecolor="none", edgecolor="tab:red", lw=1
    )
    ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)

plt.tight_layout()
plt.show()
```

nbsphinx-code-borderwhite



By looking at the maps, there seems to be a spatial correlation between cafes and restaurants. (This can be investigated using the tiles.)

Heatmaps for Amenities

We can use `amenity` to generate heatmaps based on the created polygons (tiles). First, we count `amenity` in each polygon for each tessellation method. Then we can visualize the resulted counts.

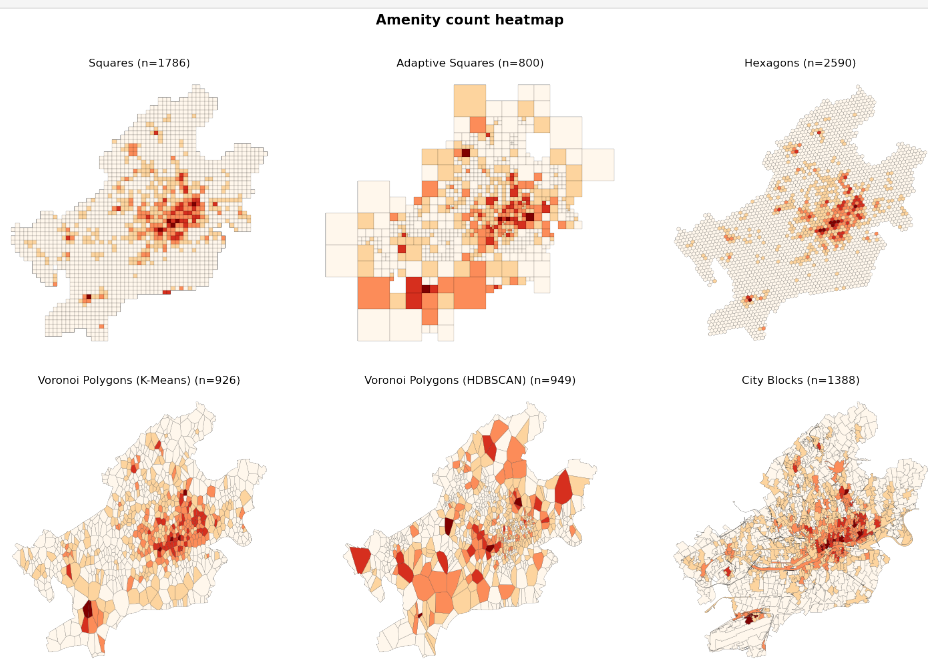
[18]:
```
# adding an ID to polygons
for df in ffm_dfs:
    df.reset_index(inplace=True)
    df.rename(columns={"index": "tile_id"}, inplace=True)

# joining the amenities with the polygons
amenity_unique_qk = gpd.sjoin(
    ffm_sqr_16, amenity_data, how="left", predicate="contains"
)
amenity_unique_adaptive_qk = gpd.sjoin(
    ffm_asq, amenity_data, how="left", predicate="contains"
)
amenity_unique_hexagon = gpd.sjoin(
    ffm_hex_9, amenity_data, how="left", predicate="contains"
)
amenity_unique_voronoi_kmeans = gpd.sjoin(
    ffm_voronoi_kmeans, amenity_data, how="left", predicate="contains"
)
amenity_unique_voronoi_hdbscan = gpd.sjoin(
    ffm_voronoi_hdbscan, amenity_data, how="left", predicate="contains"
)
amenity_unique_cityblocks = gpd.sjoin(
    ffm_cb, amenity_data, how="left", predicate="contains"
)

# counting the number of amenities in each polygon
count_amenity_qk = ffm_sqr_16.merge(
    amenity_unique_qk.groupby(by="tile_id").count()["index_right"].reset_index()
)
count_amenity_adaptive_qk = ffm_asq.merge(
    amenity_unique_adaptive_qk.groupby(by="tile_id")
    .count()["index_right"]
    .reset_index()
)
count_amenity_hexagon = ffm_hex_9.merge(
    amenity_unique_hexagon.groupby(by="tile_id").count()["index_right"].reset_index()
)
count_amenity_voronoi_kmeans = ffm_voronoi_kmeans.merge(
    amenity_unique_voronoi_kmeans.groupby(by="tile_id")
    .count()["index_right"]
    .reset_index()
)
count_amenity_voronoi_hdbscan = ffm_voronoi_hdbscan.merge(
    amenity_unique_voronoi_hdbscan.groupby(by="tile_id")
    .count()["index_right"]
    .reset_index()
)
count_amenity_cityblocks = ffm_cb.merge(
```

(continues on next page)

(continued from previous page)

```
    amenity_unique_cityblocks.groupby(by="tile_id").count()["index_right"].reset_index()
)
```

[19]:
```
count_amenity_dfs = [
    count_amenity_qk,
    count_amenity_adaptive_qk,
    count_amenity_hexagon,
    count_amenity_voronoi_kmeans,
    count_amenity_voronoi_hdbscan,
    count_amenity_cityblocks,
]

fig, axs = plt.subplots(2, 3, figsize=(15, 10))
for ax, df, title in zip(axs.flatten(), count_amenity_dfs, titles):
    ax.set_axis_off()
    df.plot(
        ax=ax,
        column="index_right",
        lw=0.1,
        alpha=1,
        scheme="fisherjenks",
        legend=False,
        cmap="OrRd",
        edgecolor="k",
    )
    ax.set_title(f"\n{title} (n={df.shape[0]})")

plt.suptitle("Amenity count heatmap", fontweight="bold", y=1, fontsize=15)
plt.tight_layout()
plt.show()
```



nbsphinx-code-borderwhite

It can be seen that the most amenities are within the city center. There are many amenities near the Frankfurt airport (Southwest of Frankfurt) and also in the districts' local centers.

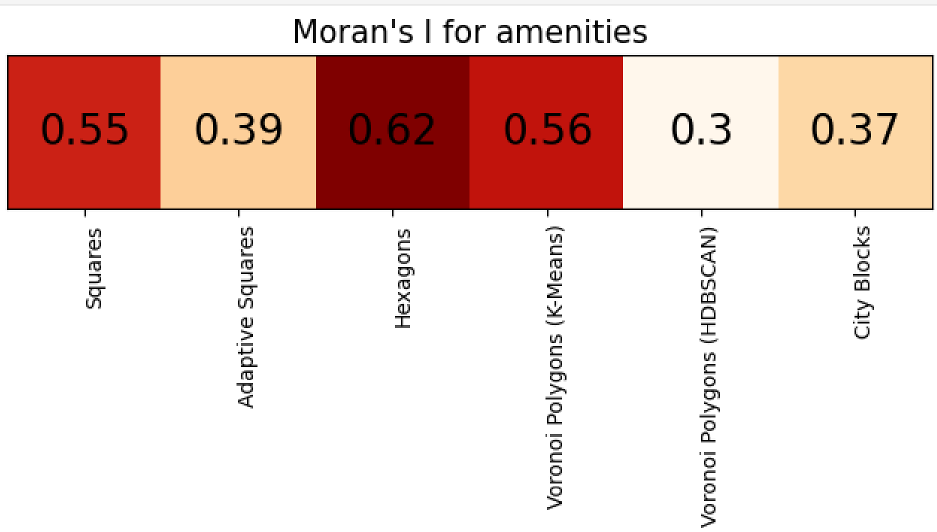
Spatial Autocorrelation: Moran's I

At first sight, there seems to be an autocorrelation between polygons for amenities. In order to investigate the spatial autocorrelation, we can calculate Moran's I index.

[20]:
```
mi_values = []
for count_df in count_amenity_dfs:
    wq = lp.weights.Queen.from_dataframe(count_df)
    wq.transform = "r"
    mi = esda.moran.Moran(count_df["index_right"], wq)
    mi_values.append(mi.I)

