momepy Documentation

Release 0.3.0+27.g5b7015a

Martin Fleischmann

Nov 02, 2020

CONTENTS

momepy

Urban Morphology Measuring Toolkit

CHAPTER

ONE

INTRODUCTION

Momepy is a library for quantitative analysis of urban form - urban morphometrics. It is built on top of GeoPandas, PySAL and networkX.

> *momepy* stands for Morphological Measuring in Python

Some of the functionality that momepy offers:

- Measuring **dimensions** of morphological elements, their parts, and aggregated structures.
- Quantifying **shapes** of geometries representing a wide range of morphological features.
- Capturing **spatial distribution** of elements of one kind as well as relationships between different kinds.
- Computing density and other types of **intensity** characters.
- Calculating **diversity** of various aspects of urban form.
- Capturing **connectivity** of urban street networks
- Generating relational **elements** of urban form (e.g. morphological tessellation)

Momepy aims to provide a wide range of tools for a systematic and exhaustive analysis of urban form. It can work with a wide range of elements, while focused on building footprints and street networks.

Comments, suggestions, feedback, and contributions, as well as bug reports, are very welcome.

https://github.com/martinfleis/momepy

CHAPTER

TWO

INSTALL

You can install `momepy` using Conda from conda-forge (recommended):

```
conda install -c conda-forge momepy
```

or from PyPI using pip:

```
pip install momepy
```

See the *installation docs* for detailed instructions. Momepy depends on python geospatial stack, which might cause some dependency issues.

CHAPTER

THREE

EXAMPLES

```
coverage = momepy.AreaRatio(tessellation, buildings, left_areas=tessellation.area,
                            right_areas='area', unique_id='uID')
tessellation['CAR'] = coverage.series
```

```
area_simpson = momepy.Simpson(tessellation, values='area',
                              spatial_weights=sw3,
                              unique_id='uID')
tessellation['area_simpson'] = area_simpson.series
```

```
G = momepy.straightness_centrality(G)
```

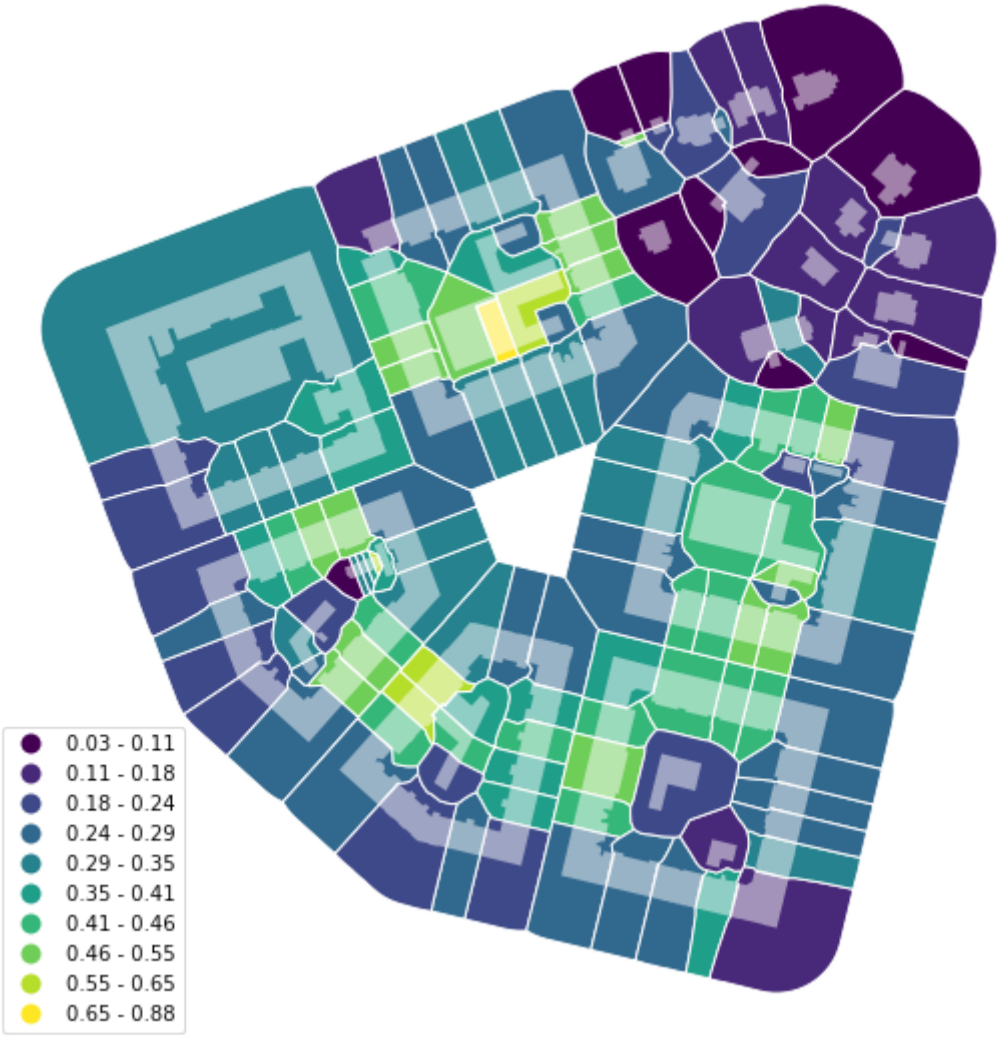


Fig. 1: Coverage Area Ratio

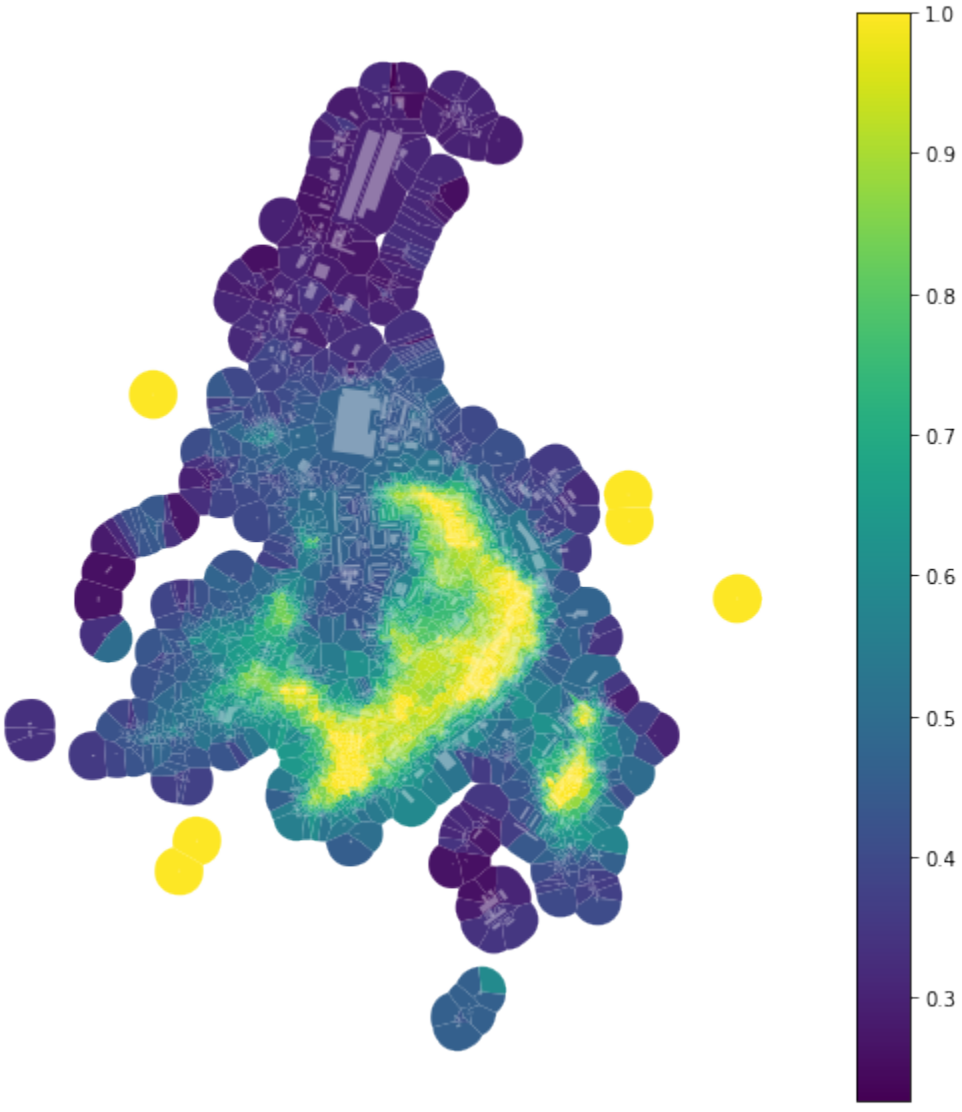


Fig. 2: Local Simpson's diversity of area

Fig. 3: Straightness centrality

CHAPTER

FOUR

CITING

To cite momepy please use following software paper published in the JOSS.

Fleischmann, M. (2019) 'momepy: Urban Morphology Measuring Toolkit', Journal of Open Source Software, 4(43), p. 1807. doi: 10.21105/joss.01807.

BibTeX:

```
@article{fleischmann_2019,
    author={Fleischmann, Martin},
    title={momepy: Urban Morphology Measuring Toolkit},
    journal={Journal of Open Source Software},
    year={2019},
    volume={4},
    number={43},
    pages={1807},
    DOI={10.21105/joss.01807}
}
```

CHAPTER

FIVE

CONTRIBUTING TO MOMEPY

Contributions of any kind to momepy are more than welcome. That does not mean new code only, but also improvements of documentation and user guide, additional tests (ideally filling the gaps in existing suite) or bug report or idea what could be added or done better.

All contributions should go through our GitHub repository. Bug reports, ideas or even questions should be raised by opening an issue on the GitHub tracker. Suggestions for changes in code or documentation should be submitted as a pull request. However, if you are not sure what to do, feel free to open an issue. All discussion will then take place on GitHub to keep the development of momepy transparent.

If you decide to contribute to the codebase, ensure that you are using an up-to-date `master` branch. The latest development version will always be there, including the documentation (powered by `sphinx`).

Details are available in the *contributing guide*.

CHAPTER

SIX

GET IN TOUCH

If you have a question regarding momepy, feel free to open an issue on GitHub. Eventually, you can contact us on dev@momepy.org.

CHAPTER

SEVEN

ACKNOWLEDGEMENTS

Initial release of momepy was a result of a research of Urban Design Studies Unit (UDSU) supported by the Axel and Margaret Ax:son Johnson Foundation as a part of "The Urban Form Resilience Project" in partnership with University of Strathclyde in Glasgow, UK. Further development was supported by Geographic Data Science Lab of the University of Liverpool wihtin Urban Grammar AI research project.

CHAPTER

EIGHT

DOCUMENTATION CONTENTS

8.1 Install

Momepy, similar to GeoPandas, can be a bit complicated to install. However, if you follow recommended instructions below, there should be no issue. For more details on issues with geospatial python stack, please refer to GeoPandas installation instructions.

8.1.1 Install via Conda

As `momepy` is dependent on geopandas and other spatial packages, we recommend to install all dependencies via conda from conda-forge:

```
conda install -c conda-forge momepy
```

Conda should be able to resolve any dependency conflicts and install momepy together with all necessary dependencies.

If you do not have conda-forge in your conda channels, you can add it using:

```
conda config --add channels conda-forge
```

To ensure that all dependencies will be installed from conda-forge, we recommend using strict channel priority:

```
conda config --env --set channel_priority strict
```

Note: We strongly recommend to install everything from the *conda-forge* channel. Mixture of conda channels or conda and pip packages can lead to import problems.

Creating a new environment for momepy

If you want to make sure, that everything will work as it should, you can create a new conda environment for momepy. Assuming we want to create a new environment called `momepy_env`:

```
conda create -n momepy_env
conda activate momepy_env
conda config --env --add channels conda-forge
conda config --env --set channel_priority strict
conda install momepy
```

8.1.2 Install via pip

Momepy is also available on PyPI, but ensure that all dependencies are properly installed before installing momepy. Some C dependencies are causing problems with installing using pip only:

```
pip install momepy
```

8.1.3 Install from the repository

If you want to work with the latest development version of momepy, you can do so by cloning GitHub repository and installing momepy from local directory:

```
git clone https://github.com/martinfleis/momepy.git
cd momepy
pip install .
```

Alternatively, you can install the latest version directly from GitHub:

```
pip install git+git://github.com/martinfleis/momepy.git
```

Installing directly from repository might face the same dependency issues as described above regarding installing using pip. To ensure that environment is properly prepared and every dependency will work as intended, you can install them using conda before installing development version of momepy:

```
conda install -c conda-forge geopandas networkx libpysal tqdm
```

8.1.4 Dependencies

Required dependencies:

- geopandas
- libpysal (>= 4.1.0)
- networkx
- tqdm

Some functions also depend on additional packages, which are optional:

- mapclassify (>= 2.1.1)
- inequality

or

- pysal (contains both inequality and mapclassify)

8.2 User Guide

This user guide covers essential features of momepy, mostly in the form of interactive Jupyter notebooks. Reading this guide, you will learn:

- data structure used in momepy,
- how to generate morphological elements like morphological tessellation and link them all together,
- how to calculate simple morphometric characters,
- how to calculate morphometric characters based on multiple sources,
- how to use spatial weights matrix with momepy, and
- how to do network analysis.

Notebooks cover just a small selection of functions as an illustration of principles. For a full overview of momepy capabilities, head to API.

8.2.1 Getting started

An introduction to momepy

Momepy is a library for quantitative analysis of urban form - urban morphometrics. It is built on top of GeoPandas, PySAL and networkX.

Some of the functionality that momepy offers:

- Measuring *dimensions* of morphological elements, their parts, and aggregated structures.
- Quantifying *shapes* of geometries representing a wide range of morphological features.
- Capturing *spatial distribution* of elements of one kind as well as relationships between different kinds.
- Computing density and other types of *intensity* characters.
- Calculating *diversity* of various aspects of urban form.
- Capturing *connectivity* of urban street networks
- Generating relational *elements* of urban form (e.g. morphological tessellation)

Momepy aims to provide a wide range of tools for a systematic and exhaustive analysis of urban form. It can work with a wide range of elements, while focused on building footprints and street networks.

Installation

Momepy can be easily installed using conda from `conda-forge`. For detailed installation instructions, please refer to the installation documentation.

```
conda install momepy -c conda-forge
```

Dependencies

To run all examples in this notebook, you will need some optional dependencies. `matplotlib`, `descartes` and `mapclassify` (or `pysal`) for plotting (these are GeoPandas dependencies) and `osmnx` to download data from OpenStreetMap. You can install all using the following commands:

```
conda install matplotlib descartes osmnx -c conda-forge
pip install mapclassify
```

Simple examples

Here are simple examples using embedded `bubenec` dataset (part of the Bubeneč neighborhood in Prague).

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

Morphometric analysis using momepy usually starts with GeoPandas GeoDataFrame containing morphological elements. In this case, we will begin with buildings represented by their footprints.

We have imported `momepy`, `geopandas` to handle the spatial data, and `matplotlib` to get a bit more control over plotting. To load `bubenec` data, we need to get retrieve correct path using `momepy.datasets.get_path("bubenec")`. The dataset itself is a GeoPackage with more layers, and now we want `buildings`.

[2]:
```
buildings = gpd.read_file(momepy.datasets.get_path('bubenec'),
                          layer='buildings')
```

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

Momepy uses classes for each measurable character, which stores the resulting values but also original input and all parameters used for the computation. The only exception is the *graph module*. To illustrate how momepy classes work, we can try to measure a few simple characters.

Area

`momepy.Area` measures the area of polygon geometry. It is a simple wrapper around `gdf.geoemtry.area`, included in momepy for consistency. `momepy.Area` does not need any attributes apart from source GeoDataFrame.

[4]:
```
blg_area = momepy.Area(buildings)
buildings['area'] = blg_area.series
```

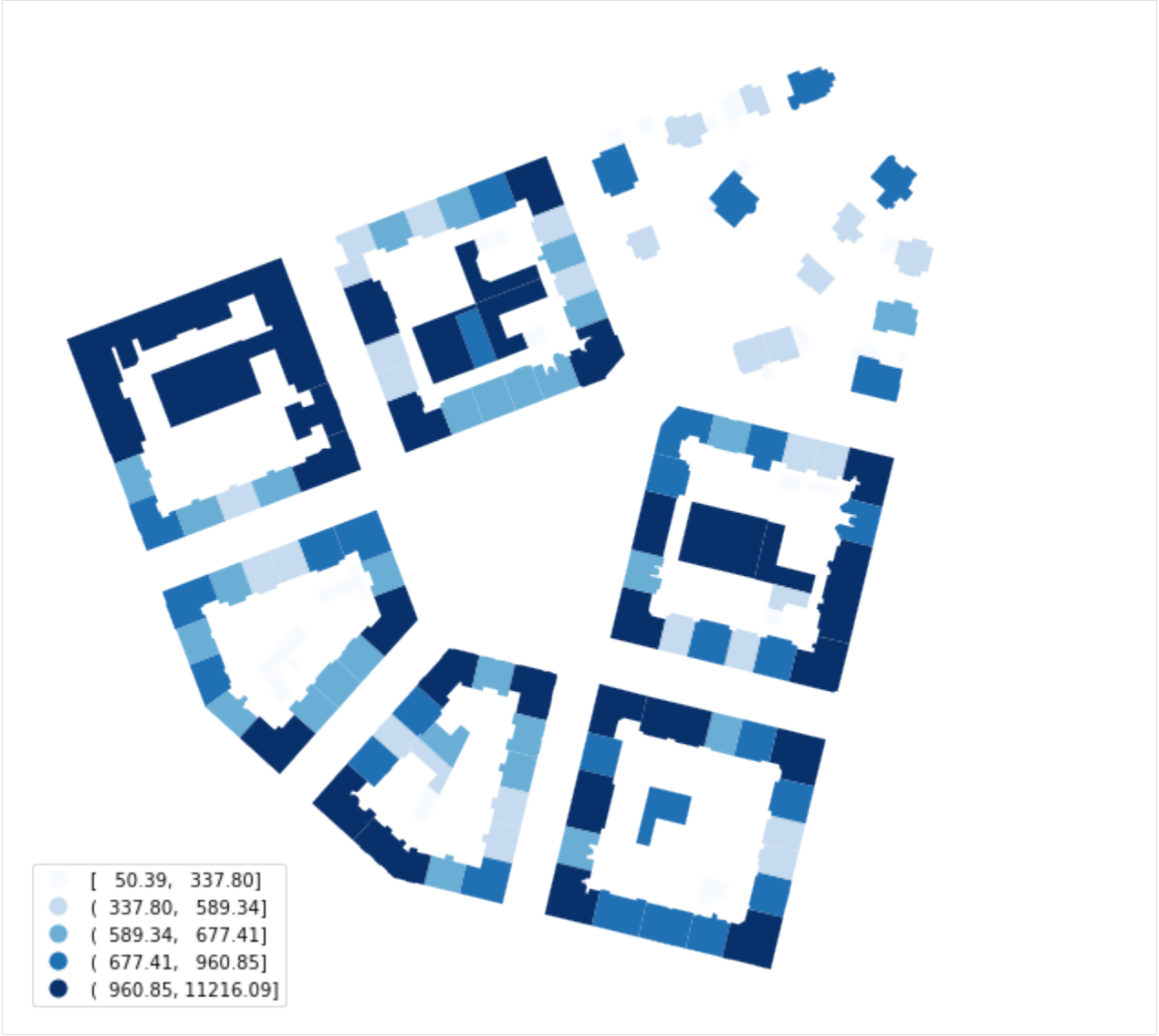
Tip: `.series` will give you resulting Pandas Series in most of the cases (unless there are more resulting Series).