moran_df = pd.DataFrame(dict(zip(titles, [[i] for i in mi_values])))
```

```
('WARNING: ', 843, ' is an island (no neighbors)')
('WARNING: ', 899, ' is an island (no neighbors)')
('WARNING: ', 1135, ' is an island (no neighbors)')
```

[21]:
```
fig, ax = plt.subplots()
ax.imshow(moran_df, cmap="OrRd", interpolation="nearest")
plt.setp(ax.get_xticklabels(), rotation=90, ha="right", rotation_mode="anchor")
plt.tick_params(axis="y", left=False, labelleft=False)
ax.set_title("Moran's I for amenities", fontsize=15)
ax.set_xticks([])
ax.set_xticks(np.arange(len(titles)), labels=titles)
for j in range(len(titles)):
    text = ax.text(
        j, 0, round(mi_values[j], 2), ha="center", va="center", color="k", fontsize=20
    )
```

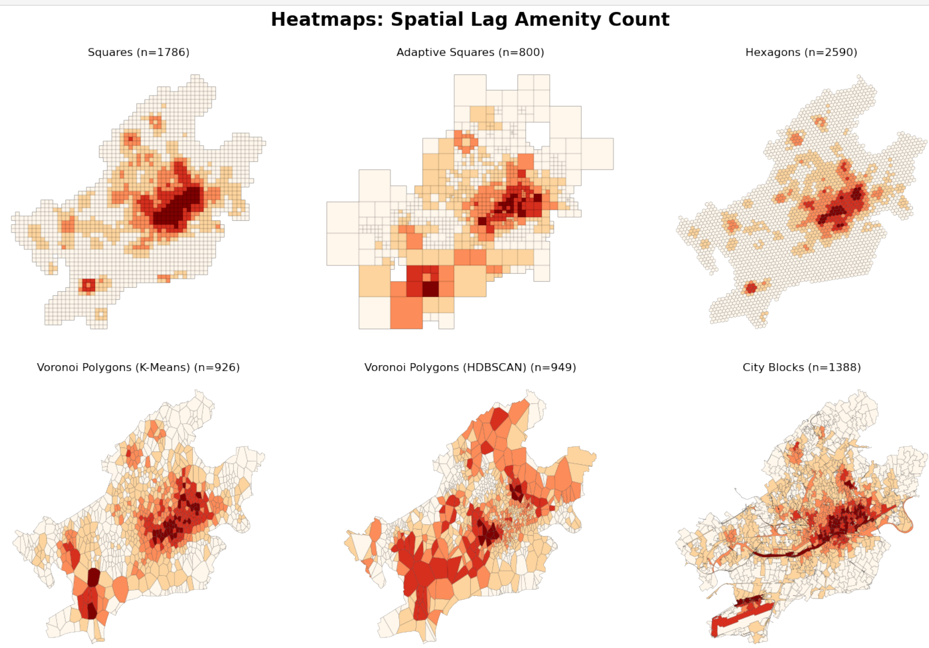


nbsphinx-code-borderwhite

Spatial Lag

Furthermore, we can calculate spatial lags and visualize them on the map.

[22]:
```
fig, axs = plt.subplots(2, 3, figsize=(15, 10))

for ax, count_df, title in zip(axs.flatten(), count_amenity_dfs, titles):
    wq = lp.weights.Queen.from_dataframe(count_df)
    y = count_df["index_right"]
    ylag = lp.weights.lag_spatial(wq, y)
    count_df.assign(ylag=ylag).plot(
        ax=ax,
        column="ylag",
        lw=0.1,
        alpha=1,
        scheme="fisherjenks",
        legend=False,
        cmap="OrRd",
        edgecolor="k",
    )
    ax.set_axis_off()
    ax.set_title(f"\n{title} (n={count_df.shape[0]})")

plt.suptitle("Heatmaps: Spatial Lag Amenity Count", fontweight="bold", fontsize=20)
plt.tight_layout()
plt.show()
```



nbsphinx-code-borderwhite

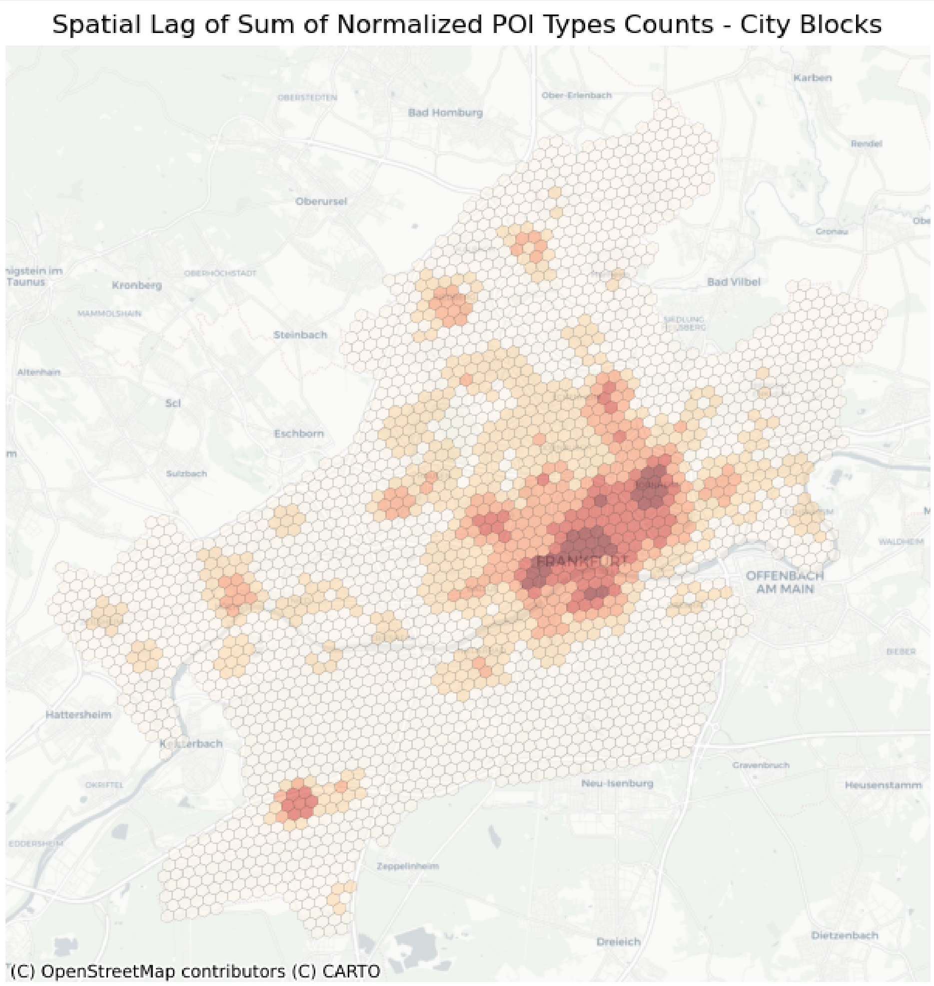
Let's take a closer look at hexagons:

[23]:
```
f, ax = plt.subplots()

wq = lp.weights.Queen.from_dataframe(count_amenity_hexagon)
```

(continues on next page)

(continued from previous page)

```
y = count_amenity_hexagon["index_right"]
ylag = lp.weights.lag_spatial(wq, y)
count_amenity_hexagon.to_crs("EPSG:3857").assign(ylag=ylag).plot(
    ax=ax,
    column="ylag",
    lw=0.1,
    alpha=0.5,
    scheme="fisherjenks",
    legend=False,
    cmap="OrRd",
    edgecolor="k",
)
ax.set_axis_off()
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
plt.title("Spatial Lag of Sum of Normalized POI Types Counts - City Blocks")
plt.show()
```



nbsphinx-code-borderwhite

Clustering Urban Areas

This notebook shows how tessellation can be used to generate clustering units in order to segment urban areas.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

plt.rcParams["figure.dpi"] = 100
plt.rcParams["figure.figsize"] = (8, 8)

import geopandas as gpd
from shapely.geometry import Point, LineString, Polygon, MultiPoint
import contextily as ctx
```

[2]:
```
from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
```

[38]:
```
# Shapely 1.8.1 makes pandas to produce many warnings; this is to get rid of these␣
↪warnings
import warnings

warnings.simplefilter("ignore")
```

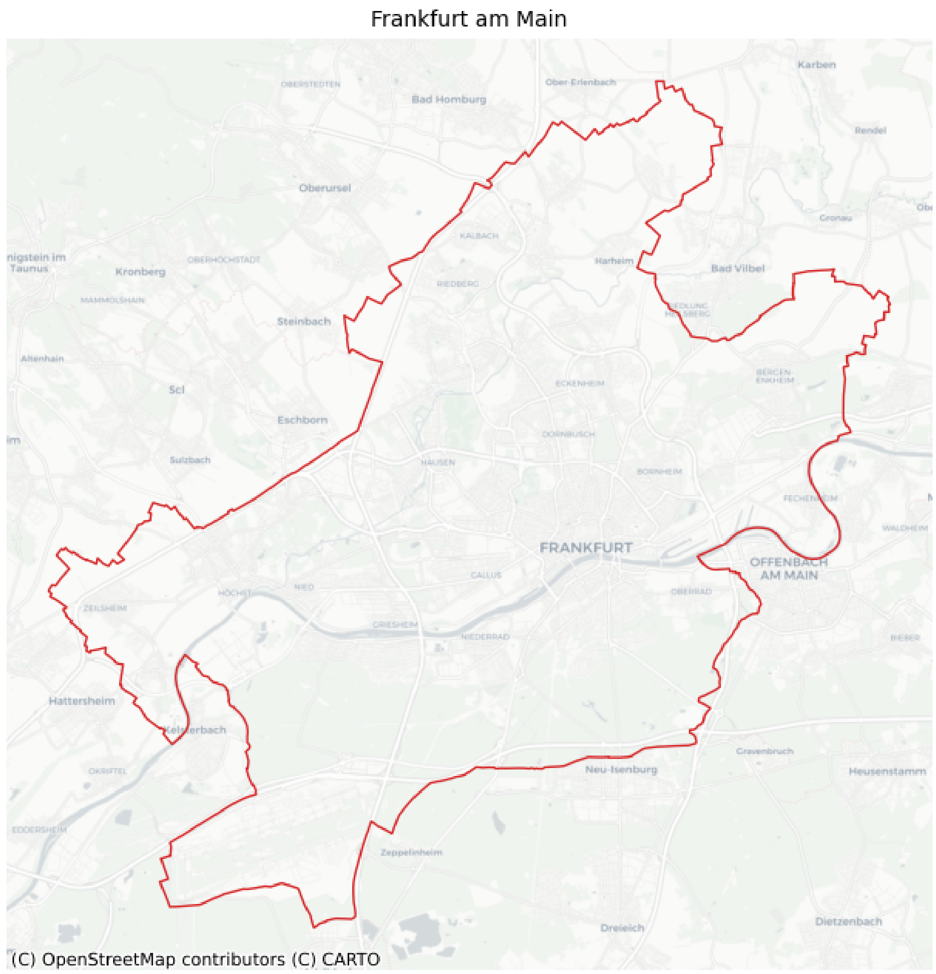
[4]:
```
from tesspy import Tessellation
```

Area

We use **Frankfurt am Main** in Germany as a case study. First, we get the city boundary.

[5]:
```
ffm = Tessellation("Frankfurt am Main")
ffm_polygon = ffm.get_polygon()
```

[6]:
```
# visualization of area
ax = ffm_polygon.to_crs("EPSG:3857").plot(facecolor="none", edgecolor="tab:red", lw=1)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.set_title("Frankfurt am Main", fontsize=10)
plt.show()
```



nbsphinx-code-borderwhite

Tessellation

We tessellate the area using Voronoi Diagrams.

[7]:
```
poi_categories = ["shop", "building", "amenity", "office", "public_transport"]

# voronoi Diagrams
ffm_voronoi_kmeans = ffm.voronoi(poi_categories=poi_categories, n_polygons=1000)
```

[8]:
```
# adding an ID to tiles
ffm_voronoi_kmeans.reset_index(inplace=True)
ffm_voronoi_kmeans.rename(columns={"index": "tile_id"}, inplace=True)

# check polygons
ffm_voronoi_kmeans.tail()
```

[8]:
```
     tile_id                                           geometry
917      994  POLYGON ((8.69887 50.10739, 8.70045 50.10867, ...
918      995  POLYGON ((8.72891 50.09957, 8.73156 50.09439, ...
919      996  POLYGON ((8.64662 50.11059, 8.64275 50.10865, ...
```

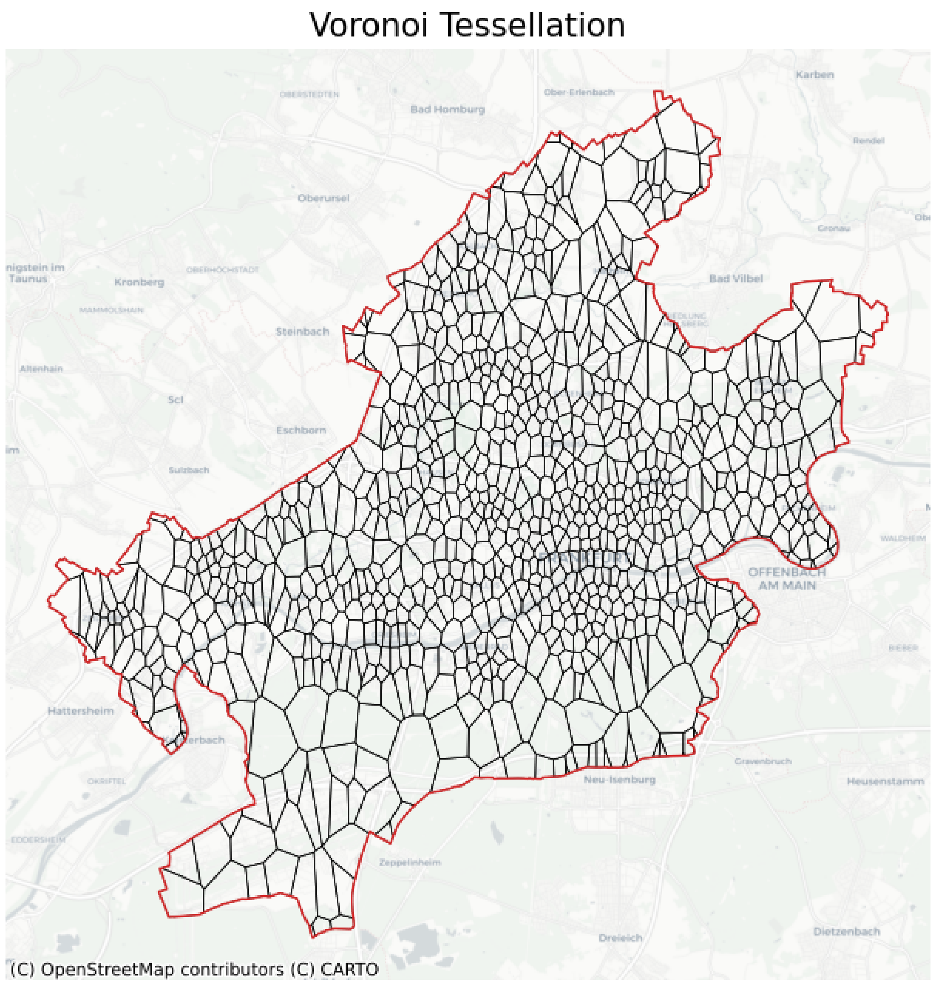
(continues on next page)

(continued from previous page)

```
920      997  POLYGON ((8.66783 50.10082, 8.67002 50.09716, ...
921      999  POLYGON ((8.61203 50.18259, 8.61447 50.18296, ...
```