[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot('area', ax=ax, legend=True, scheme='quantiles', cmap='Blues',
```

(continues on next page)

(continued from previous page)

```
                legend_kwds={'loc': 'lower left'})
ax.set_axis_off()
plt.show()
```



Above, we have calculated the area of each building and then extracted resulting values (pandas.Series) to save them as a new column of `buildings`. You can also get original GeoDataFrame:

[6]:
```
blg_area.gdf.head()
```

[6]:

	uID	geometry	area
0	1	POLYGON ((1603599.221 6464369.816, 1603602.984...	728.557495
1	2	POLYGON ((1603042.880 6464261.498, 1603038.961...	11216.093578
2	3	POLYGON ((1603044.650 6464178.035, 1603049.192...	641.059515
3	4	POLYGON ((1603036.557 6464141.467, 1603036.969...	903.746689
4	5	POLYGON ((1603082.387 6464142.022, 1603081.574...	641.629131

Equivalent rectangular index

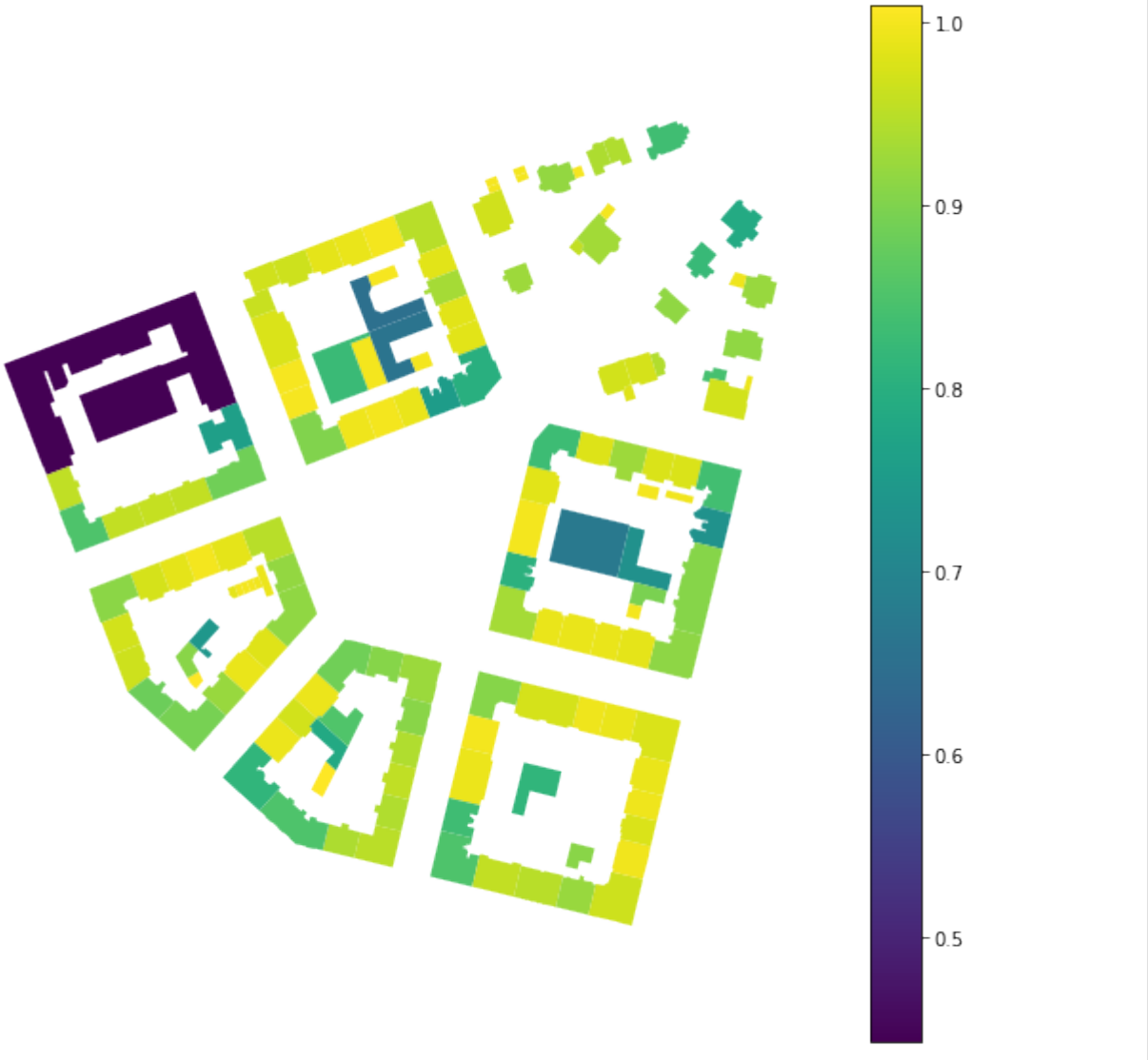
`momepy.EquivalentRectangularIndex` is an example of a morphometric character capturing the shape of each object. It can be calculated in an analogical way as the area above:

[7]:
```
blg_ERI = momepy.EquivalentRectangularIndex(buildings)
```

To calculate the equivalent rectangular index, we need to know the area and perimeter of each polygon. While momepy can compute both automatically, we might want to save time passing already computed areas directly:

[8]:
```
blg_ERI = momepy.EquivalentRectangularIndex(buildings, areas='area')
buildings['eri'] = blg_ERI.series
```

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot('eri', ax=ax, legend=True)
ax.set_axis_off()
plt.show()
```



Apart from resulting values stored in `blg_ERI.series` we can also see all values used for computation, including

perimeters we have not passed above.

[10]:
```
blg_ERI.areas.head()
```

[10]:
```
0      728.557495
1    11216.093578
2      641.059515
3      903.746689
4      641.629131
Name: area, dtype: float64
```

[11]:
```
blg_ERI.perimeters.head()
```

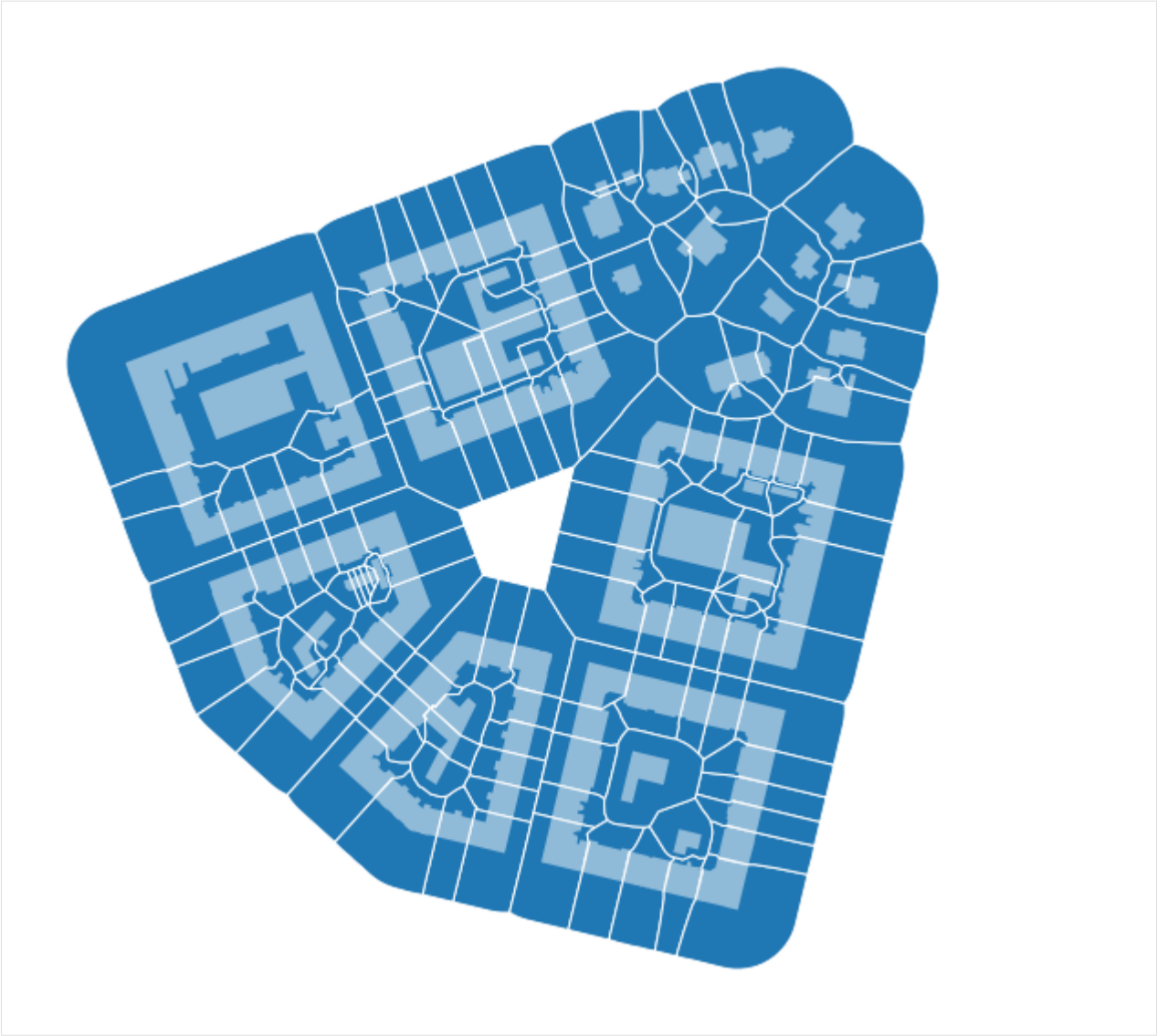
[11]:
```
0    137.186310
1    991.345770
2    107.488923
3    141.740042
4    107.158092
Name: mm_p, dtype: float64
```

Capturing relation between different elements

Urban form is a complex entity that needs to be represented by multiple morphological elements, and we need to be able to describe the relationship between them. With the absence of plots for our `bubenec` case, we can use *morphological tessellation* as the smallest spatial division.

[12]:
```
tessellation = gpd.read_file(momepy.datasets.get_path('bubenec'),
                             layer='tessellation')
```

[13]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, edgecolor='white')
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```



Coverage area ratio

Now we can calculate how big part of each tessellation cell is covered by a related building, using `momepy.AreaRatio`. Momepy classes usually accept any list-like object to be passed as values on top of a column name. Below we are passing Series to `left_areas` and column name to `right_areas`. Buildings have the same `uID` as related tessellation cells, which is used to link both together (see *Data Structure*).

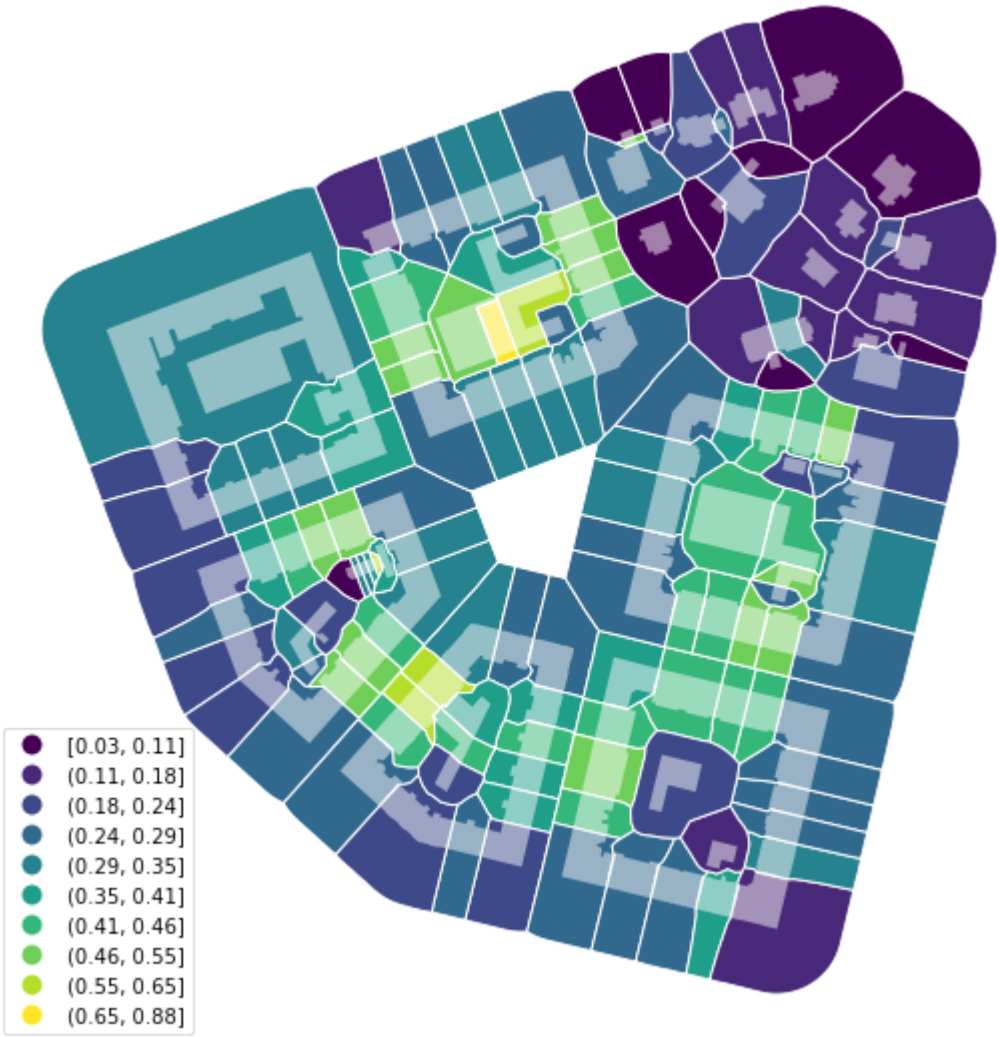
[14]:
```
coverage = momepy.AreaRatio(tessellation, buildings, left_areas=tessellation.area,
                            right_areas='area', unique_id='uID')
tessellation['CAR'] = coverage.series
```

[15]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot('CAR', ax=ax, edgecolor='white', legend=True, scheme='NaturalBreaks
↪', k=10, legend_kwds={'loc': 'lower left'})
buildings.plot(ax=ax, color='white', alpha=.5)
```

(continues on next page)

(continued from previous page)

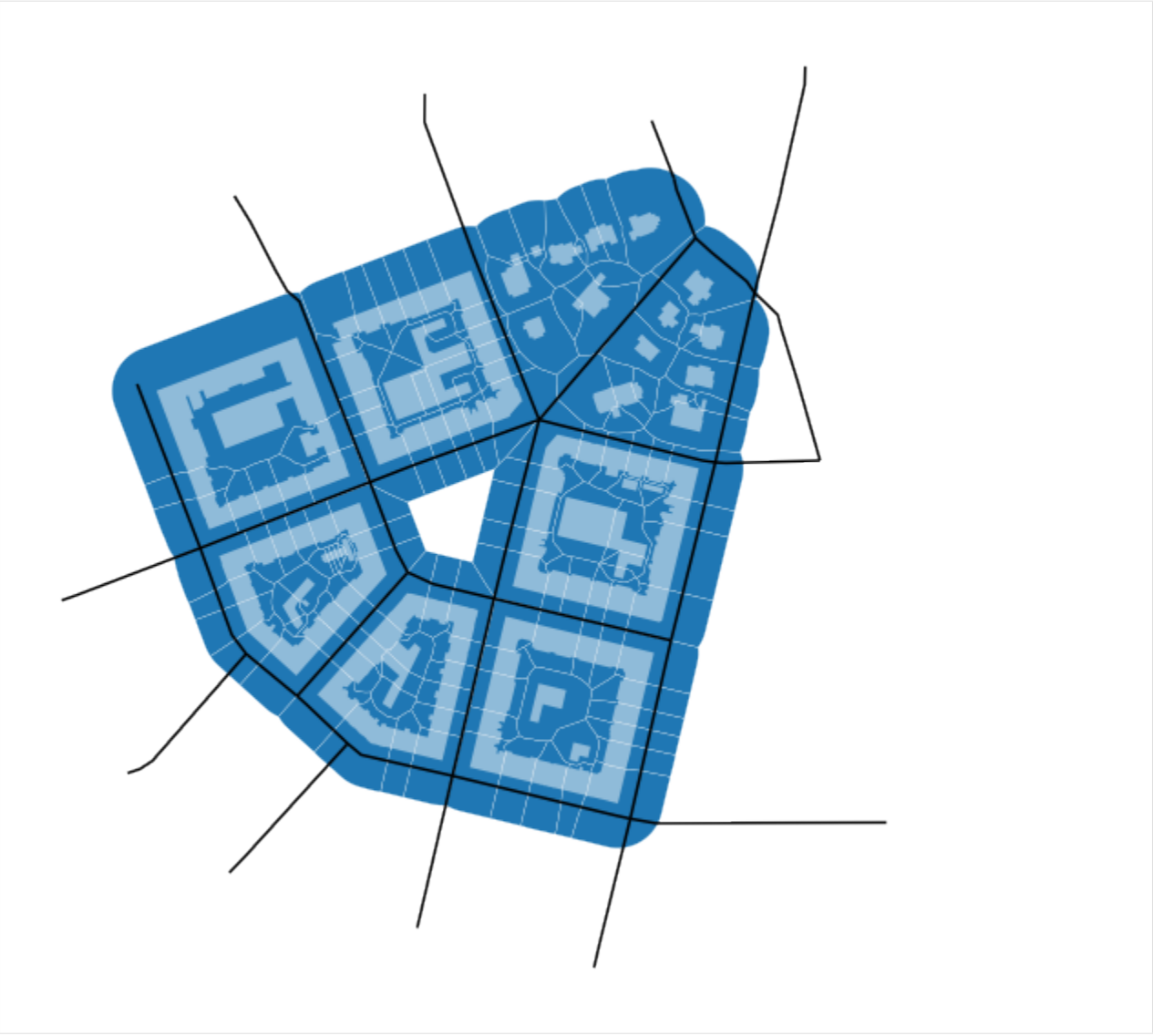
```
ax.set_axis_off()
plt.show()
```



Finally, to cover the last of the essential elements, we import the street network.

[16]:
```
streets = gpd.read_file(momepy.datasets.get_path('bubenec'),
                        layer='streets')
```

[17]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, edgecolor='white', linewidth=0.2)
buildings.plot(ax=ax, color='white', alpha=.5)
streets.plot(ax=ax, color='black')
ax.set_axis_off()
plt.show()
```



Street profile

`momepy.StreetProfile` captures the relations between the segments of the street network and buildings.

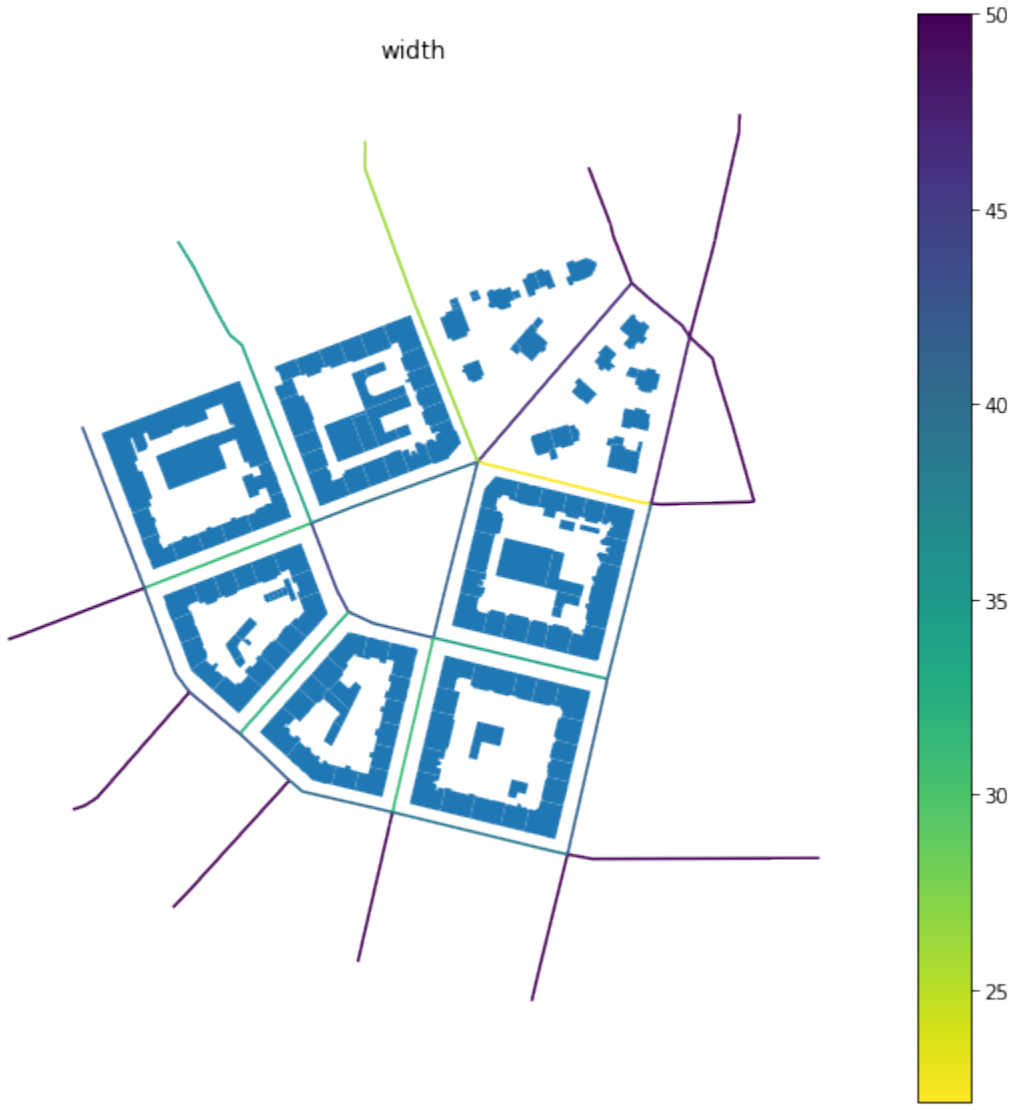
[18]:
```
profile = momepy.StreetProfile(streets, buildings)
```
```
100%|| 35/35 [00:01<00:00, 27.62it/s]
```

We have now captured multiple characters of the street profile. As we did not specify building height (we do not know it for our data), we have widths (mean) - `profile.w`, standard deviation of widths (along the segment) - `profile.wd`, and the degree of openness - `profile.o`.

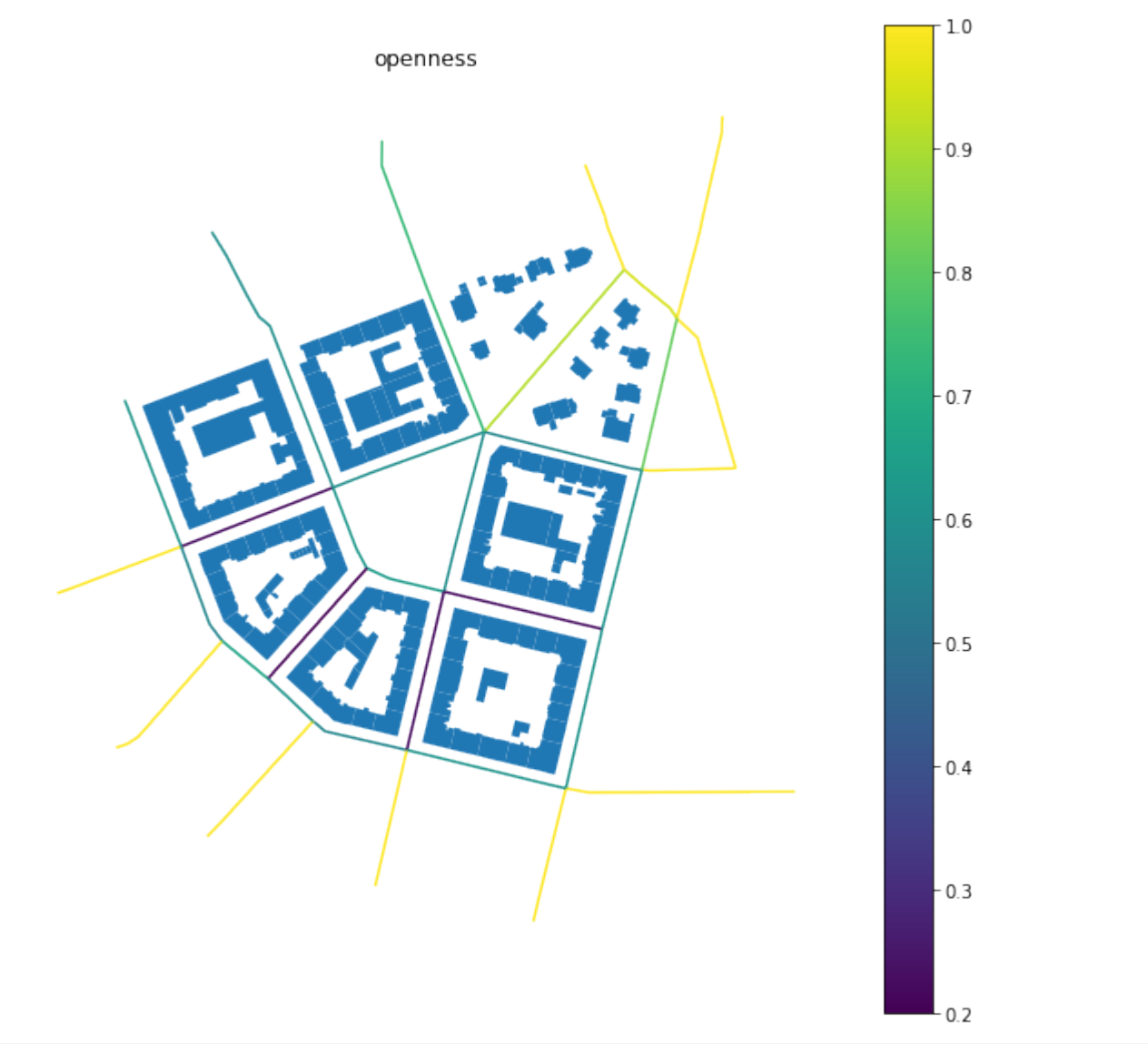
Note: `StreetProfile` measures several characters at the same time. Each of them is saved in its own Series, so `.series` would not work here.

[19]:
```
streets['width'] = profile.w
streets['openness'] = profile.o
```

[20]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax)
streets.plot('width', ax=ax, legend=True, cmap='viridis_r')
ax.set_axis_off()
ax.set_title('width')
plt.show()
```



[21]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax)
streets.plot('openness', ax=ax, legend=True)
ax.set_axis_off()
ax.set_title('openness')
plt.show()
```



Using OpenStreetMap data

In some cases (based on the completeness of OSM), we can use OpenStreetMap data without the need to save them to file and read them via GeoPandas. We can use OSMnx to retrieve them directly. In this example, we will download the building footprint of Kahla in Germany and project it to projected CRS. Momepy expects that all GeoDataFrames have the same (projected) CRS.

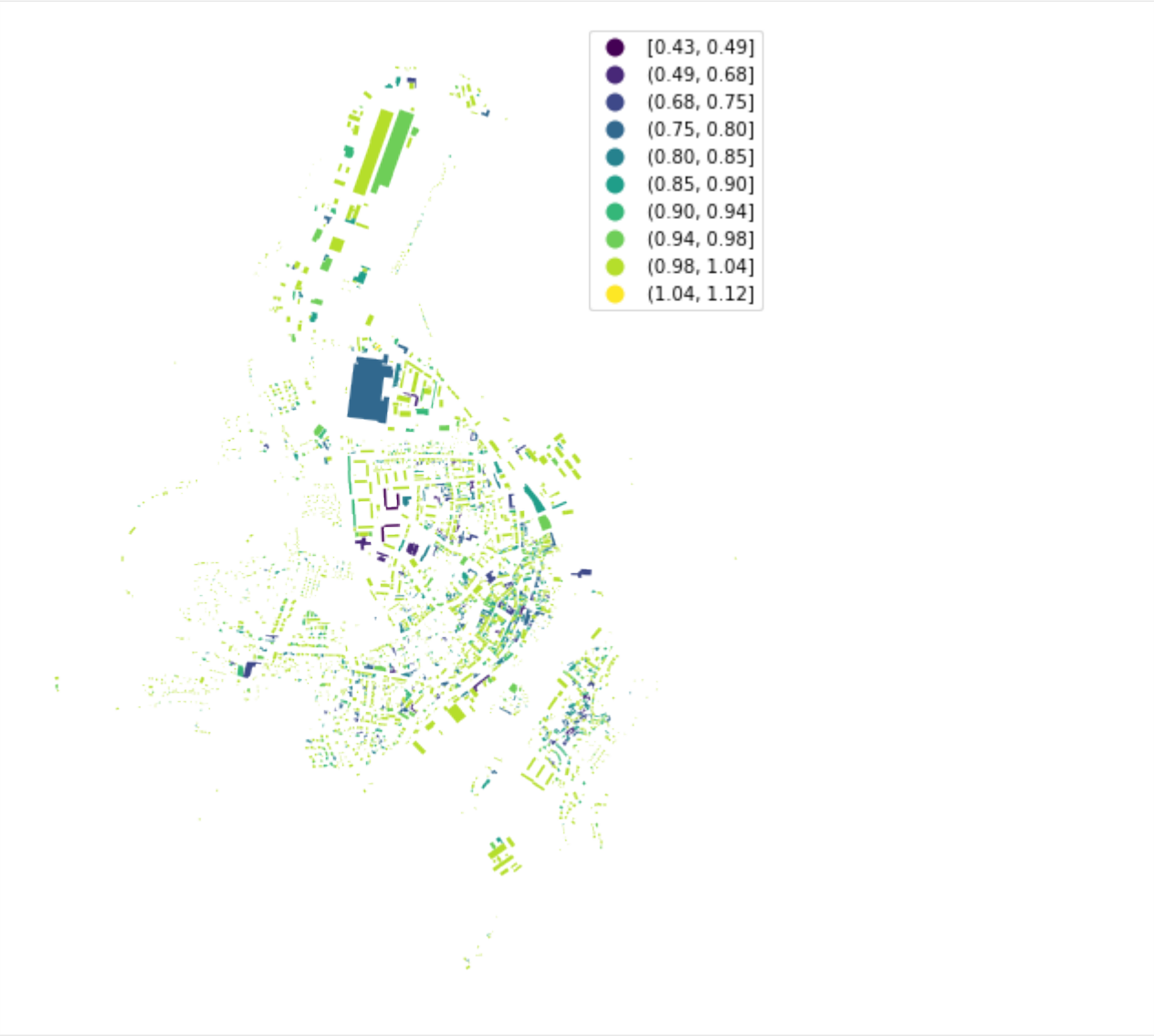
[22]:
```
import osmnx as ox

gdf = ox.footprints.footprints_from_place(place='Kahla, Germany')
gdf_projected = ox.projection.project_gdf(gdf)
```

[23]:
```
f, ax = plt.subplots(figsize=(10, 10))
gdf_projected.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

Now we are in the same situation as we were above, and we can start our morphometric analysis as illustrated on the equivalent rectangular index below.

[24]:
```
gdf_ERI = momepy.EquivalentRectangularIndex(gdf_projected)
gdf_projected['eri'] = gdf_ERI.series
```

[25]:
```
f, ax = plt.subplots(figsize=(10, 10))
gdf_projected.plot('eri', ax=ax, legend=True, scheme='NaturalBreaks', k=10)
ax.set_axis_off()
plt.show()
```



Next steps

Now we have basic knowledge of what momepy is and how it works. It is time to *install* momepy (if you haven't done so yet) and browse the rest of this user guide to see more examples. Once done, head to the *API reference* to see the full extent of characters momepy can capture and find those you need for your research.

8.2.2 Data Structure

Momepy is built on top of geopandas `GeoDataFrame` objects and, for network analysis, on networkx `Graph`.

For any kind of morphometric analysis, data needs to be provided as `GeoDataFrames`. Results of morphometric analysis from `momepy` can be generally returned as pandas `Series` to be added as a column of existing `GeoDataFrame`. All the detailes and attributes of each class are clearly described in the *API*.

Morphometric functions

Morphometric functions available in `momepy` could be divided into four different groups based on their approach to data requirements and outputs.

1. Simple characters

   Simple morphometric characters are using single `GeoDataFrame` as a source of the data.

2. Relational characters

   Relational characters are based on relations between two or more `GeoDataFrames`. Typical example is `AreaRatio`, which requires a) features to be covered (e.g. land unit) and b) features which are covering them (e.g. buildings).

3. Network analysis

   Network analysis (`graph` module) characters are based on `networkx.Graph` and returns `networkx.Graph` with additional node or edge attributes.

Morphological elements

Additional modules (`elements` and `utils`) cover functions generating new morphological elements (like morphological tessellation) or links between them. For details, please refer to the *API*.

Majority of functions used within `momepy` is not limited to one type of morphological elements. However, the whole package is built with a specific set of elements in mind, based on the research done at the University of Strathclyde by the Urban Design Studies Unit. This is true especially for morphological tessellation, partitioning of space based on building footprints. Morphological tessellation can substitute plots for certain types of analysis and provide additional information, like the adjacency, for the other. More information on tessellation is in dedicated *section* of this guide.

Generally, we can work with any kind of morphological element which fits selected function, there is no restriction. Sometimes, where documentation refers to buildings, other elements like blocks can be used as well as long as the principle remains the same.

For example, you can use `momepy` to do morphometric analysis of:

- buildings,
- plots,
- morphological cells,
- streets,
  - profiles,
  - networks,
- blocks,

and more.

Links between elements

When using more than one morphological element, `momepy` needs to understand what is the relationship between them. For this, it relies on `unique_id` attributes. It is expected, that every building lies on certain plot or morphological cell, on certain street or within certain block. To use `momepy`, each feature of each layer needs its own `unique_id`. Moreover, each feature also needs to bear `unique_id` of related elements. Consider following sample rows of `buildings_gdf`:

building_id	block_id	network_edge_id
1	143	22
2	143	25
3	144	25
4	144	25
5	144	29

Each building has its own unique `building_id`, while more building share `block_id` of block they belong to. In this sense, in `blocks_gdf` each feature would have its own unique `block_id` used as a reference for `buildings_gdf`. In principle, elements on the smaller scale contains IDs of elements on the larger - blocks will not have building IDs.

Momepy can generate unique ID using `momepy.unique_id()` and *link certain types of elements together*

Spatial weights

Unique IDs are also used as an ID within spatial weights matrices. Thanks to this, spatial weights generated on morphological tessellation (like Queen contiguity) can be directly used on buildings and vice versa. Detailed information on using spatial weights within momepy will be *discussed later*.

8.2.3 Generating morphological elements

Apart from morphometric analysis, momepy can generate certain kinds of morphological elements and link different elements together via unique_id (see *Data structure*). The main feature of `elements` module is an algorithm generating **morphological tessellation** and consequent one generating blocks based on geometry of morphological tessellation.

This section covers:

Morphological tessellation

One of the main features of `momepy` is the ability to generate and analyse morphological tessellation (MT). One can imagine MT like Voronoi tessellation generated around building polygons instead of points. The similarity is not accidental - the core of MT is a Voronoi diagram generated by `scipy.spatial.Voronoi`. We'll explain key parts of tessellation and explore its application in the real world.

Using exemplary data

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

[2]:
```
buildings = gpd.read_file(momepy.datasets.get_path('bubenec'),
                          layer='buildings')
```

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

To generate MT, each building needs to have an `unique_id` assigned, which will later link generated cell to its parent building. Exemplar `GeoDataFrame` comes with such ID in `uID` column.

[4]:
```
buildings.head()
```

[4]:
```
   uID                                               geometry
0    1  POLYGON ((1603599.221 6464369.816, 1603602.984...
1    2  POLYGON ((1603042.880 6464261.498, 1603038.961...
2    3  POLYGON ((1603044.650 6464178.035, 1603049.192...
3    4  POLYGON ((1603036.557 6464141.467, 1603036.969...
4    5  POLYGON ((1603082.387 6464142.022, 1603081.574...
```

As Voronoi tessellation tends to go to infinity for edge points, we have to define a limit for tessellation. It can be the area of your case study represented as a Polygon or MultiPolygon or you can use `momepy.buffered_limit` to generate such limit as a set maximal distance from buildings.

[5]:
```
limit = momepy.buffered_limit(buildings, buffer=100)
limit
```

[5]:

Other crucial attributes of tessellation algorithm are `segment` and `shrink`. Both are predefined as balanced values between the computational demands and a quality of a result. Segment defines the maximal distance between points generated to represent building footprint, shrink defines how much should be building buffered (inwards) to generate a gap between adjacent polygons. If you want to reduce memory requirements, you can use larger `segment` distance, but it may cause imprecision.

[6]:
```
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit)
```

```
Inward offset...
Discretization...
100%|| 144/144 [00:00<00:00, 899.36it/s]
Generating input point array...
Generating Voronoi diagram...