[9]:

```
ax = ffm_voronoi_kmeans.to_crs("EPSG:3857").plot(
    facecolor="none", edgecolor="k", lw=0.5
)
ffm_polygon.to_crs("EPSG:3857").plot(ax=ax, facecolor="none", edgecolor="tab:red", lw=1)
ctx.add_basemap(ax=ax, source=ctx.providers.CartoDB.Positron)
ax.set_axis_off()
ax.set_title("Voronoi Tessellation", fontsize=15)
plt.show()
```



nbsphinx-code-borderwhite

Data Processing

We create count variables. For each polygon (tile), the number of each POI category is calculated.

```
[39]: # create a GeoDataFrame of POI data
poi_df = ffm.get_poi_data()
poi_geodata = gpd.GeoDataFrame(
    data=poi_df.drop(columns=["geometry"]),
    geometry=poi_df[["center_longitude", "center_latitude"]]
    .apply(Point, axis=1)
    .values,
    crs="EPSG:4326",
)
```

```
[11]: poi_geodata.head()
```

```
[11]:   type                                              tags  center_latitude  \
0  way  {'access': 'customers', 'amenity': 'parking', ...        50.107625
1  way  {'amenity': 'school', 'contact:email': 'schull...        50.167461
2  way  {'amenity': 'prison', 'barrier': 'fence', 'nam...        50.149950
3  way  {'addr:city': 'Frankfurt am Main', 'addr:house...        50.145195
4  way  {'addr:city': 'Frankfurt am Main', 'addr:postc...        50.052712

   center_longitude  amenity  building  office  public_transport   shop  \
0          8.734458     True     False   False             False  False
1          8.675039     True     False   False             False  False
2          8.695300     True     False   False             False  False
3          8.650782     True     False   False             False  False
4          8.568809    False      True   False             False  False

                   geometry
0  POINT (8.73446 50.10762)
1  POINT (8.67504 50.16746)
2  POINT (8.69530 50.14995)
3  POINT (8.65078 50.14519)
4  POINT (8.56881 50.05271)
```

```
[12]: # Count POI
poi_counts = gpd.sjoin(
    ffm_voronoi_kmeans, poi_geodata, how="left", predicate="contains"
)
poi_counts[poi_categories] = poi_counts[poi_categories].applymap(
    lambda x: np.nan if not x else x
)
poi_counts = poi_counts.groupby(by="tile_id").count().reset_index()
poi_counts.head()
```

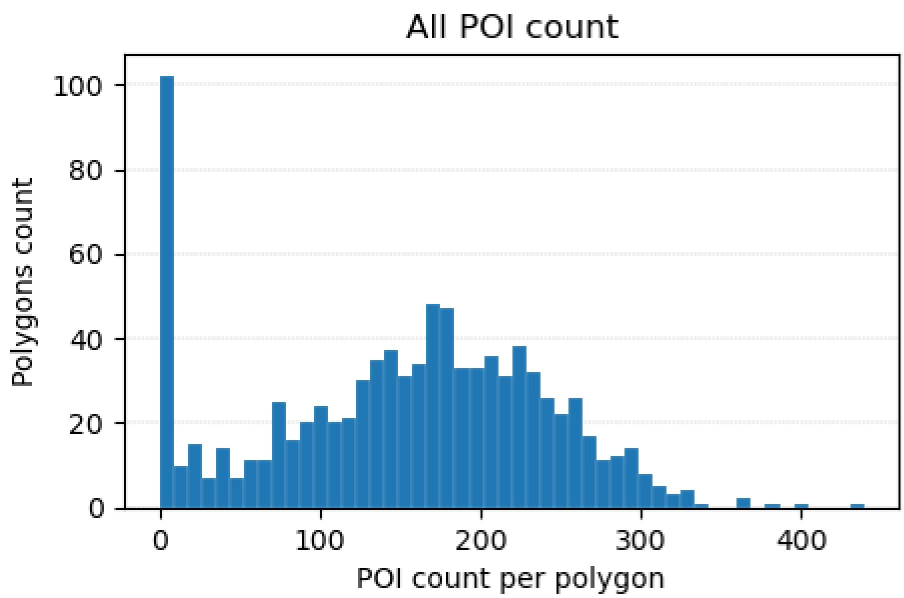
```
[12]:   tile_id  geometry  index_right  type  tags  center_latitude  \
0        0       255          255   255   255              255
1        1       297          297   297   297              297
2        2       216          216   216   216              216
3        3       171          171   171   171              171
4        4       243          243   243   243              243
```

(continues on next page)

(continued from previous page)

```
   center_longitude  amenity  building  office  public_transport  shop
0               255        8       234       0                10     3
1               297       20       277       0                 0     0
2               216        0       214       0                 2     0
3               171        3       169       0                 0     0
4               243       11       225       0                 4     3
```

[13]:
```
fig, ax = plt.subplots(figsize=(5, 3))
poi_counts["type"].plot(
    kind="hist", bins=50, ax=ax, lw=0.1, edgecolor="white", zorder=3
)
ax.set_title("All POI count")
ax.set_xlabel("POI count per polygon")
ax.set_ylabel("Polygons count")
ax.grid(axis="y", lw=0.3, ls="--", zorder=0)
```



nbsphinx-code-borderwhite

We normalize the data using `MinMaxScaler`.

[14]:
```
data = poi_counts[poi_categories].values
data_norm = MinMaxScaler().fit_transform(data)
```

Clustering

K-Means is used for clustering polygons.

[15]:
```
kmc = KMeans(n_clusters=4, random_state=0).fit(data_norm)
```

[22]:
```
clustering_results = gpd.GeoDataFrame(
    geometry=ffm_voronoi_kmeans["geometry"], data=poi_counts[poi_categories]
)
clustering_results["cluster"] = kmc.labels_
```

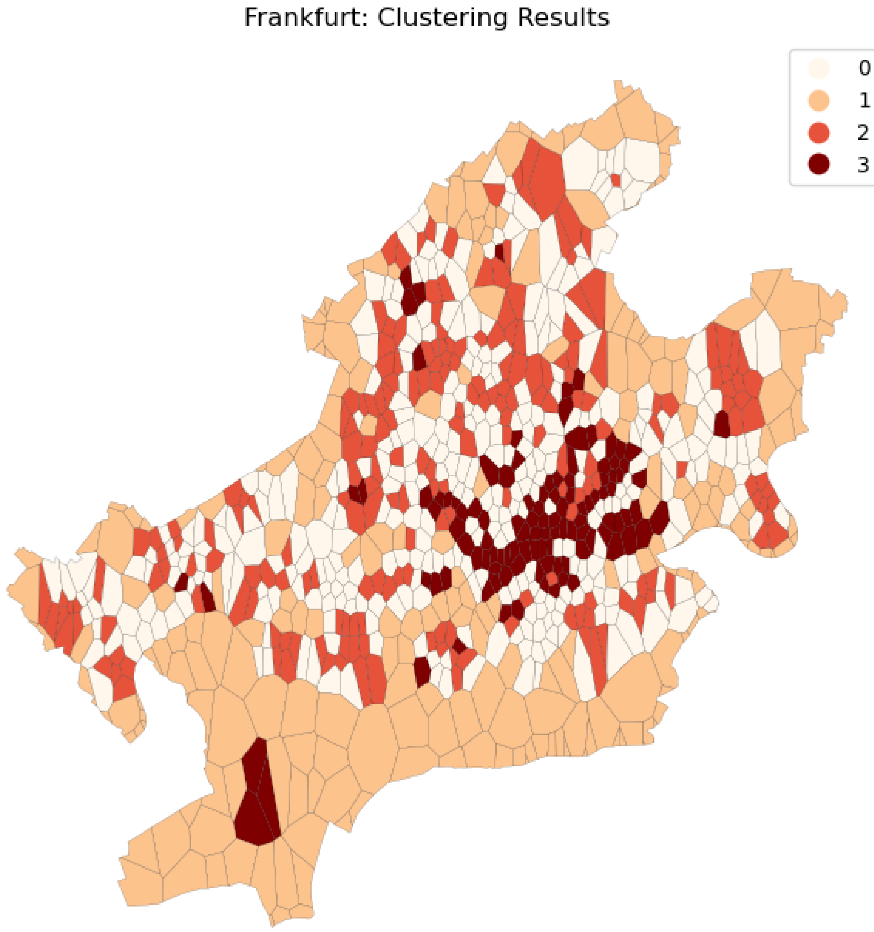
(continues on next page)

(continued from previous page)

```
# pretiffy the results
dct_ = dict(
    zip(
        clustering_results.groupby("cluster")
        .count()
        .sort_values("geometry", ascending=False)
        .index,
        clustering_results.groupby("cluster").count().index,
    )
)


def sort_class_labels(i, dct):
    return dct[i]


clustering_results["cluster"] = clustering_results["cluster"].apply(
    lambda x: sort_class_labels(x, dct_)
)
```