Vertices to Polygons:  17%|        | 5678/32914 [00:00<00:00, 56773.15it/s]
Generating GeoDataFrame...
Vertices to Polygons: 100%|| 32914/32914 [00:00<00:00, 46928.01it/s]
Dissolving Voronoi polygons...
0it [00:00, ?it/s]
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...

```

[7]:
```
tessellation
```

[7]:
```
<momepy.elements.Tessellation at 0x7f8098e0a970>
```

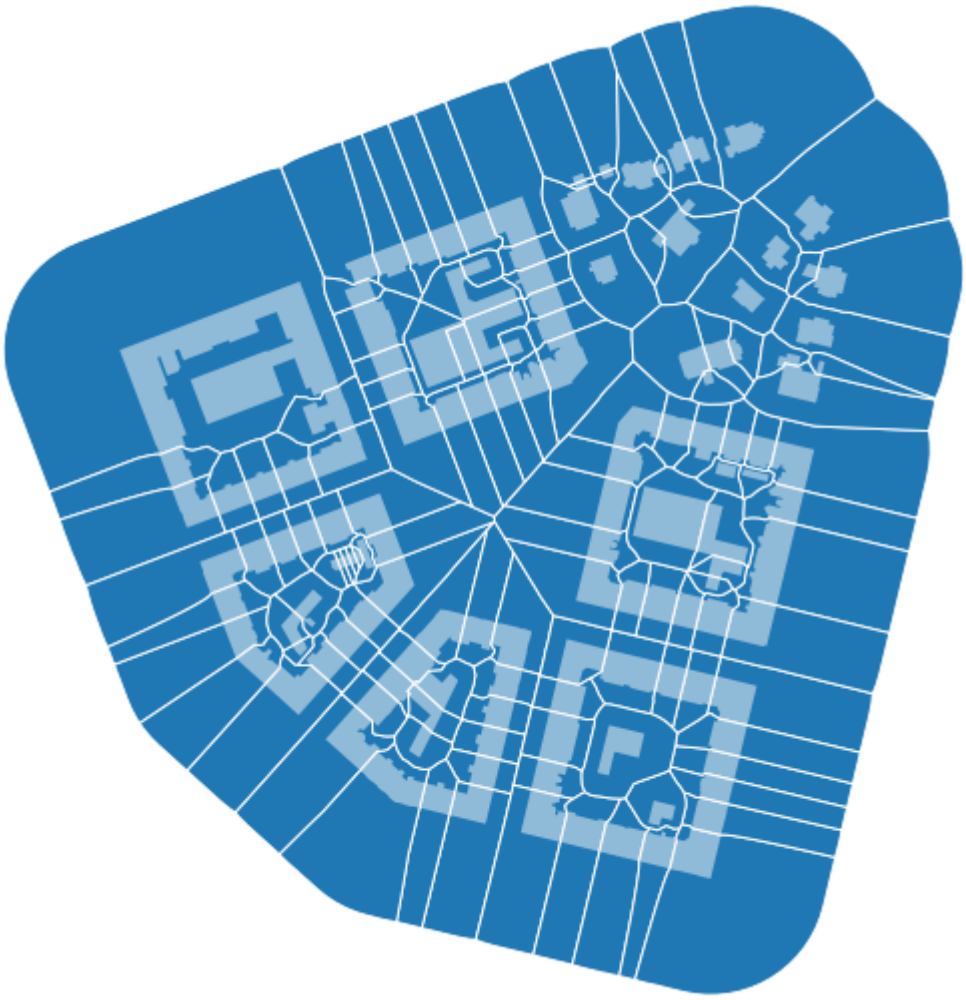
GeoDataFrame containing the tessellation itself can be accessed using `tessellation` attribute. Similarly, used values and input geometry can be accessed using `shrink`, `segment`, `limit` of `gdf` attributes.

[8]:
```
tessellation_gdf = tessellation.tessellation
```

[9]:
```
tessellation.shrink
```

[9]:
```
0.4
```

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation_gdf.plot(ax=ax, edgecolor='white')
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```



Generated tessellation can be linked to buildings using `unique_id`, being the common column between both `GeoDataFrames`.

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

[11]:

	uID	geometry
0	1	POLYGON ((1603586.677 6464344.668, 1603578.491...
1	2	POLYGON ((1603013.702 6464169.324, 1603009.972...
2	3	POLYGON ((1602931.388 6464138.882, 1603006.941...
3	4	POLYGON ((1603055.834 6464093.615, 1602963.025...
4	5	POLYGON ((1603083.773 6464104.212, 1603083.766...

Generating tessellation based on OpenStreetmap

To illustrate a more real-life example, let's try to generate tessellation based on a small town retrieved from OSM. We will use `osmnx` package to get the data.

[12]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)
```

[13]:
```
f, ax = plt.subplots(figsize=(10, 10))
gdf_projected.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

While working with real-life data, we often face issues with their quality. To avoid some of the possible errors, we should preprocess (clean) the data. It is often done semi-manually within the GIS environment. `momepy` offers (experimental) `momepy.preprocess` to handle some of the expected issues.

[14]:
```
buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=0.2, islands=True)
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 2932/2932 [00:00<00:00, 24412.02it/s]
Changing geometry: 100%|| 31/31 [00:00<00:00, 118.10it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 2520/2520 [00:00<00:00, 27371.02it/s]
Changing geometry: 100%|| 2/2 [00:00<00:00, 99.65it/s]
```

What has happened?

1. All auxiliary buildings (smaller than 30 square meters defined in `size`) were dropped.
2. Possible adjacent structures of specific circular compactness values (long and narrow) were joined to their parental buildings.
3. All buildings fully within other buildings (share 100% of the exterior boundary) were joined to their parental buildings.

Tessellation requires two other arguments: unique ID and limit. We will generate unique ID using `momepy.unique_id` and limit of tessellation using the same buffer method as above.

[15]:
```
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
```

At this moment, we have everything we need to generate morphological tessellation. It might take a while for larger `GeoDataFrames`.

[16]:
```
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit)
tessellation_gdf = tessellation.tessellation
```

```
Inward offset...
Discretization...
```

```
  4%|         | 89/2521 [00:00<00:03, 649.45it/s]
```

```
Generating input point array...
```

```
100%|| 2521/2521 [00:01<00:00, 2058.04it/s]
```

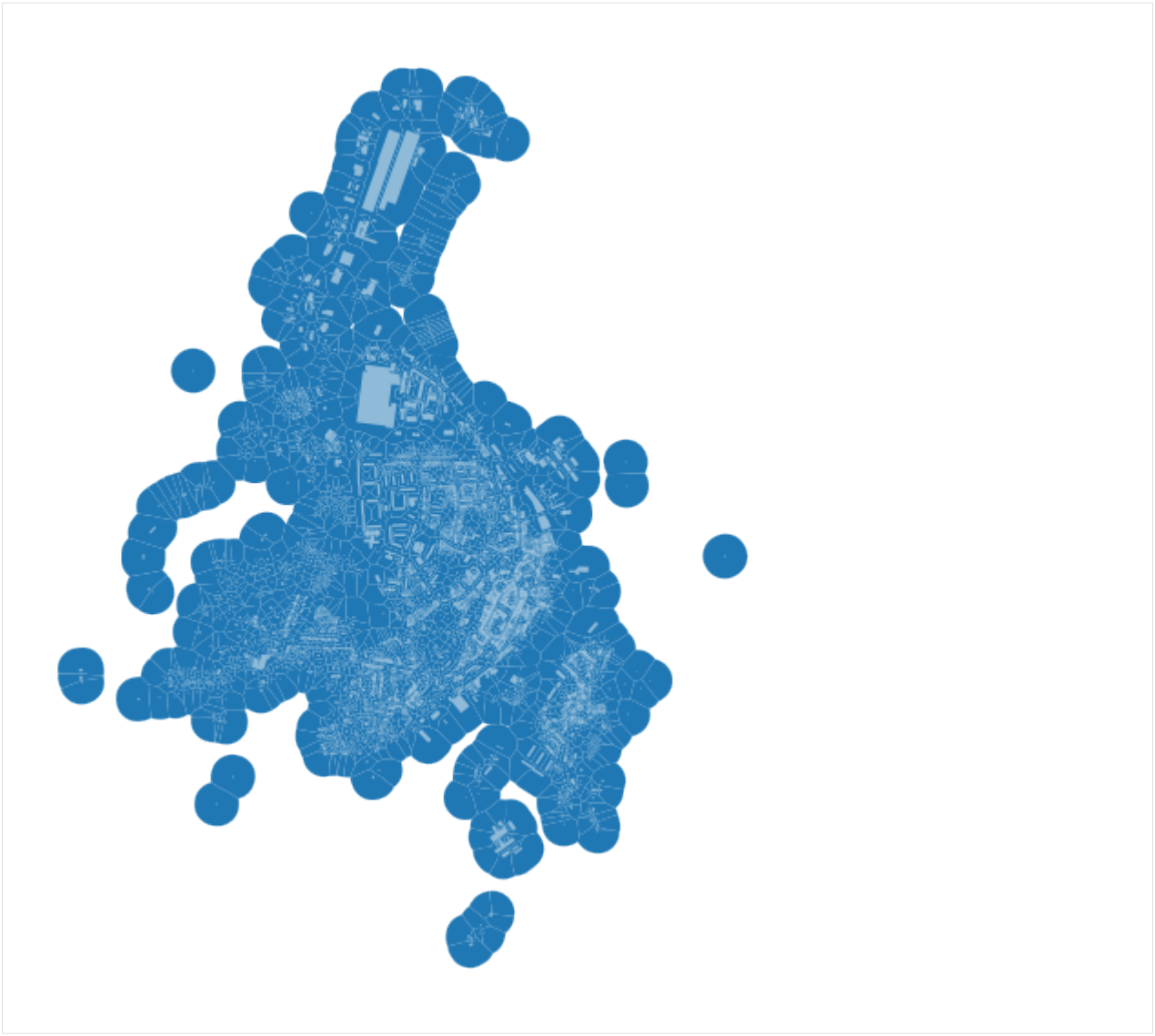
```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 267563/267563 [00:04<00:00, 57768.80it/s]
```

```
Dissolving Voronoi polygons...
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

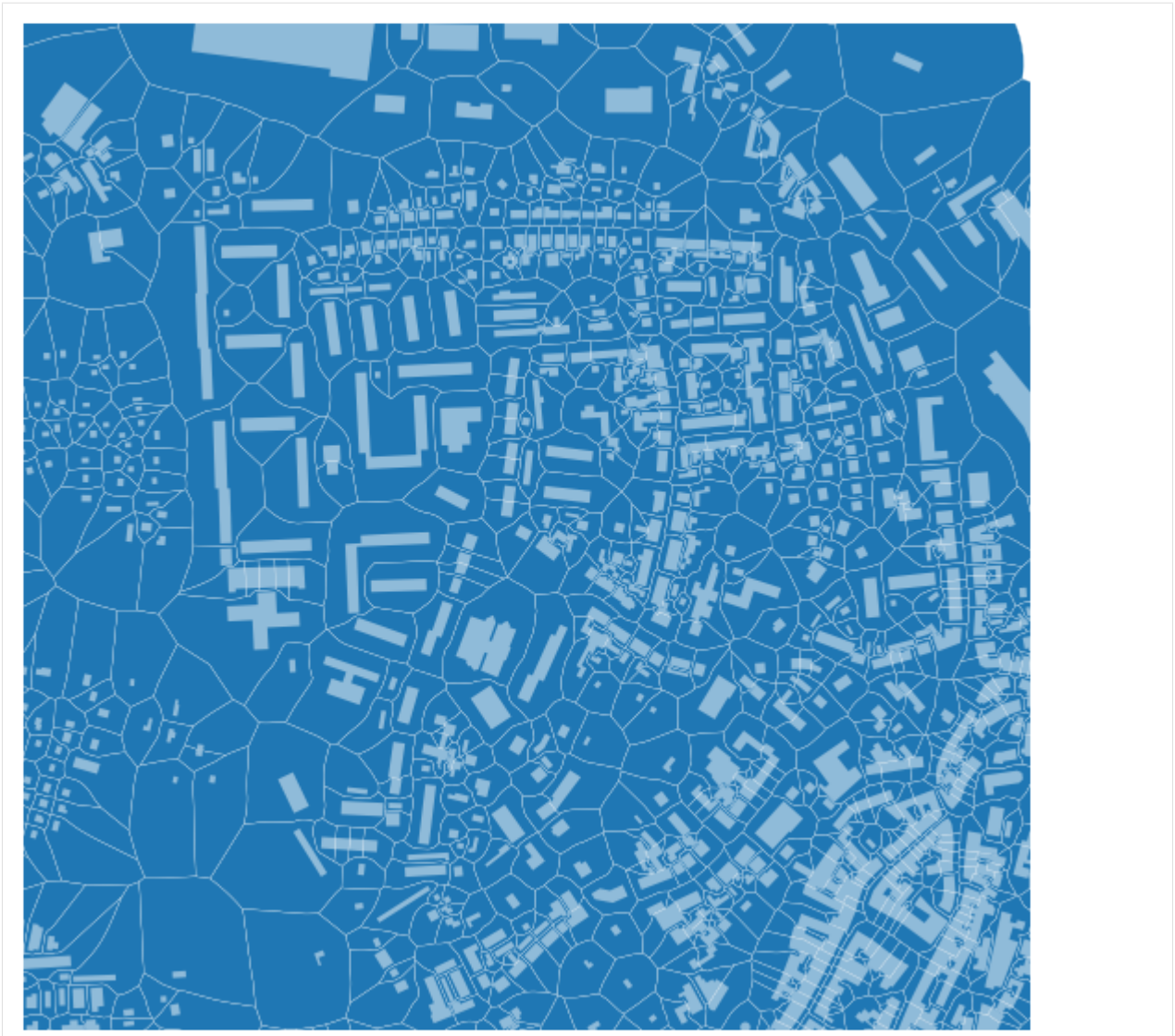
```
0it [00:00, ?it/s]
```

[17]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation_gdf.plot(ax=ax)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```



Zooming closer to check the result:

[18]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation_gdf.plot(ax=ax, edgecolor='white', linewidth=0.2)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
ax.set_xlim(681500, 682500)
ax.set_ylim(5631000, 5632000)
plt.show()
```



And we are done. Morphological tessellation is generated and ready for any further analysis.

Troubleshooting

In some cases, `momepy.Tessellation` raises errors or warnings. In 99% of cases, this is due to errors in input data. Two types of warnings are possible:

- `Tessellation does not fully match buildings. 10 element(s) collapsed "during generation - unique_id: [list of ids]`

In this case, some of the building shapes collapsed during the shrinkage. It should not happen as shrink distance is usually quite small, but you might be able to resolve it by setting smaller `shrink` distance. However, we would recommend fixing the data manually.

- `Tessellation contains MultiPolygon elements. Initial objects should be edited. unique_id of affected elements: [list of ids]`

This is a more common issue, which is again caused by imprecise data. Often caused by long and extremely narrow

shapes or overlap of buildings. While some of the analysis might work even with MultiPolygon geometry, it does not really make sense, so we would recommend fixing the data beforehand.

However, for most of the data of higher quality, you should not see any of these warnings.

Enclosed tessellation

Enclosed tessellation is an enhanced *morphological tessellation*, based on predefined enclosures and building footprints. We can see enclosed tessellation as two-step partitioning of space based on building footprints and boundaries (e.g. street network, railway). Original morphological tessellation is used under the hood to partition each enclosure.

Note

Enclosed tessellation has been developed as a part of Urban Grammar AI research project which is publicly available and provides an example of real-world application of the concept.

In this notebook, we will look at the concept of *enclosures* behind enclosed tessellation, generate tessellation itself and compare it to a simpler morphological tessellation.

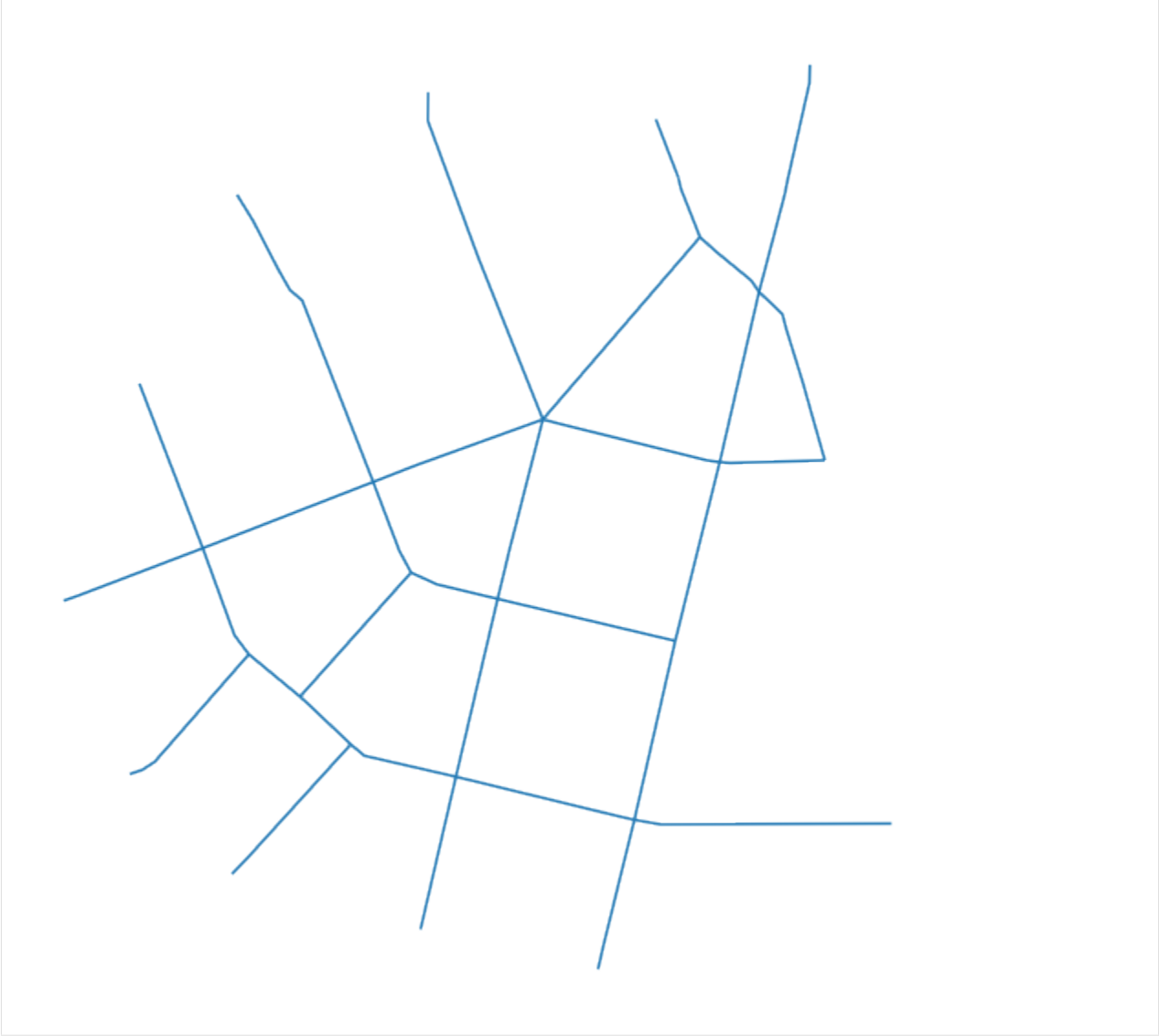
Enclosures

Enclosures are areas enclosed from all sides by at least one type of a barrier. Barriers are typically roads, railways, natural features like rivers and other water bodies or a coastline. In our example, we will work with roads, and illustrate the behaviour of additional barriers using artificial data.

[1]:
```
import momepy
import geopandas as gpd
```

[2]:
```
streets = gpd.read_file(momepy.datasets.get_path('bubenec'),
                        layer='streets')
```

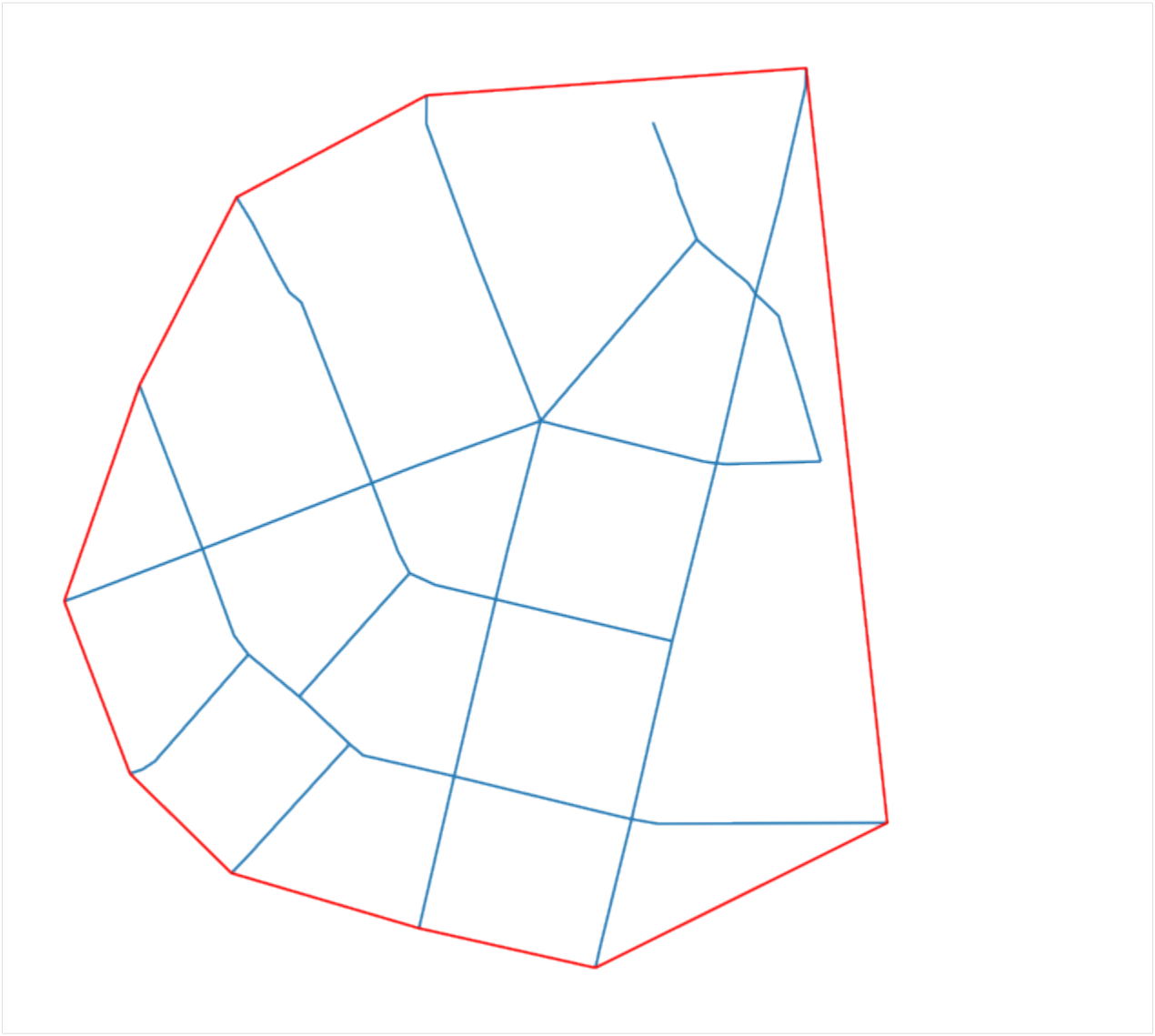
[4]:
```
streets.plot(figsize=(10, 10)).set_axis_off()
```



It is optimal (although not necessary) to specify the external boundary of the area for which we want to generate enclosures. In this case, we use a convex hull around our street network. In the case of islands, a typical limit is a coastline, but in most of the situations, it will the boundary of the case study area.

[5]:
```
convex_hull = streets.unary_union.convex_hull
```

[6]:
```
ax = streets.plot(figsize=(10, 10))
gpd.GeoSeries([convex_hull.boundary]).plot(ax=ax, color='r')
ax.set_axis_off()
```

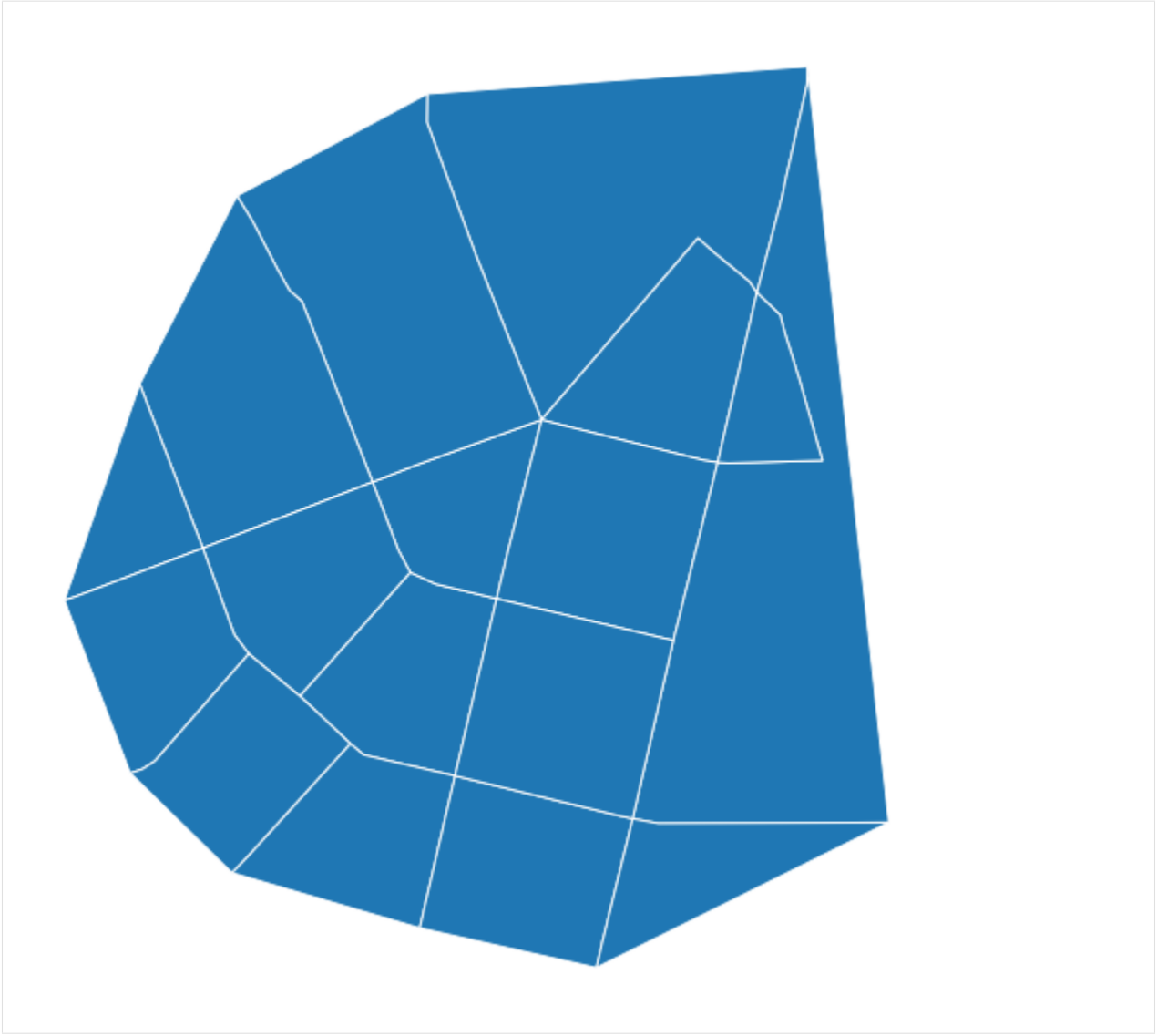


The `momepy.enclosures` function requires `limit` as geopandas.GeoSeries or GeoDataFrame and can contain multiple objects.

Generating enclosures is then straightforward:

[7]:
```
enclosures = momepy.enclosures(streets, limit=gpd.GeoSeries([convex_hull]))
```

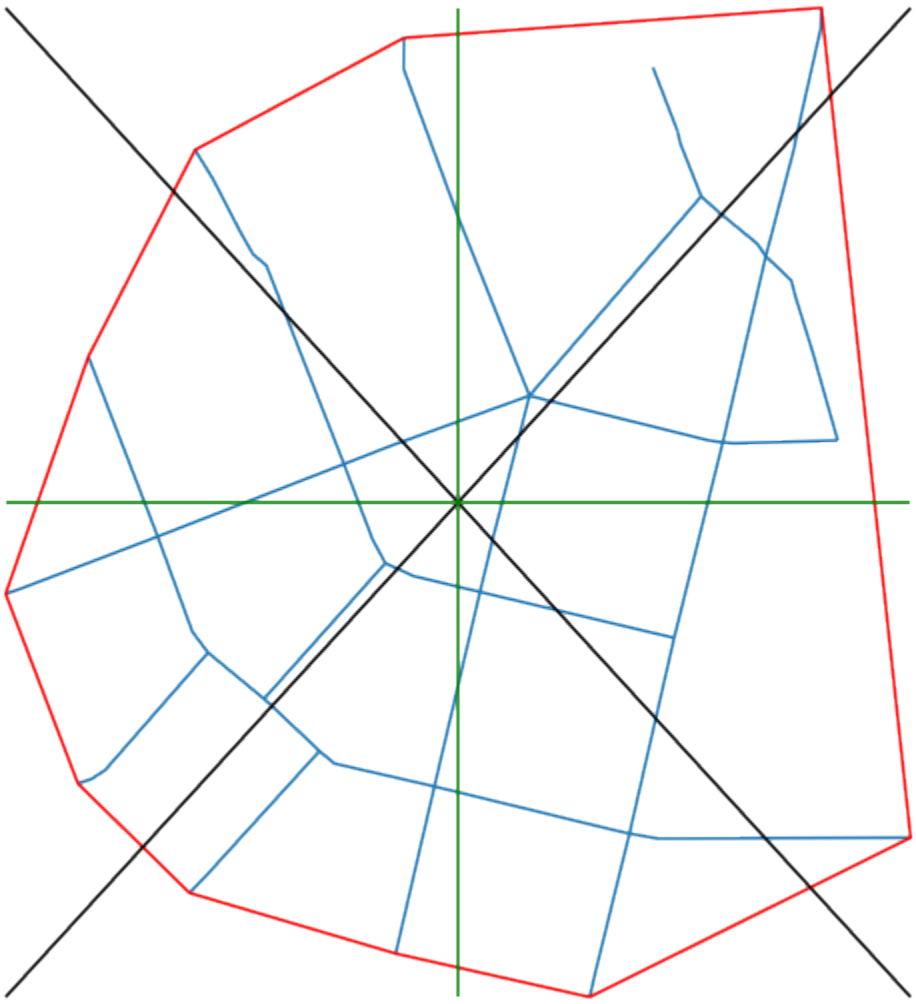
[17]:
```
enclosures.plot(figsize=(10, 10), edgecolor='w').set_axis_off()
```



The resulting enclosures are a result of polygonization of the input. However, if there are `additional_barriers`, polygons above are further subdivided. Let's now pretend that two diagonals represent the railway as an additional barrier, and one horizontal line represents a river:

[10]:
```
import numpy as np
from shapely.geometry import LineString

b = convex_hull.bounds
railway = gpd.GeoSeries(
    [LineString([(b[0], b[1]), (b[2], b[3])]), LineString([(b[0], b[3]), (b[2],␣
↪b[1])])]
)
rivers = gpd.GeoSeries(
    [
        LineString([(b[0], np.mean([b[1], b[3]])), (b[2], np.mean([b[1], b[3]]))]),
        LineString([(np.mean([b[0], b[2]]), b[1]), (np.mean([b[0], b[2]]), b[3])]),
    ]
)
```

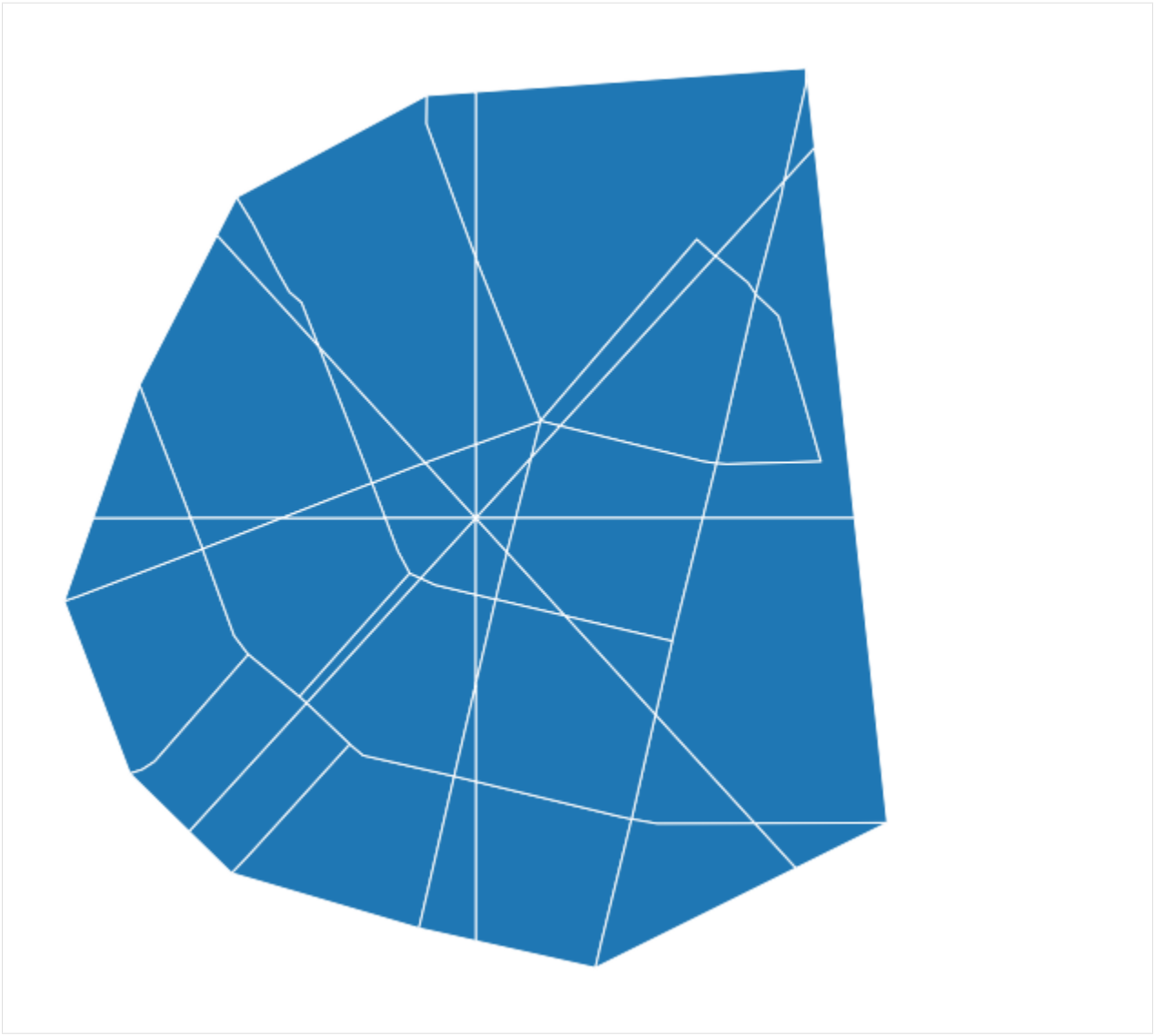
[11]:
```
ax = streets.plot(figsize=(10, 10))
gpd.GeoSeries([convex_hull.boundary]).plot(ax=ax, color='r')
railway.plot(ax=ax, color='k')
rivers.plot(ax=ax, color='g')
ax.set_axis_off()
```



Enclosures are now defined using all the barriers.

[12]:
```
enclosures_additional = momepy.enclosures(
    streets, limit=gpd.GeoSeries([convex_hull]), additional_barriers=[railway, rivers]
)
```

[16]:
```
enclosures_additional.plot(figsize=(10, 10), edgecolor='w').set_axis_off()
```



Enclosed tessellation

Having enclosures, we can now use enclosed tessellation. That, in principle, applies morphological tessellation to each enclosure using it as its limit.

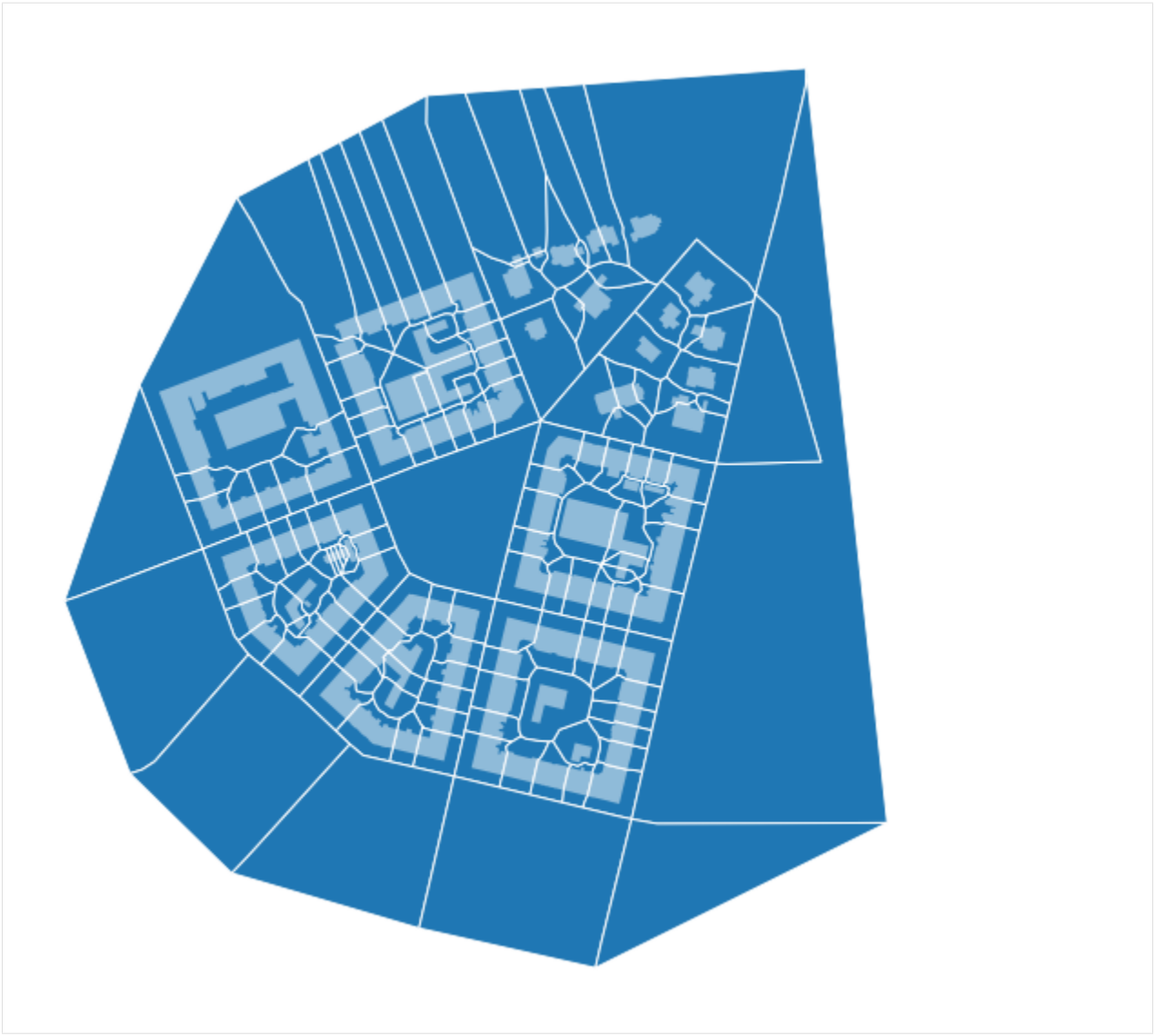
For clarity, we will use the simpler enclosures we generated above.

[19]:
```
buildings = gpd.read_file(momepy.datasets.get_path('bubenec'),
                          layer='buildings')
```

[20]:
```
buildings.plot(figsize=(10, 10)).set_axis_off()
```

[26]:
```
enclosed_tess = momepy.Tessellation(buildings, unique_id='uID',␣
↪enclosures=enclosures).tessellation
```

[27]:
```
ax = enclosed_tess.plot(edgecolor='white', figsize=(10, 10))
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
```

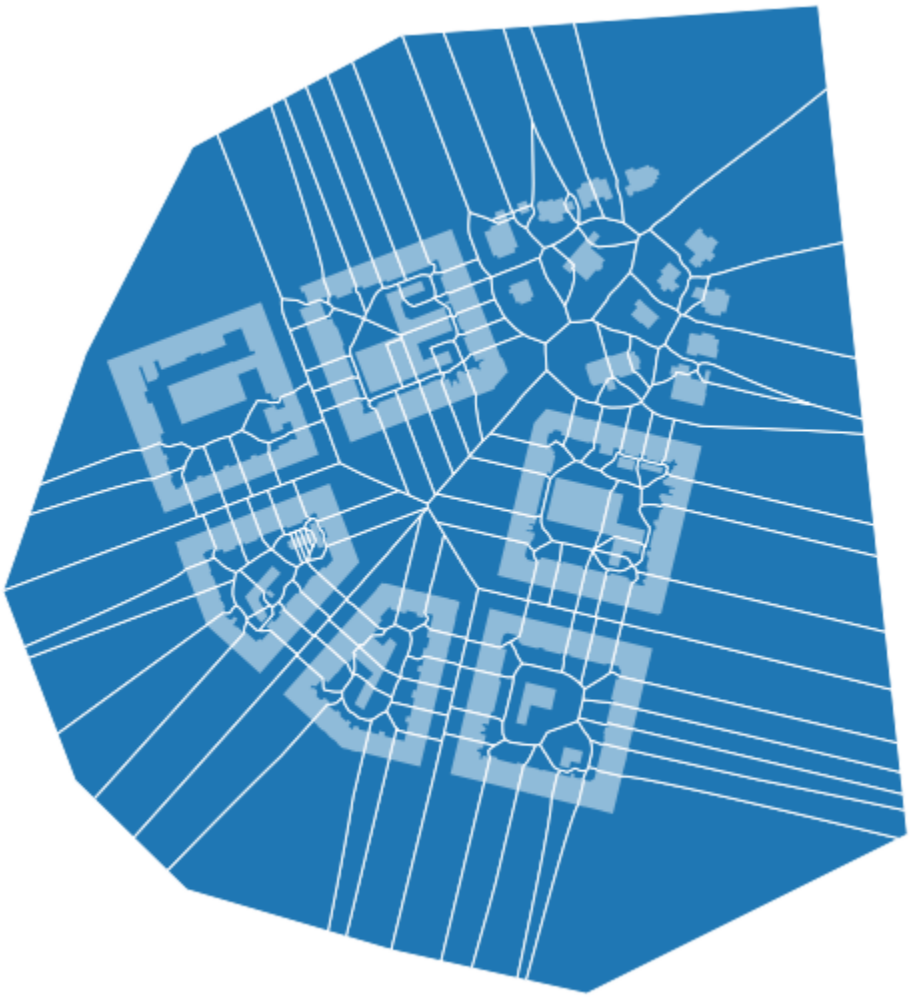


Comparison to morphological tessellation

Let's not see how enclosed tessellation differs from morphological tessellation. First, we generate morphological tessellation within the same limit.

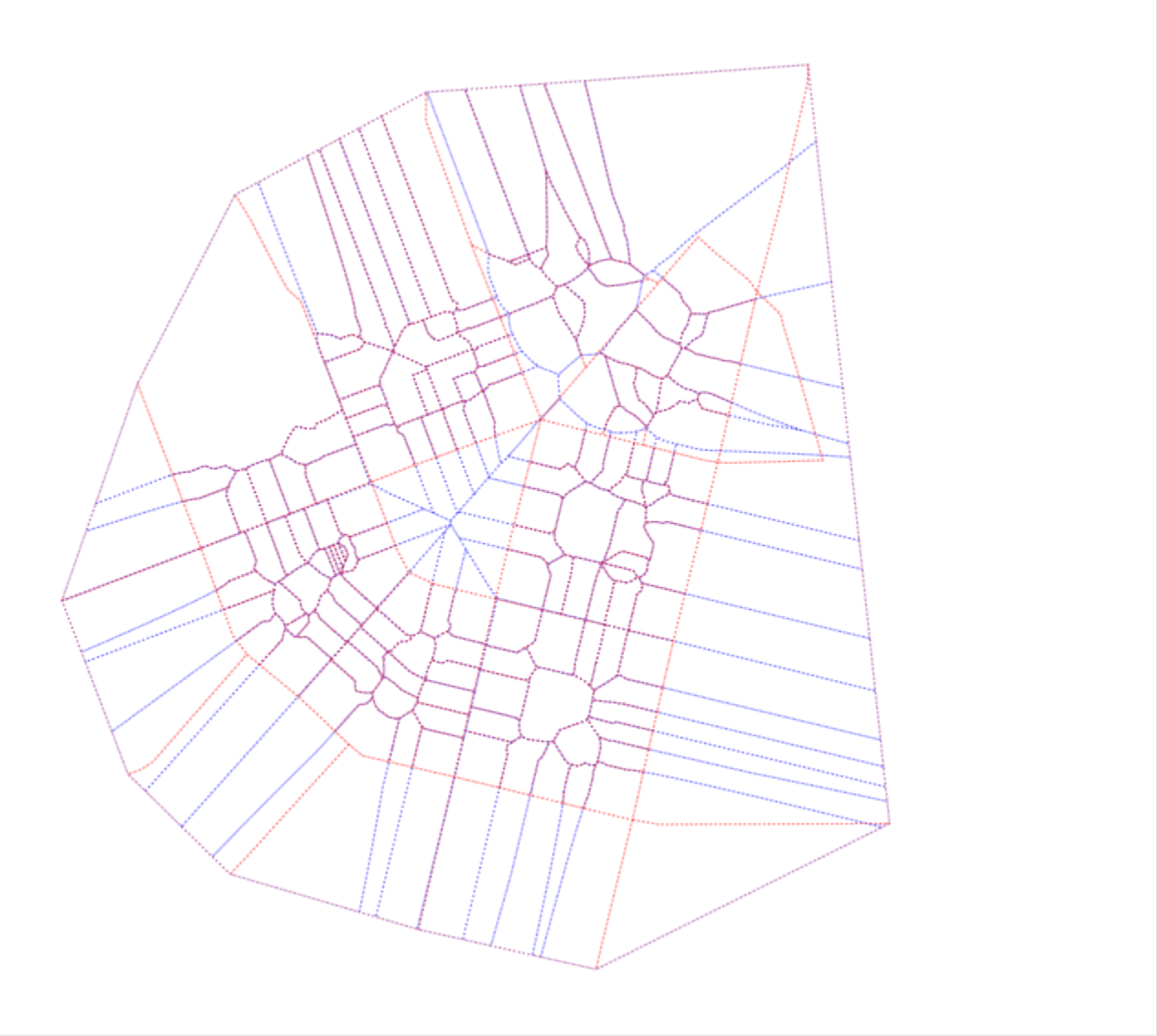
[28]:
```
morphological_tess = momepy.Tessellation(buildings, unique_id='uID', limit=convex_
↪hull, verbose=False).tessellation
```

[29]:
```
ax = morphological_tess.plot(edgecolor='white', figsize=(10, 10))
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
```



We can immediately see that the enclosed tessellation is tidier and resembles plots. We can overlay both for a direct comparison.

[30]:
```
ax = morphological_tess.plot(edgecolor='blue', facecolor='none', linestyle='dotted',␣
↪alpha=.5, figsize=(10, 10))
enclosed_tess.plot(ax=ax, edgecolor='red', facecolor='none', linestyle='dotted',␣
↪alpha=.5)
ax.set_axis_off()
```



From this figure, we can see that a large portion of geometry overlaps, but there are apparent differences when it comes to open spaces.

Performance

Enclosed tessellation usually is much faster and less demanding algorithm than morphological tessellation. Furthermore, it is by default parallelised using `dask`. If you do not have `dask` installed in your environment or do not want to use it, you can set `use_dask=False` to make a simple loop instead.

[32]:
```
enclosed_tess = momepy.Tessellation(buildings, unique_id='uID', enclosures=enclosures,
↪ use_dask=False)
```

Enclosed tessellation based on OpenStretMap

To illustrate a more real-life example, let's try to generate tessellation based on a small town retrieved from OSM. We will use `osmnx` package to get the data.

[33]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)
buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=0.2, islands=True, verbose=False)

streets_graph = ox.graph_from_place('Kahla, Germany', network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
streets = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                           node_geometry=False, fill_edge_geometry=True)
```

[34]:
```
ax = buildings.plot(figsize=(10, 10))
streets.plot(ax=ax, color='k', linewidth=.5)
ax.set_axis_off()
```

Enclosures

We will generate enclosures based on the street network and limit of using the buffer method.

[35]:
```
limit = momepy.buffered_limit(buildings)
enclosures = momepy.enclosures(streets, limit=gpd.GeoSeries([limit]))
```

`Tessellation` requires `unique_id` to match resulting cells to original buildings. We will generate a unique ID using `momepy.unique_id`.

[36]:
```
buildings['uID'] = momepy.unique_id(buildings)
```

At this moment, we have everything we need to generate enclosed tessellation.

Enclosed tessellation

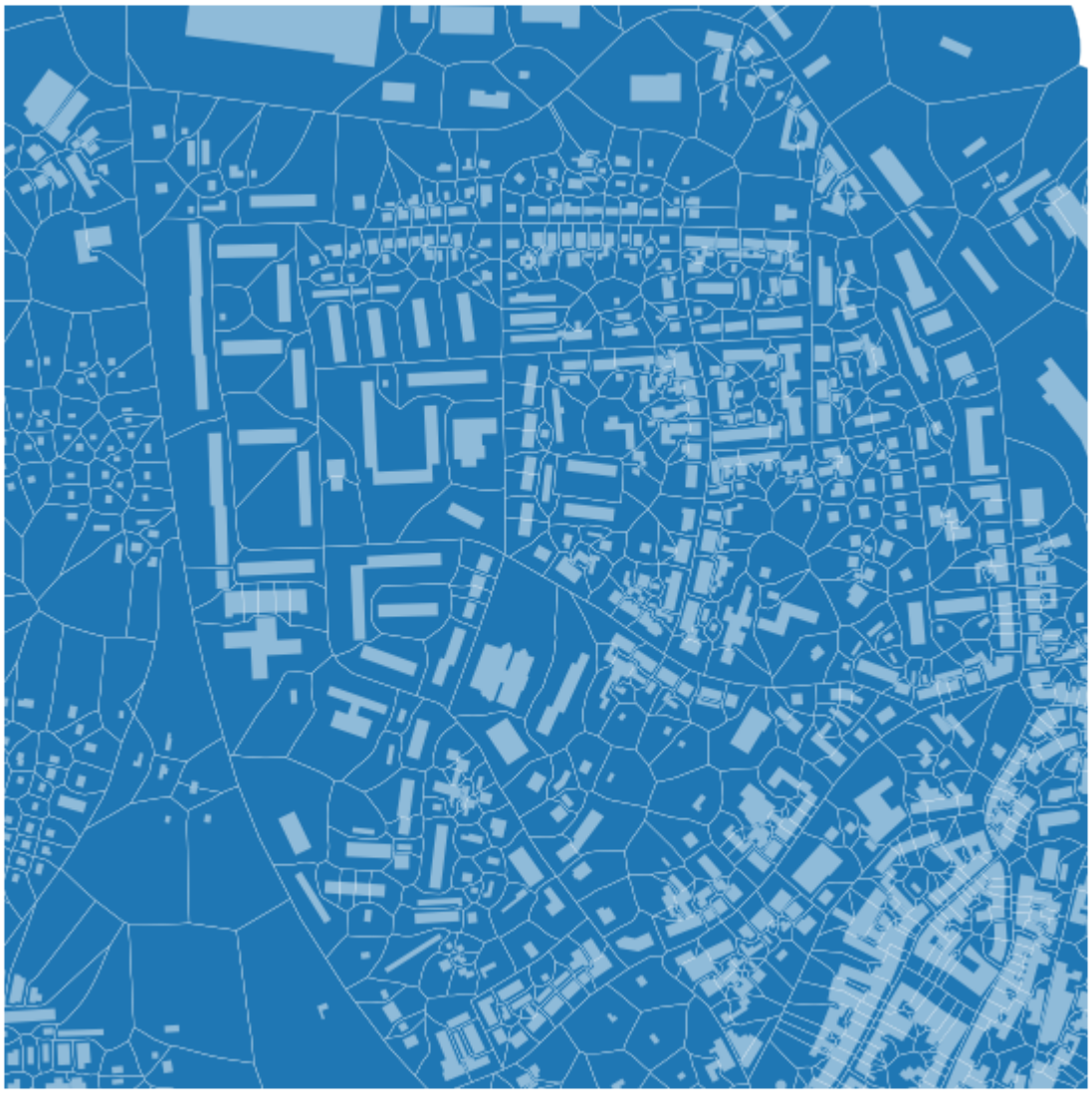
[37]:
```
enclosed_tess = momepy.Tessellation(buildings, unique_id='uID',␣
↪enclosures=enclosures).tessellation
```

[38]:
```
ax = enclosed_tess.plot(figsize=(10, 10))
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
```

Zooming closer:

[39]:
```
ax = enclosed_tess.plot(edgecolor='white', linewidth=0.2, figsize=(10, 10))
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
ax.set_xlim(681500, 682500)
ax.set_ylim(5631000, 5632000)
```

[39]:
```
(5631000.0, 5632000.0)
```



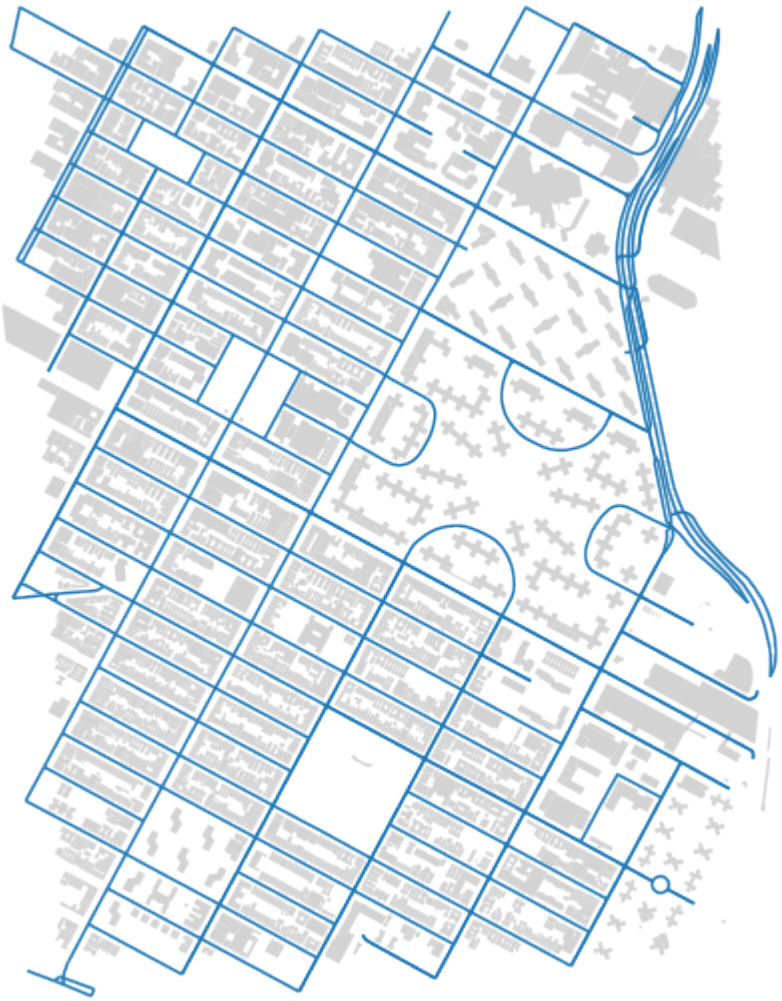
Performance comparison

Let's time parallel enclosed tessellation, loop-based option and morphological tessellation. The notebook is run using a 4-core processor in a high-end laptop.