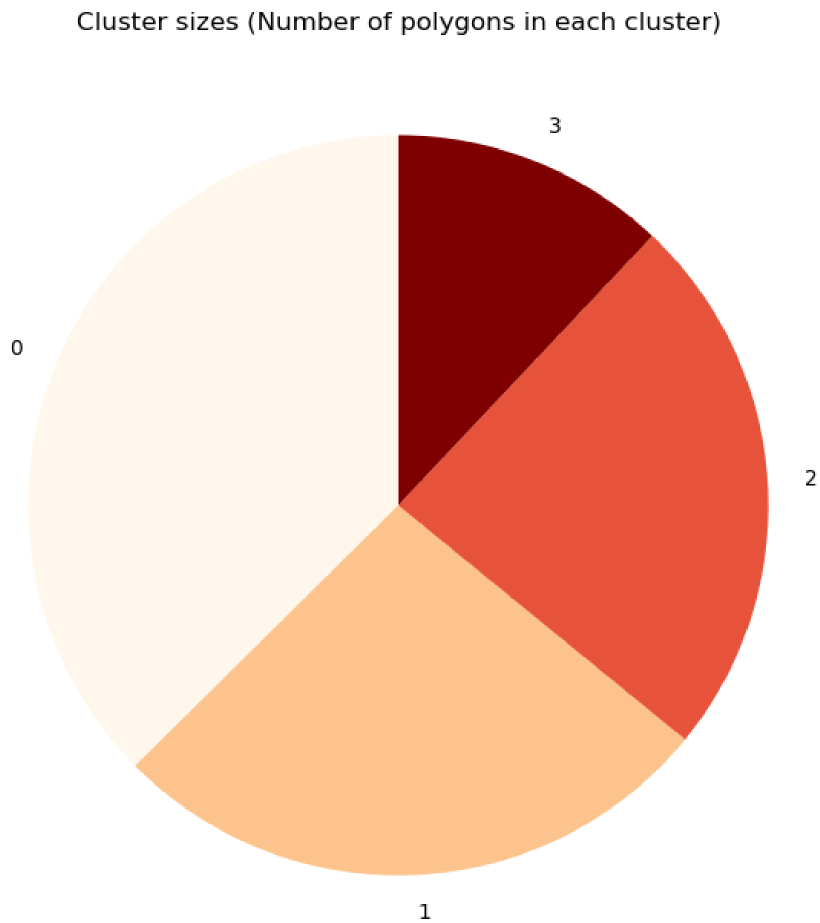
Results

[23]:
```
fig, ax = plt.subplots()
ax.set_axis_off()
clustering_results.plot(
    column="cluster",
    categorical=True,
    cmap="OrRd",
    linewidth=0.1,
    edgecolor="k",
    legend=True,
    ax=ax,
)
ax.set_title(f"Frankfurt: Clustering Results")
plt.show()
```



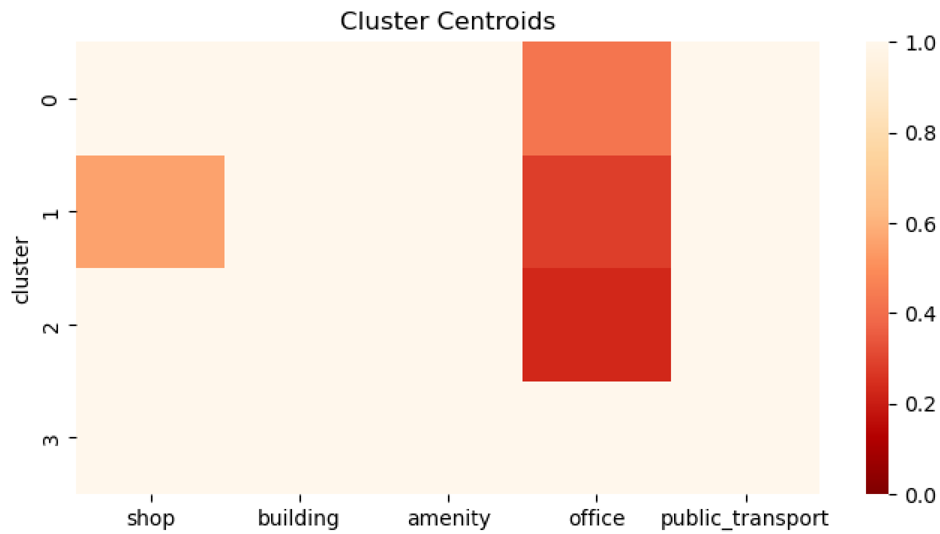
nbsphinx-code-borderwhite

[25]:
```
from matplotlib import cm

cs = cm.OrRd(np.linspace(0, 1, 4))

fig, ax = plt.subplots()
labels = clustering_results["cluster"].value_counts().index
sizes = clustering_results["cluster"].value_counts().values
ax.pie(sizes, labels=labels, startangle=90, colors=cs)
ax.set_title(f"Cluster sizes (Number of polygons in each cluster)")

plt.show()
```



nbsphinx-code-borderwhite

[37]:
```
cluster_centroids = clustering_results.groupby("cluster").mean()
fig, ax = plt.subplots(figsize=(8, 4))
sns.heatmap(cluster_centroids, ax=ax, cmap="OrRd_r", vmin=0, vmax=1)
ax.set_title(f"Cluster Centroids")
plt.show()
```



nbsphinx-code-borderwhite

Example for the concept of LGU

This notebook shows how tessellation and its functionalities can help to create Local Geographic Units. Local Geographic Units are the combination of spatial discretization with additional information assigned to each tile. This conecept makes further analysis easy.

[1]:
```
import tesspy as tp
from tesspy import Tessellation
import matplotlib.pyplot as plt
from tesspy.tessellation import count_poi_per_tile
import numpy as np
import contextily as ctx

import warnings
warnings.filterwarnings('ignore')
```

Tessellation

First, we define a tessellation area. In the case Frankfurt am Main and Nairobi are chosen as case study. A regular tessellation method, i.e., hexagons, and an irregular method, i.e., city blocks, are used.

[2]:
```
ffm = Tessellation("Frankfurt am Main")
africa = Tessellation("Nairobi")
```

[3]:
```
africa_hex = africa.hexagons(resolution=9)
```

[4]:
```
ffm_cb = ffm.city_blocks(n_polygons=1000)
```

```
MultiPolygon found. Splitting it up...
```

Creating LGU datasets

Using the function `count_poi_per_tile`, an LGU dataset is created based on the tessellation and additional information (POI).

[5]:
```
gdf_africa_hex = count_poi_per_tile("Nairobi", africa_hex, poi_categories=["amenity",
↪"building"])
gdf_ffm_cb = count_poi_per_tile("Frankfurt", ffm_cb, poi_categories=["amenity","leisure",
↪ "office"])
```

[7]:
```
gdf_ffm_cb
```

[7]:
```
        cityblock_id                                           geometry  \
0        cityblockID0  POLYGON ((8.68230 50.10893, 8.68232 50.10887, ...
1        cityblockID1  POLYGON ((8.67928 50.10815, 8.67876 50.10797, ...
2        cityblockID2  POLYGON ((8.68671 50.11190, 8.68670 50.11194, ...
3        cityblockID3  POLYGON ((8.69298 50.11307, 8.69298 50.11309, ...
4        cityblockID4  POLYGON ((8.68545 50.10926, 8.68522 50.10925, ...
...               ...                                                ...
1383  cityblockID1383  POLYGON ((8.59033 50.03088, 8.59031 50.03080, ...
1384  cityblockID1384  POLYGON ((8.60164 50.04618, 8.60279 50.04852, ...
1385  cityblockID1385  POLYGON ((8.60788 50.04855, 8.60787 50.04857, ...
1386  cityblockID1386  POLYGON ((8.58302 50.02316, 8.58270 50.02307, ...
1387  cityblockID1387  POLYGON ((8.58384 50.02264, 8.58339 50.02278, ...

      amenity  leisure  office
0        74.0      1.0     2.0
1        60.0      1.0     1.0
2       175.0      3.0     7.0
3        95.0     10.0     4.0
4        40.0      1.0     0.0
...       ...      ...     ...
1383      0.0      0.0     0.0
1384      2.0      0.0     0.0
1385      0.0      0.0     0.0
1386      3.0      0.0     1.0
1387      0.0      0.0     0.0

[1388 rows x 5 columns]
```