[40]:
```
point = (40.731603, -73.977857)
dist = 1000
gdf = ox.geometries.geometries_from_point(point, dist=dist-100, tags={'building':True}
↪)
buildings = ox.projection.project_gdf(gdf)
buildings = buildings[buildings.geom_type.isin(['Polygon', 'MultiPolygon'])]
buildings['height'] = buildings['height'].fillna(0).astype(float)
buildings = buildings.explode()
buildings.reset_index(inplace=True, drop=True)
```

[41]:
```
streets_graph = ox.graph_from_point(point, dist, network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
edges = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

[42]:
```
ax = buildings.plot(color='lightgrey', figsize=(10, 10))
edges.plot(ax=ax)
ax.set_axis_off()
```



[43]:
```
limit = gpd.GeoSeries([buildings.unary_union.convex_hull])
enclosures = momepy.enclosures(edges, limit=limit)
```

[44]:
```
buildings['uID'] = momepy.unique_id(buildings)
```

[47]:
```
from time import time

times = {}
```

(continues on next page)

(continued from previous page)

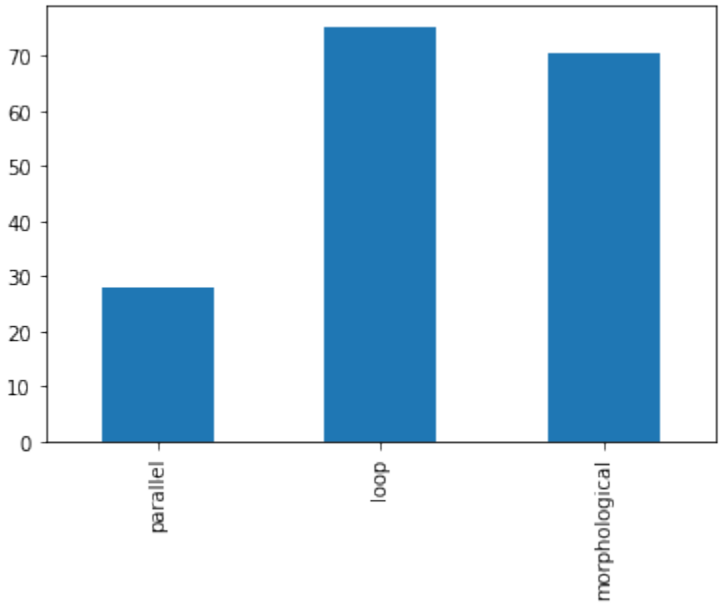
```
s = time()
%time parallel = momepy.Tessellation(buildings, unique_id='uID',␣
↪enclosures=enclosures)
times['parallel'] = time() - s

s = time()
%time loop = momepy.Tessellation(buildings, unique_id='uID', enclosures=enclosures,␣
↪use_dask=False)
times['loop'] = time() - s

s = time()
%time morphological = momepy.Tessellation(buildings, unique_id='uID', limit=limit,␣
↪verbose=False)
times['morphological'] = time() - s
```

```
CPU times: user 1min 37s, sys: 4.22 s, total: 1min 41s
Wall time: 27.8 s
CPU times: user 1min 13s, sys: 654 ms, total: 1min 14s
Wall time: 1min 15s
CPU times: user 1min 9s, sys: 787 ms, total: 1min 9s
Wall time: 1min 10s
```

```
/Users/martin/Git/momepy/momepy/elements.py:365: UserWarning: Tessellation contains␣
↪MultiPolygon elements. Initial objects should be edited. unique_id of affected␣
↪elements: [108, 217, 630, 1623, 2432, 2435, 2520, 2523, 2608, 2615, 2619]
  "unique_id of affected elements: {}".format(list(self.multipolygons))
```

[48]:
```
import pandas as pd

pd.Series(times).plot(kind='bar')
```

[48]:
```
<AxesSubplot:>
```



We can see about 2.5x speedup if we use the parallelised option on this particular machine. You can then scale the computation to larger machines or clusters.

Tessellation-based blocks

Ideal case with ideal data

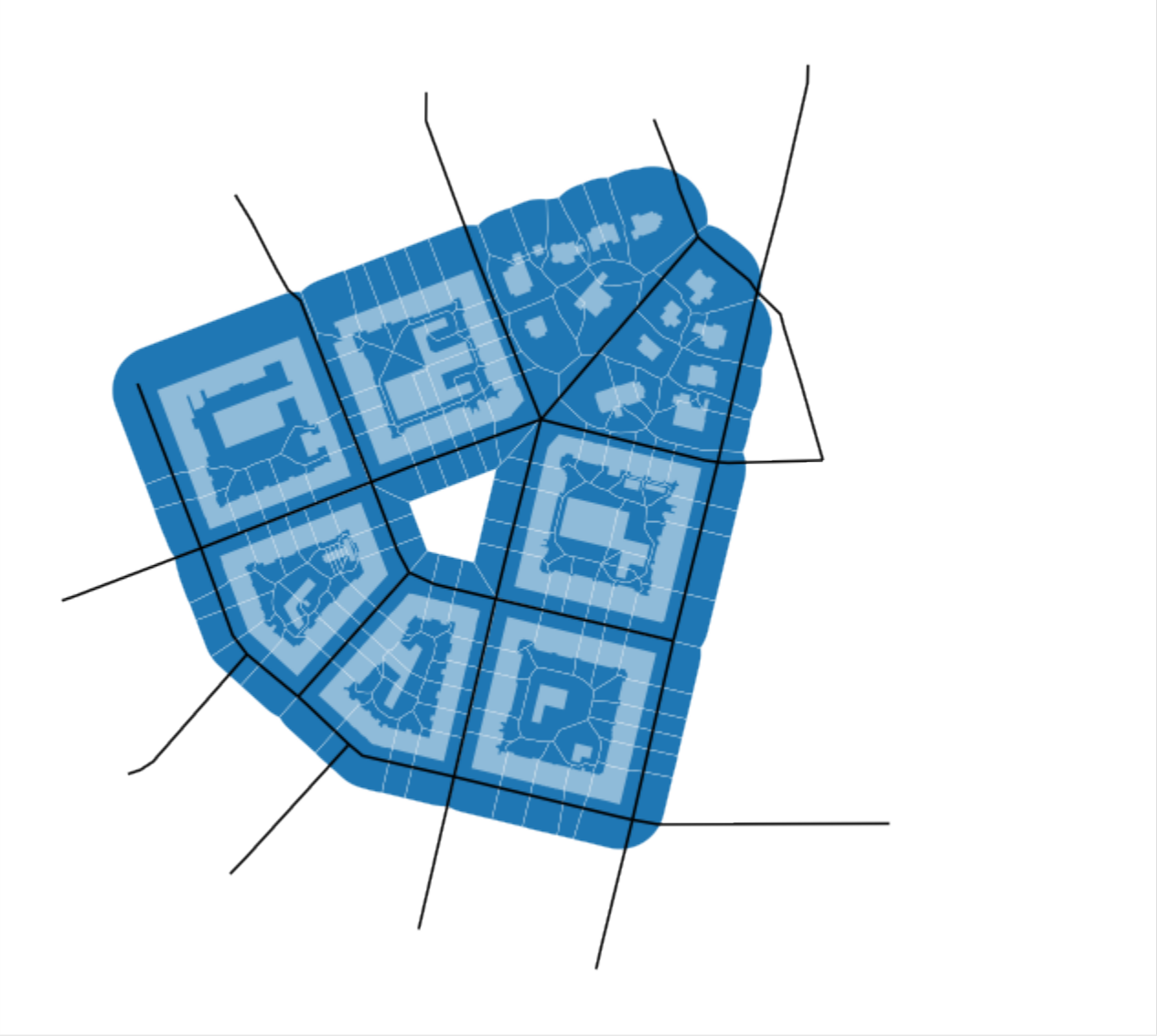
Assume following situation: We have a layer of buildings for selected city, a street network represented by centrelines, we have generated morphological tessellation, but now we would like to do some work on block scale. We have two options - generate blocks based on street network (each closed loop is a block) or use morphological tessellation and join those cells, which are expected to be part of one block. In that sense, you will get tessellation-based blocks which are following the same logic as the rest of your data and are hence fully compatible. With `momepy` you can do that using `momepy.Blocks`.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

For illustration, we can use `bubenec` dataset embedded in `momepy`.

[2]:
```
path = momepy.datasets.get_path('bubenec')
buildings = gpd.read_file(path, layer='buildings')
streets = gpd.read_file(path, layer='streets')
tessellation = gpd.read_file(path, layer='tessellation')
```

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, edgecolor='white', linewidth=0.2)
buildings.plot(ax=ax, color='white', alpha=.5)
streets.plot(ax=ax, color='black')
ax.set_axis_off()
plt.show()
```



This example is useful for illustration why pure network-based blocks are not ideal. 1. In the centre of the area would be the block, while it is clear that there is only open space. 2. Blocks on the edges of the area would be complicated, as streets do not fully enclose them.

None of it is an issue for tessellation-based blocks. `momepy.Blocks` requires buildings, tessellation, streets and IDs. We have it all, so we can give it a try.

[4]:
```
blocks = momepy.Blocks(
    tessellation, streets, buildings, id_name='bID', unique_id='uID')
```

```
 25%|       | 36/144 [00:00<00:00, 359.36it/s]
```

```
Buffering streets...
Generating spatial index...
Difference...
```

```
100%|| 144/144 [00:00<00:00, 541.63it/s]
100%|| 254/254 [00:00<00:00, 293243.38it/s]
```

```
Defining adjacency...
Defining street-based blocks...
Defining block ID...
Generating centroids...
Spatial join...
Attribute join (tesselation)...
Generating blocks...
Multipart to singlepart...
Attribute join (buildings)...
Attribute join (tesselation)...
```

GeoDataFrame containing blocks can be accessed using `blocks`. Moreover, block ID for buildings and tessellation can be accessed using `buildings_id` and `tessellation_id`.

[5]:
```
blocks_gdf = blocks.blocks
buildings['bID'] = blocks.buildings_id
tessellation['bID'] = blocks.tessellation_id
```

[6]:
```
f, ax = plt.subplots(figsize=(10, 10))
blocks_gdf.plot(ax=ax, edgecolor='white', linewidth=0.5)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

Fixing the street network

The example above shows how it works in the ideal case - streets are attached, there are no gaps. However, that is often not true. For that reason, `momepy` includes `momepy.snap_street_network_edge` utility which is supposed to fix the issue. It can do two types of network adaptation:

1. **Snap network onto the network where false dead-end is present.** It will extend the last segment by set distance and snap it onto the first reached segment.
2. **Snap network onto the edge of tessellation.** As we need to be able to define blocks until the very edge of tessellation, in some cases, it is necessary to extend the segment and snap it to the edge of the tessellated area.

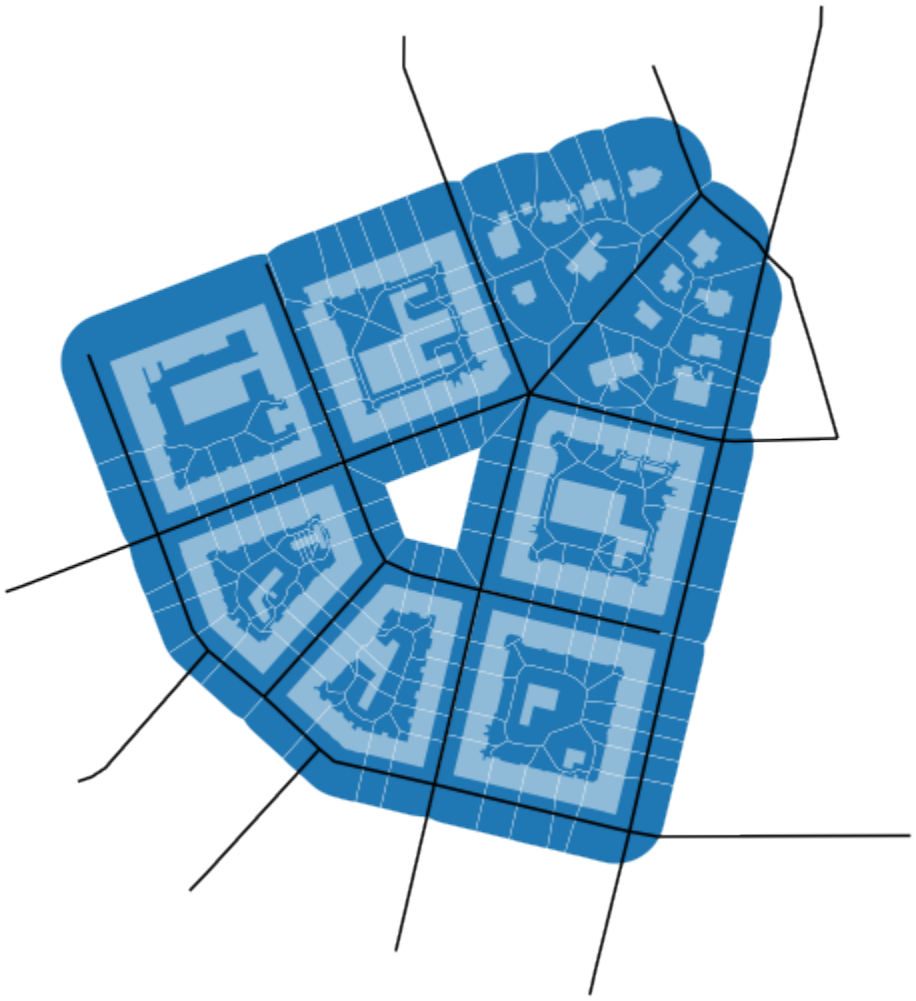
Let's adapt our ideal street network and break it to illustrate the issue.

[7]:
```
import shapely
geom = streets.iloc[33].geometry
splitted = shapely.ops.split(geom, shapely.geometry.Point(geom.coords[5]))
```

(continues on next page)

(continued from previous page)

```
streets.loc[33, 'geometry'] = splitted[1]
geom = streets.iloc[19].geometry
splitted = shapely.ops.split(geom, geom.representative_point())
streets.loc[19, 'geometry'] = splitted[0]

f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, edgecolor='white', linewidth=0.2)
buildings.plot(ax=ax, color='white', alpha=.5)
streets.plot(ax=ax, color='black')
ax.set_axis_off()
plt.show()
```



In this example, one of the lines on the right side is not snapped onto the network and other on the top-left leaves the gap between the edge of the tessellation. Let's see what would be the result of blocks without any adaptation of this network.

[8]:
```
buildings.drop(['bID'], axis=1, inplace=True)
tessellation.drop(['bID'], axis=1, inplace=True)
```

[9]:
```
blocks = momepy.Blocks(tessellation, streets, buildings,
                       id_name='bID', unique_id='uID').blocks
```

```
100%|| 144/144 [00:00<00:00, 723.85it/s]
```

```
Buffering streets...
Generating spatial index...
Difference...
Defining adjacency...
```

```
100%|| 252/252 [00:00<00:00, 312989.22it/s]
```

```
Defining street-based blocks...
Defining block ID...
Generating centroids...
Spatial join...
Attribute join (tesselation)...
Generating blocks...
Multipart to singlepart...
Attribute join (buildings)...
Attribute join (tesselation)...
```

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
blocks.plot(ax=ax, edgecolor='white', linewidth=0.5)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

We can see that blocks are merged. To avoid this effect, we will first use `momepy.snap_street_network_edge` and then generate blocks.

[11]:
```
snapped = momepy.snap_street_network_edge(streets, buildings, tolerance_street=15,
                                          tessellation=tessellation,
                                          tolerance_edge=40)
```

```
Snapping: 100%|| 35/35 [00:00<00:00, 270.63it/s]
```

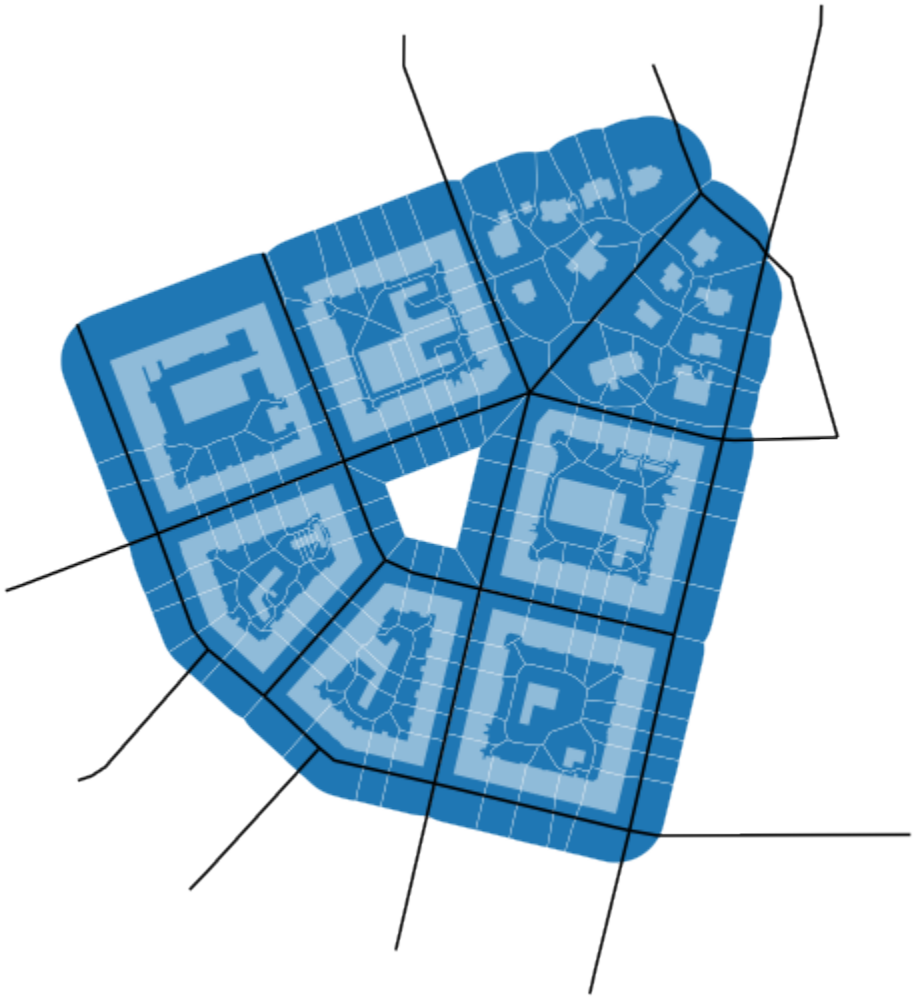
```
Building spatial index for network...
Building spatial index for buildings...
Dissolving tesselation...
```

[12]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, edgecolor='white', linewidth=0.2)
buildings.plot(ax=ax, color='white', alpha=.5)
snapped.plot(ax=ax, color='black')
```

(continues on next page)

(continued from previous page)

```
ax.set_axis_off()
plt.show()
```



What should be snapped is now snapped. We might have noticed that we had to give buildings to the `momepy.snap_street_network_edge`. That is to avoid extending segments through buildings.

With a fixed network, we can then generate correct blocks.