[11]:
```
gdf_africa_hex
```

[11]:
```
               hex_id                                           geometry  \
0     897a6e54c37ffff  POLYGON ((36.72584 -1.36099, 36.72588 -1.35910...
1     897a6e47117ffff  POLYGON ((36.73657 -1.24548, 36.73661 -1.24360...
2     897a6e5a343ffff  POLYGON ((37.03414 -1.27108, 37.03418 -1.26920...
3     897a6e5505bffff  POLYGON ((36.83938 -1.32114, 36.83942 -1.31926...
4     897a6e5408bffff  POLYGON ((36.78535 -1.36950, 36.78539 -1.36761...
...               ...                                                ...
6328  897a6e50817ffff  POLYGON ((36.88515 -1.35163, 36.88519 -1.34974...
6329  897a6e54c6fffff  POLYGON ((36.73271 -1.34999, 36.73275 -1.34811...
6330  897a6e51153ffff  POLYGON ((36.97667 -1.33054, 36.97671 -1.32866...
6331  897a6e40387ffff  POLYGON ((36.85312 -1.21716, 36.85316 -1.21527...
6332  897a6e0966bffff  POLYGON ((36.69048 -1.30834, 36.69052 -1.30646...
```

(continues on next page)

(continued from previous page)

```
      amenity  building
0         3.0      64.0
1         0.0       0.0
2         0.0       0.0
3        11.0      65.0
4         0.0       0.0
...       ...       ...
6328      0.0       0.0
6329      0.0      45.0
6330      0.0       0.0
6331      0.0      33.0
6332      0.0       0.0

[6333 rows x 4 columns]
```

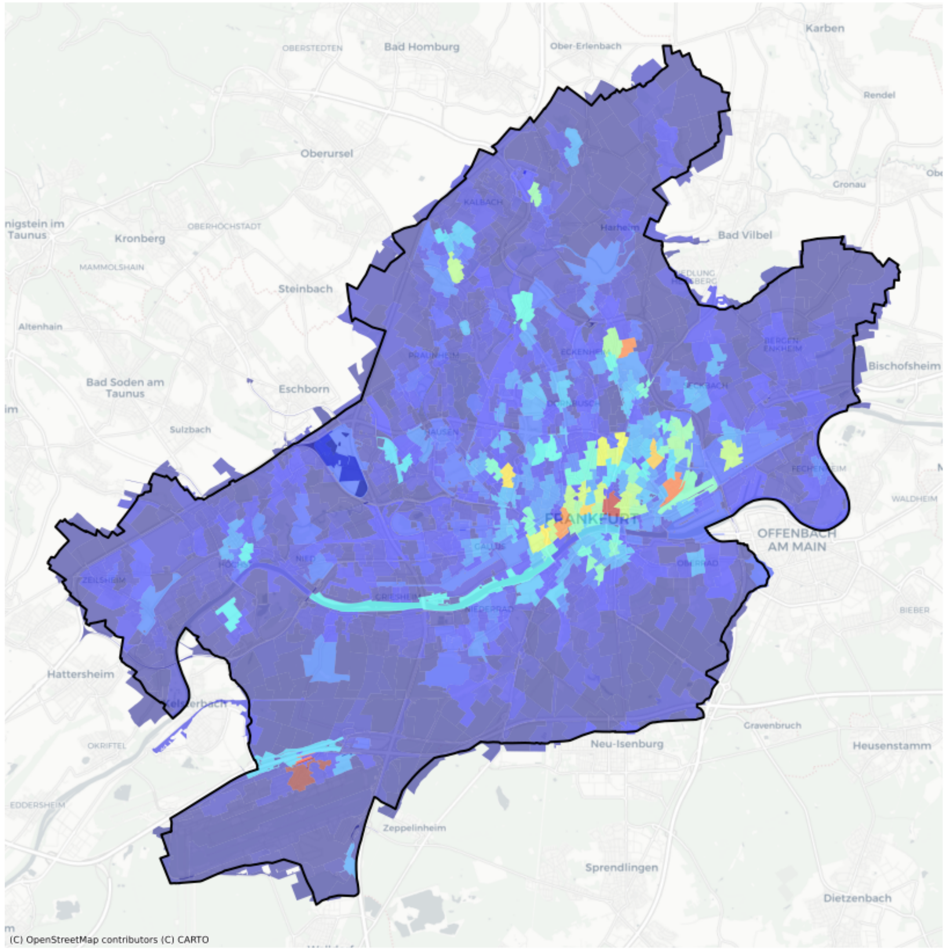
[12]:

```
gdf_africa_hex["overall_count"]=gdf_africa_hex.iloc[:,-2:].sum(axis=1)
gdf_ffm_cb["overall_count"]=gdf_ffm_cb.iloc[:,-3:].sum(axis=1)
```

[14]:

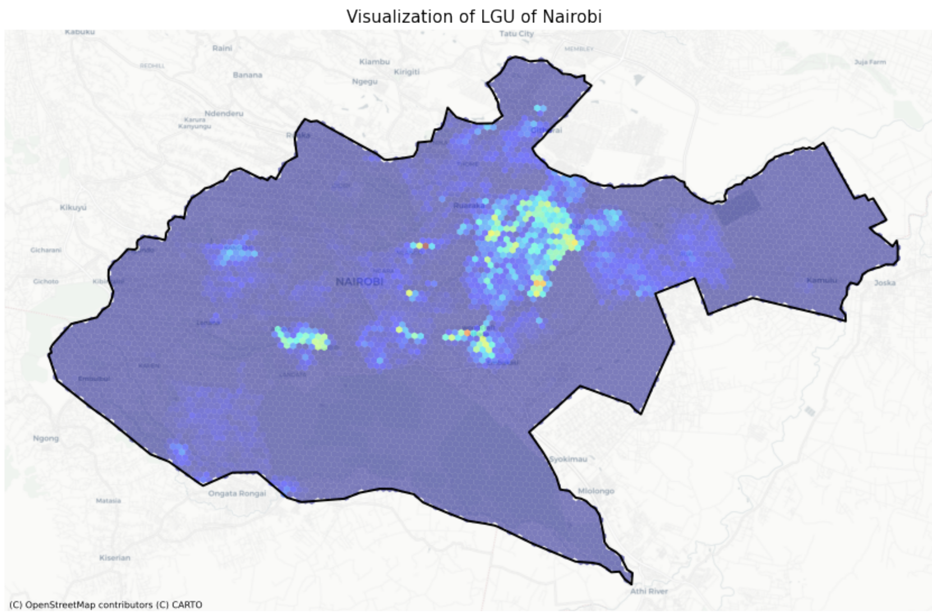
```
fig, axs = plt.subplots(figsize=(15,15))

ffm.get_polygon().to_crs('EPSG:3857').boundary.plot(ax=axs, linewidth=2, color='black')
gdf_ffm_cb.to_crs('EPSG:3857').plot(column='overall_count',cmap='jet', ax=axs,alpha=0.5)
ctx.add_basemap(ax=axs, source=ctx.providers.CartoDB.Positron, crs='EPSG:3857')


axs.axis('off')
axs.set_title(f"Visualization of LGU of Frankfurt", fontsize=15)

plt.show()
```

Visualization of LGU of Frankfurt



nbsphinx-code-borderwhite

[22]:
```
fig, axs = plt.subplots(figsize=(15,15))

africa.get_polygon().to_crs('EPSG:3857').boundary.plot(ax=axs, linewidth=2, color='black
↪')
gdf_africa_hex.to_crs('EPSG:3857').plot(column='overall_count',cmap='jet', ax=axs,
↪alpha=0.5)
ctx.add_basemap(ax=axs, source=ctx.providers.CartoDB.Positron, crs='EPSG:3857')


axs.axis('off')
axs.set_title(f"Visualization of LGU of Nairobi", fontsize=15)

plt.show()
```



nbsphinx-code-borderwhite

```
[ ]:
```

4.5.2 tesspy package

tesspy.poi_data

class tesspy.poi_data.**POIdata**(*area*, *poi_categories*, *timeout*, *verbose*)

Bases: `object`

This class creates a query for the investigated area and POI categories. The query is sent to osm using overpass API and the data is retrieved.

Parameters

- **area** (*geopandas.GeoDataFrame or str*) – GeoDataFrame must have a single shapely Polygon or MultiPolygon in geometry column and its CRS must be defined. str must be a name of a city, or an address of a region
- **poi_categories** (*A list of OSM primary map features or 'all'*) –
- **timeout** (*int*) – The TCP connection timeout for the overpass request
- **verbose** (*bool*) – If True, print information while computing

create_overpass_query_string()

creates the query string to be passed to overpass

Returns
query_string

Return type
str

get_poi_data()

sends the query to osm using the overpass API and gets the data

Returns
poi_df – A dataframe containing the POI, POI type, and coordinates

Return type
pandas.DataFrame

`static osm_primary_features()`

list of primary OSM features available at https://wiki.openstreetmap.org/wiki/Map_features

Returns
osm_primary_features_lst

Return type
list

`class tesspy.poi_data.RoadData(area, detail_deg=None, split_roads=True, verbose=False)`

Bases: `object`

This class creates a custom filter for the investigated area and highway types. The custom filter is used to collect road data using osmnx.

Parameters

- **area** (*geopandas.GeoDataFrame or str*) – GeoDataFrame must have a single shapely Polygon or MultiPolygon in geometry column and its CRS must be defined. str must be a name of a city, or an address of a region
- **detail_deg** (*int*) – integer to select the top (int) highway types
- **split_roads** (*bool*) – decide if LineStrings will be split up such that each LineString contains exactly 2 Points
- **verbose** (*bool*) – If True, print information while computing

`create_custom_filter()`

Creates custom filter that is used to collect road data with osmnx. The detail_deg is used to use only the top highway types.

Returns
custom_filter – string with highway types that is used to collect data using osmnx

Return type
list

`get_road_network()`

Collects the road network data based on the defined custom filter. The initial road data is based on graphs and transformed into a GeoDataFrame. Within the RoadData Class the user can divide to split the data such that each LineString (representing a street segment) contains only two points and not multiple points.

Returns
graph_edges_as_gdf – GeoDataFrame containing road network

Return type
geopandas.GeoDataFrane

`static osm_highway_types()`

list of OSM highway types available at https://wiki.openstreetmap.org/wiki/Key:highway

Returns
osm_highways_lst

Return type
list

tesspy.poi_data.**geom_ceil**(*coordinate*, *precision=4*)

tesspy.poi_data.**geom_floor**(*coordinate*, *precision=4*)

tesspy.tessellation

class tesspy.tessellation.**Tessellation**(*area*)

Bases: `object`

Creates a Tessellation object using a GeoDataFrame or a city name, which allows for different tessellation methods

Parameters

area (*geopandas.GeoDataFrame or str*) – GeoDataFrame must have a single shaply Polygon or MultiPolygon in geometry column and its CRS must be defined. str must be a name of a city, or an address of a region

Examples

```
>>> ffm= Tessellation('Frankfurt am Main')
```

adaptive_squares(*start_resolution: int*, *poi_categories=['amenity', 'building']*, *threshold=None*, *timeout=60*, *verbose=False*)

Generate adaptive squares based on the input POI data. Squares are created at the start resolution. Each square is broken into four smaller squares while the number of its POI exceeds the threshold. POI categories should be a list of OSM primary map features. A complete list can be found on the OSM website: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **start_resolution** (*int*) – Specifies the size of initial squares
- **poi_categories** (*A list of OSM primary map features or 'all'*) – default=[“amenity”, ‘building’] ‘all’ means all the available POI categories Possible values in the list: [‘aerialway’, ‘aeroway’, ‘amenity’, ‘barrier’, ‘boundary’, ‘building’, ‘craft’, ‘emergency’, ‘geological’, ‘healthcare’, ‘highway’, ‘historic’, ‘landuse’, ‘leisure’, ‘man_made’, ‘military’, ‘natural’, ‘office’, ‘place’, ‘power’, ‘public_transport’, ‘railway’, ‘route’, ‘shop’, ‘sport’, ‘telecom’, ‘tourism’, ‘water’, ‘waterway’]
- **threshold** (*int, default=None*) – Threshold for the number of POI in a single square. If square has more, it is divided into four squares. If None passed, the median number of POI per square in the initial level is used as threshold.
- **timeout** (*int, default=60*) – The TCP connection timeout for the request
- **verbose** (*bool, default=False*) – If True, print information while computing

Returns

df_adaptive_squares – Dataframe containing adaptive squares

Return type

pandas.DataFrame

city_blocks(*n_polygons=None*, *detail_deg=None*, *split_roads=True*, *verbose=False*)

Create city bocks (tiles) using road data from the area. To collect road data, specify the highway types by modifying detail_deg

Parameters

- **n_polygons** (*int, default = None*) – targeted number of city blocks, this is an approximation the final number of polygons can vary slightly
- **detail_deg** (*int, default = None*) – define the number of the top (osm) highway types to use for creating city blocks
- **split_roads** (*bool, default = True*) – if True, LineStrings are split up such that each LineString contains exactly 2 Points This usually make the polygonizing more robust but slower.
- **verbose** (*bool, default = False*) – If True, print information while computing

Returns
final_city_blocks – GeoDataFrame with city block tiles

Return type
geopandas.GeoDataFrame

`get_poi_data()`

Returns
area_gdf – the POI data in GeoDataFrame format

Return type
geopandas.GeoDataFrame

`get_polygon()`

Returns
area_gdf – the area polygon in GeoDataFrame format

Return type
geopandas.GeoDataFrame

`get_road_network()`

Returns
road_network – the road network data in GeoDataFrame format

Return type
geopandas.GeoDataFrame

`hexagons(resolution: int)`

Generate hexagon grid laying over the area

Parameters
resolution (*int*) – Specifies the size of hexagons A positive number between

Returns
df_h3_hexagons – Dataframe containing hexagons

Return type
pandas.DataFrame

`static osm_highway_types()`

list of all highway types

Returns
osm_highways_lst – list of highway types

Return type
list

`static osm_primary_features()`

list of primary OSM features available at https://wiki.openstreetmap.org/wiki/Map_features

Returns
osm_primary_features_lst

Return type
list

`squares(resolution: int)`

Generate square grid laying over the area

Parameters
resolution (*int*) – Specifies the size of squares A positive number between

Returns
df_qk_squares – Dataframe containing squares

Return type
pandas.DataFrame

`voronoi(cluster_algo='k-means', poi_categories=['amenity', 'building'], timeout=60, n_polygons=100, min_cluster_size=15, verbose=False)`

Generate Voronoi polygons based on the input POI data. POI categories should be a list of OSM primary map features. A complete list can be found on the OSM website: https://wiki.openstreetmap.org/wiki/Map_features

Parameters

- **cluster_algo** (*{'k-means', 'hdbscan', None}, default='k-means'*) – Algorithm for clustering the POI data before creating Voronoi generators. If None passed, POI data are directly used as generators.
- **poi_categories** (*A list of OSM primary map features or 'all'*) – default=["amenity", 'building'] 'all' means all the available POI categories Possible values in the list: ['aerialway', 'aeroway', 'amenity', 'barrier', 'boundary', 'building', 'craft', 'emergency', 'geological', 'healthcare', 'highway', 'historic', 'landuse', 'leisure', 'man_made', 'military', 'natural', 'office', 'place', 'power', 'public_transport', 'railway', 'route', 'shop', 'sport', 'telecom', 'tourism', 'water', 'waterway']
- **timeout** (*int, default=60*) – The TCP connection timeout for the request
- **n_polygons** (*int, default=100*) – Only when cluster_algo="k-means", approximate number of polygons to be created. Positive number and less than the initial POI numbers
- **min_cluster_size** (*int, default=15*) – Only when cluster_algo="hdbscan", minimum cluster size Positive number
- **verbose** (*bool, default=False*) – If True, print information while computing

Returns
df_voronoi – Dataframe containing Voronoi polygons

Return type
pandas.DataFrame

`tesspy.tessellation.count_poi_per_tile(city, gdf, poi_categories=['amenity', 'building'], timeout=120)`

Counts different POI-categories per tile. For each POI-categories an additional column is added to the Tessellation GeoDataFrame (gdf). After counting each POI-category the final Tessellation GeoDataFrame with the additional count-columns is returned.

Parameters

- **city** (*str*) – Tessellation object or string of the underlying city
- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame of the Tessellation for the underlying city
- **poi_categories** (*list, default=["amenity", 'building']*) – POI which will be count per tile. These POI may differ from the POI used to create the tessellation This can be a list of OSM primary map features or 'all'

  'all' means all the available POI categories Possible values in the list: ['aerialway', 'aeroway', 'amenity', 'barrier', 'boundary', 'building', 'craft', 'emergency', 'geological', 'healthcare', 'highway', 'historic', 'landuse', 'leisure', 'man_made', 'military', 'natural', 'office', 'place', 'power', 'public_transport', 'railway', 'route', 'shop', 'sport', 'telecom', 'tourism', 'water', 'waterway']
- **timeout** (*int, default=120*) – positive number indicating the time to wait for OSM to return POI data

Returns
gdf – Tessellation GeoDataFrame with additional columns (count_poi-columns are added)

Return type
geopandas.GeoDataFrame

`tesspy.tessellation.get_city_polygon(city: str)`

Gets the polygon of a city or an area

Parameters
city (*str*) – city must be a name of a city, or an address of a region

Returns
df_city – GeoDataFrame containing the polygon of the city

Return type
geopandas.GeoDataFrame

tesspy.tessellation_functions

`tesspy.tessellation_functions.count_poi(df, points)`

Counts the number POI in each tile

Parameters

- **df** (*geopandas.GeoDataFrame*) – GeoDataFrame containing the tiles (polygons)
- **points** (*geopandas.GeoDataFrame*) – GeoDataFrame containing the POI

Returns
final_df – GeoDataFrame containing the tiles and the POI count

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.create_blocks(road_network)`

This function uses polygonize to create blocks by using road data.

Parameters
road_network (*geopandas.GeoDataFrame*) – GeoDataFrame (coming from RoadData class) containing street segments

Returns
blocks – GeoDataFrame with polygons, that were created by using the road data

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`explode`**(*gdf*)

Multipolygon into multiple Polygons

Parameters
gdf (*geopandas.GeoDataFrame*) – GeoDataFrame that can have multi-part geometries that will be exploded

Returns
gdf_out – GeoDataFrame with single geometries

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`get_adaptive_squares`**(*input_gdf*, *threshold*)

Adaptive tessellation. Subdivides all squares of input tessellation where threshold is exceeded.

Parameters

- **input_gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame containing the tiles (polygons)
- **threshold** (*int*) – threshold, which controls the division of squares

Returns
gdf – GeoDataFrame containing the squares

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`get_h3_hexagons`**(*gdf*, *resolution*)

Hexagon tessellation based on the h3 implementation of Uber

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame containing the tiles (polygons)
- **resolution** (*int*) – Resolution, which controls the hexagon sizes

Returns
gdf – GeoDataFrame containing the hexagons

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`get_hierarchical_clustering_parameter`**(*coordinates*, *threshold*)

This function returns the distance_threshold that is used in the hierarchical clustering algorithm. Therefore, different distant_threshold are tested until the hierarchical clustering return less clusters than the input threshold. If no dist_threshold fulfills the threshold-requirement nothing is returned. In the main method tesspy.Tessellation.city_blocks() is a differentiation between both cases. If th is an integer, the hierarchical clustering use distance_threshold parameter, if th is None, the hierarchical clustering use n_clusters parameter with the users input n_polygons

Parameters

- **coordinates** (*numpy.ndarray*) – stock of coordinates of Data
- **threshold** (*int*) – positive number indicating the number of Polygons that should not be exceeded

Returns
th – defines the distance_threshold for hierarchical clustering

Return type
int

`tesspy.tessellation_functions.`**`get_rest_polygon`**(*blocks*, *area*)

This function creates the rest polygons that can occur by creating city blocks. Dead ends or vegetation at the boundary of the area will not be defined as blocks because there are no road to define a block. Since the tessellation should cover the whole area those "rest polygons" will be created by subtracting all blocks from the area to fill these gaps. The rest_polygons are most likely a Multi-Polygon which is exploded into many polygons. Using this, the city blocks method tessellates the whole area without gaps.

Parameters

- **blocks** (*geopandas.GeoDataFrame*) – GeoDataFrame containing city blocks
- **area** (*geopandas.GeoDataFrame*) – boundary polygon

Returns
rest_polygons – GeoDataFrame containing rest polygons

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`get_squares_polyfill`**(*gdf*, *zoom_level*)

Square tessellation based on the quadKeys concept

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame containing the tiles (polygons)
- **zoom_level** (*int*) – Resolution, which controls the square sizes

Returns
gdf – GeoDataFrame containing the squares

Return type
geopandas.GeoDataFrame

`tesspy.tessellation_functions.`**`split_linestring`**(*df*)

The linestring, which are split up will have the same osmid since it's still the same street

Parameters
df (*pandas.DataFrame*) – dataframe of shapely.LineStrings containing more than two points

Returns
dataset – dataframe with shapely.linestring with two points each

Return type
pandas.DataFrame

`tesspy.tessellation_functions.`**`voronoi_polygons`**(*sp_voronoi_obj*, *diameter*)

Voronoi Diagram. Create Voronoi polygons based on scipy Voronoi object

Parameters

- **sp_voronoi_obj** (*scipy.spatial.Voronoi*) – scipy Voronoi object is created using the point coordinates
- **diameter** (*float*) – controls the size of infinite polygons. It should be large enough depending on the input values.

Returns
gdf – Voronoi polygons

Return type
shapely.Polygon

4.5.3 All functions

- genindex

4.5.4 Contribution

All kinds of contributions are welcome:

- Improvement of code with new features, bug fixes, and bug reports
- Improvement of documentation
- Additional tests

If you want to contribute to code:

1. Fork the latest main branch
2. Create a dev environment: install dependencies and install tesspy in develop mode
3. Write failing tests
4. Write new code
5. Run tests and make sure they pass
6. Update documentation
7. Format code
8. Pull Request

If you have any ideas or questions, feel free to open an issue.

1. Fork tesspy

This consists of three main steps:

1. Forking tesspy
2. Cloning your forked repo
3. Sync your fork with the original tesspy repo

A fork is simply a copy of a repository. It allows to freely experiment with changes without having an effect on the original project. The easiest way to fork a repository (first step) is by using the web interface on GitHub.org:

1. Navigate to tesspy repository.
2. Click **Fork** in the top-right corner of the page.
3. Follow the procedure (select owner, add a description, etc.)
4. Click **Create fork**.

Then you need to clone your forked repo by using the command line:

```
$ git clone https://github.com/YOUR-USERNAME/tesspy tesspy
```

Finally, you need to sync your repository to the upstream (original project) tesspy repository:

```
$ cd tesspy
$ git remote add upstream https://github.com/siavash-saki/tesspy.git
```

Verify the new upstream repository by:

```
$ git remote -v
```

For more information about fork, you can visit the GitHub docs.

2. Create a dev env

First, you need to create a clean conda environment by:

```
$ conda create -c conda-forge -n tesspy_dev
```

Then, the tesspy dependencies must be installed as follows:

```
$ conda activate tesspy_dev
$ conda install -c conda-forge geopandas scipy h3-py osmnx hdbscan mercantile scikit-
↪learn
```

Next, navigate to the directory, where your clone of tesspy is located. Install tesspy in develop mode using:

```
$ python setup.py develop
```

This way, you have installed tesspy in develop mode. You can now track your changes in real-time in your dev environment and do not need to install the package each time you change something in the code.

3. Write failing tests

It is a good practice to start writing tests even before writing any code. All the tests should initially fail. Think about your desired feature and write corresponding test cases. All tests are in the `tests` directory. New tests must also be saved here. For more info on Test-driven development, take a look here.

4. Write new code

Then it is time to start modifying the code, writing new functions, building new features, etc. You should write to the point when your initially created test cases pass.

5. Run tests

You should then run the test suit inside your own clone of the repository using `pytest`.

6. Update documentation

tesspy documentation is in the folder `docs`. It is written using reStructuredText, which is explained here.

After adding any code or feature, please also add to the documentation. Make sure to check the documentation to build correctly by rendering it using `sphinx`.

7. Format code

tesspy uses the PEP8 standard and `black` to ensure a consistent code format throughout the project.

So, before committing your changes, format the code using `black`.

it is a good idea to integrate `black` into your IDE. For example, it is explained here how to use black with `pycharm`.

8. Pull Request

When you're finished making your changes and have made sure everything is working properly, you can submit a Pull Request. You can find more information on PRs here.

4.5.5 Acknowledgements

`tesspy` is the result of the research project ClusterMobil conducted by the Research Lab for Urban Transport. This research project is funded by the state of Hesse and HOLM funding under the "Innovations in Logistics and Mobility" measure of the Hessian Ministry of Economics, Energy, Transport and Housing. [HA Project No.: 1017/21-19]

PYTHON MODULE INDEX

t

INDEX

V