[13]:
```
blocks = momepy.Blocks(
    tessellation, snapped, buildings, id_name='bID', unique_id='uID').blocks
```

```
 98%|| 141/144 [00:00<00:00, 647.70it/s]
```

```
Buffering streets...
Generating spatial index...
Difference...
```

```
100%|| 144/144 [00:00<00:00, 705.41it/s]
100%|| 255/255 [00:00<00:00, 504979.94it/s]
```

```
Defining adjacency...
Defining street-based blocks...
Defining block ID...
Generating centroids...
Spatial join...
Attribute join (tesselation)...
Generating blocks...
Multipart to singlepart...
Attribute join (buildings)...
Attribute join (tesselation)...
```

[14]:
```
f, ax = plt.subplots(figsize=(10, 10))
blocks.plot(ax=ax, edgecolor='white', linewidth=0.5)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

Linking elements together

As explained in the Data Structure chapter, `momepy` relies on links between different morphological elements. Each element needs ID, and each of the small-scale elements also needs to know the ID of the relevant higher-scale element. The case of block ID is explained in the previous chapter, `momepy.Blocks` generates it together with blocks gdf.

Getting the ID of the street network

This notebook will explore how to link street network, both nodes and edges, to buildings and tessellation.

Edges

For linking street network edges to buildings (or tessellation or other elements), `momepy` offers `momepy.get_network_id`. It simply returns a `Series` of network IDs for analysed gdf.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

For illustration, we can use `bubenec` dataset embedded in `momepy`.

[2]:
```
path = momepy.datasets.get_path('bubenec')
buildings = gpd.read_file(path, layer='buildings')
streets = gpd.read_file(path, layer='streets')
tessellation = gpd.read_file(path, layer='tessellation')
```

First, we have to be sure that streets segments have their unique IDs.

[3]:
```
streets['nID'] = momepy.unique_id(streets)
```

Then we can link it to buildings. The only argument we might want to look at is `min_size`, which should be a value such that if you build a box centred in each building centroid with edges of size `2 * min_size`, you know a priori that at least one segment is intersected with the box. You can see it as a sort of tolerance.

[4]:
```
buildings['nID'] = momepy.get_network_id(buildings, streets,
                                         'nID', min_size=100)
```

```
Snapping: 100%|| 144/144 [00:00<00:00, 2477.79it/s]
```

```
Generating centroids...
Generating rtree...
```

[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='nID', categorical=True, cmap='tab20b')
streets.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

Note: colormap does not have enough colours, that is why everything on the top-left looks the same. It is not.

Nodes

The situation with nodes is slightly more complicated as you usually don't have or even need nodes. However, `momepy` includes some functions which are calculated on nodes (mostly in `graph` module). For that reason, we will pretend that we follow the usual workflow:

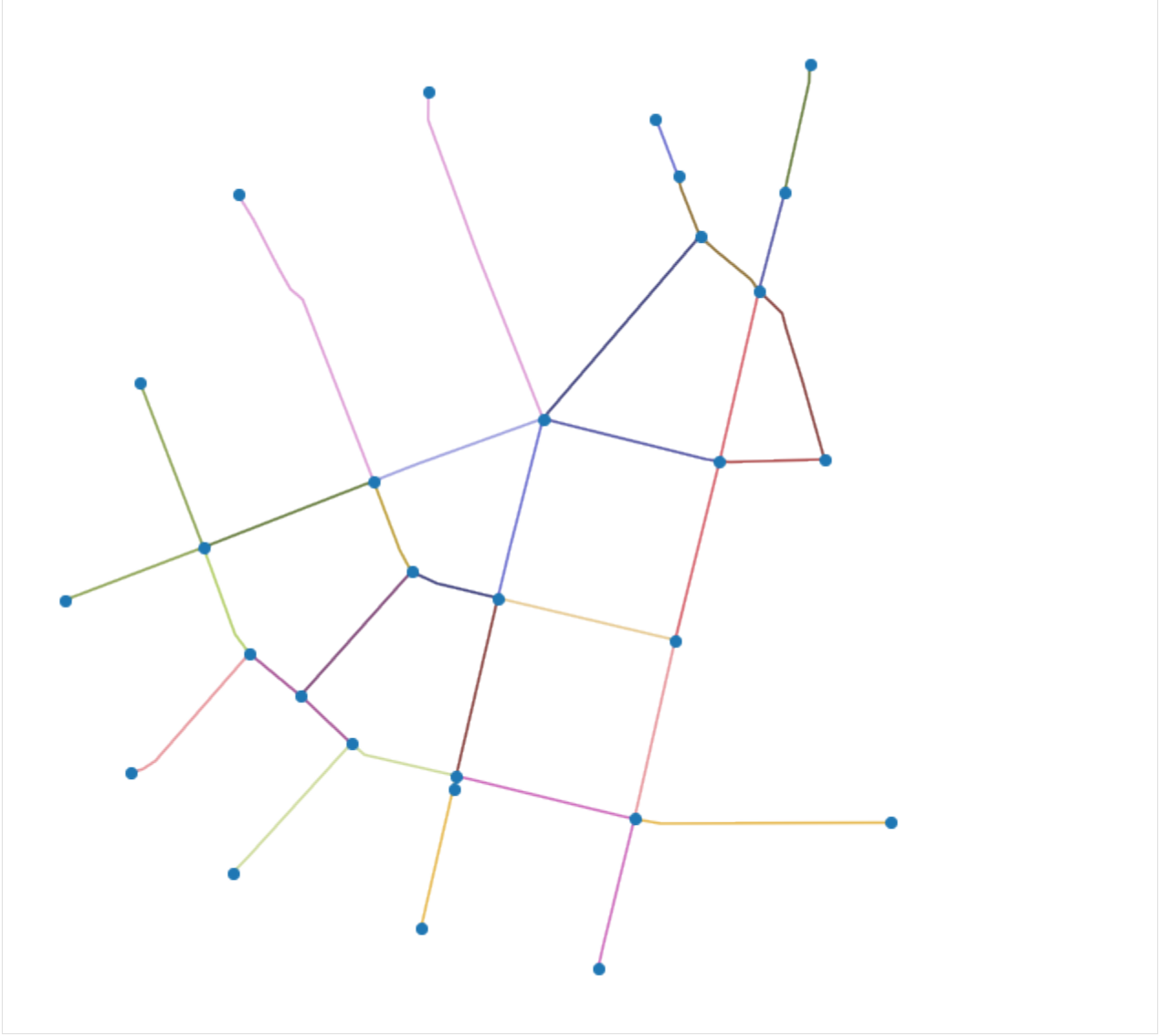
1. Street network `GeoDataFrame` (edges only)
2. networkx `Graph`
3. Street network - edges and nodes as separate `GeoDataFrames`.

[6]:
```
graph = momepy.gdf_to_nx(streets)
```

Some *graph-based analysis* happens here.

[7]:
```
nodes, edges = momepy.nx_to_gdf(graph)
```

[8]:
```
f, ax = plt.subplots(figsize=(10, 10))
edges.plot(ax=ax, column='nID', categorical=True, cmap='tab20b')
nodes.plot(ax=ax, zorder=2)
ax.set_axis_off()
plt.show()
```



For attaching node ID to buildings, we will need both, nodes and edges. We have already determined which edge building belongs to, so now we only have to find out which end of the edge is the closer one. Nodes come from `momepy.nx_to_gdf` automatically with node ID:

[9]:
```
nodes.head()
```

[9]:

	nodeID	geometry
0	1	POINT (1603585.640 6464428.774)
1	2	POINT (1603413.206 6464228.730)
2	3	POINT (1603268.502 6464060.781)

(continues on next page)

(continued from previous page)

```
3       4  POINT (1603363.558 6464031.885)
4       5  POINT (1603607.303 6464181.853)
```

The same ID is now inlcuded in edges as well, denoting each end of edge. (Length of the edge is also present as it was necessary to keep as an attribute for the graph.)

[10]:
```
edges.head()
```

[10]:
```
                                              geometry  nID      mm_len  \
0  LINESTRING (1603585.640 6464428.774, 1603413.2...    0  264.103950
1  LINESTRING (1603561.740 6464494.467, 1603564.6...   14   70.020202
2  LINESTRING (1603585.640 6464428.774, 1603603.0...   15   88.924305
3  LINESTRING (1603607.303 6464181.853, 1603592.8...    2  199.746503
4  LINESTRING (1603363.558 6464031.885, 1603376.5...    5  203.014090

   node_start  node_end
0           1         2
1           1         9
2           1         7
3           2         5
4           2         4
```

[11]:
```
buildings['nodeID'] = momepy.get_node_id(buildings, nodes, edges,
                                         'nodeID', 'nID')
```

```
100%|| 144/144 [00:00<00:00, 1740.07it/s]
```

[12]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='nodeID', categorical=True, cmap='tab20b')
nodes.plot(ax=ax, zorder=2)
edges.plot(ax=ax, zorder=1)
ax.set_axis_off()
plt.show()
```

Transfer IDs to tessellation

All IDs are now stored in buildings gdf. We can copy them to tessellation using `merge`. First, we select columns we are interested in, then we merge them with tessellation based on the shared unique ID. Usually, we will have more columns than we have now.

[13]:
```
buildings.columns
```

[13]:
```
Index(['uID', 'geometry', 'nID', 'nodeID'], dtype='object')
```

[14]:
```
columns = ['uID', 'nID', 'nodeID']
tessellation = tessellation.merge(buildings[columns], on='uID')
tessellation.head()
```

[14]:

	uID	geometry	nID	nodeID
0	1.0	POLYGON ((1603578.489 6464344.527, 1603577.040...	0	1
1	2.0	POLYGON ((1603067.112 6464177.926, 1603054.848...	33	10

(continues on next page)

(continued from previous page)

```
2  3.0  POLYGON ((1602978.618 6464156.859, 1603006.384...   10       12
3  4.0  POLYGON ((1603056.595 6464093.903, 1603011.539...   10       12
4  5.0  POLYGON ((1603110.459 6464114.367, 1603109.099...    8       12
```

Now we should be able to link all elements together as needed for all types of morphometric analysis in momepy.

8.2.4 Calculating simple characters

Simple characters are those requiring only GeoDataFrame itself. Most of the characters are not considered simple as they depend on relations between two (or more) GeoDataFrames or spatial weights matrices. But some are, and you can learn how to work with them in the notebooks below:

This section covers:

Dimension

While the majority of momepy functions require interaction of more GeoDataFrames or using spatial weights matrix, there are some which are calculated on single GeoDataFrame assessing the dimensions or shapes of features. This notebook illustrates this group on small part of Manhattan, New York.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use osmnx to get the data for our example and after preprocessing of building layer will generate tessellation. You can show the code with the button on the right side.

[11]:
```
import osmnx as ox

point = (40.731603, -73.977857)
dist = 1000
gdf = ox.geometries.geometries_from_point(point, dist=dist, tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)
gdf_projected = gdf_projected[gdf_projected.geom_type.isin(['Polygon', 'MultiPolygon
↪'])]

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tess = momepy.Tessellation(buildings, unique_id='uID', limit=limit)
tessellation = tess.tessellation
```

```
/opt/miniconda3/envs/test/lib/python3.8/site-packages/ipykernel/ipkernel.py:287:␣
↪DeprecationWarning: `should_run_async` will not call `transform_cell` automatically␣
↪in the future. Please pass the result to `transformed_cell` argument and any␣
↪exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
↪17 and above.
  and should_run_async(code)
```

```
Inward offset...
Discretization...
```

```
 10%|         | 83/836 [00:00<00:00, 824.76it/s]
```

```
Generating input point array...
```

```
100%|| 836/836 [00:01<00:00, 569.30it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 356875/356875 [00:06<00:00, 55758.92it/s]
```

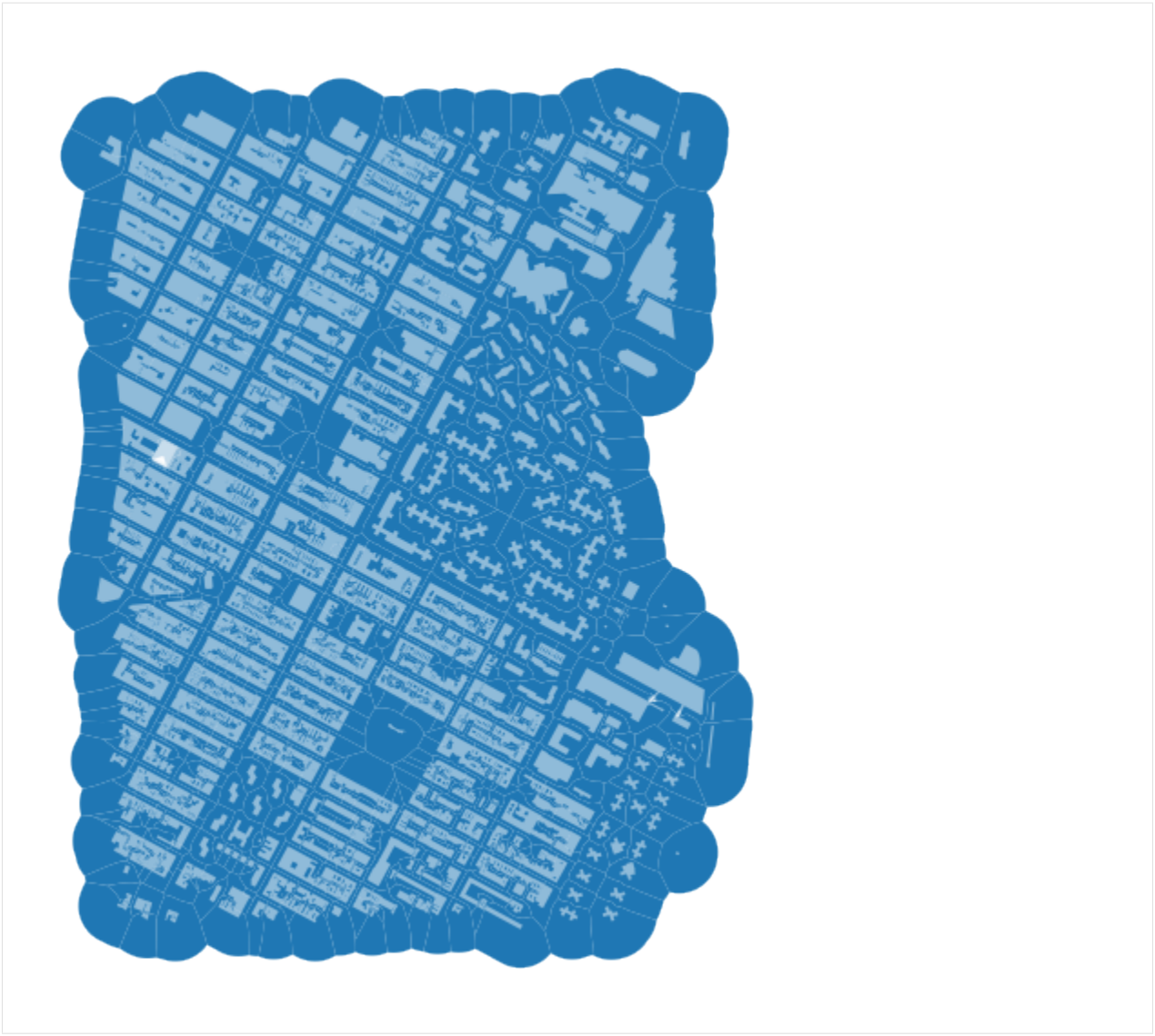
```
Dissolving Voronoi polygons...
```

```
100%|| 3/3 [00:00<00:00, 1068.43it/s]
```

```
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

[12]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

```
/opt/miniconda3/envs/test/lib/python3.8/site-packages/ipykernel/ipkernel.py:287:␣
↪DeprecationWarning: `should_run_async` will not call `transform_cell` automatically␣
↪in the future. Please pass the result to `transformed_cell` argument and any␣
↪exception that happen during thetransform in `preprocessing_exc_tuple` in IPython 7.
↪17 and above.
  and should_run_async(code)
```



We have some edge effect here as we are using the buffer as a limit for tessellation in the middle of urban fabric, but for this examples we can work with it anyway.

Area

Some work the same for more elements (buildings, tessellation, plots) like area, some makes sense only for a relevant ones. Area works for both, buildings and tessellation of our case study.

Resulting values can be accessed using `area` attribute, while original gdf using `gdf`.

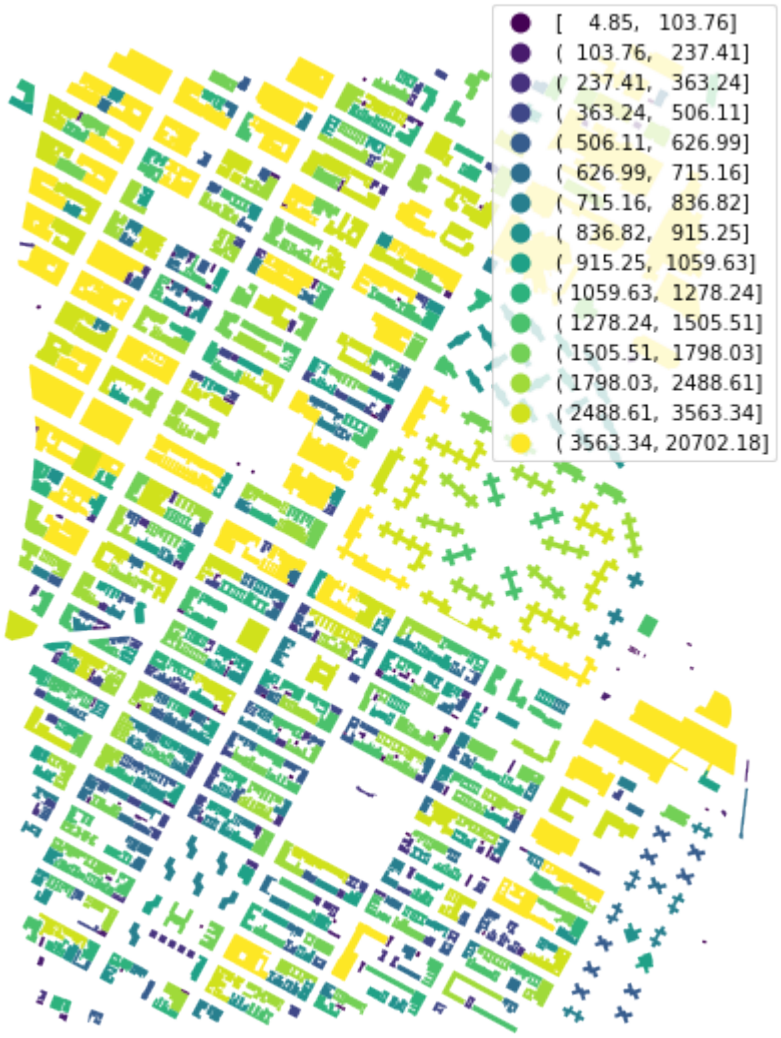
[4]:
```
blg_area = momepy.Area(buildings)
buildings['area'] = blg_area.series
```

[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='area', legend=True, scheme='quantiles', k=15, cmap=
↪'viridis')
ax.set_axis_off()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



[6]:
```
tes_area = momepy.Area(tessellation)
tessellation['area'] = tes_area.series
```

[7]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area', legend=True, scheme='quantiles', k=10, cmap=
↪'viridis')
buildings.plot(ax=ax, color='white', alpha=0.5)
ax.set_axis_off()
plt.show()
```

Height

We can also work with building heights (if we have the data). This part of New York has height data, only stored as strings, so we have to convert them to `floats` (or `int`) and fill `NaN` values with zero.

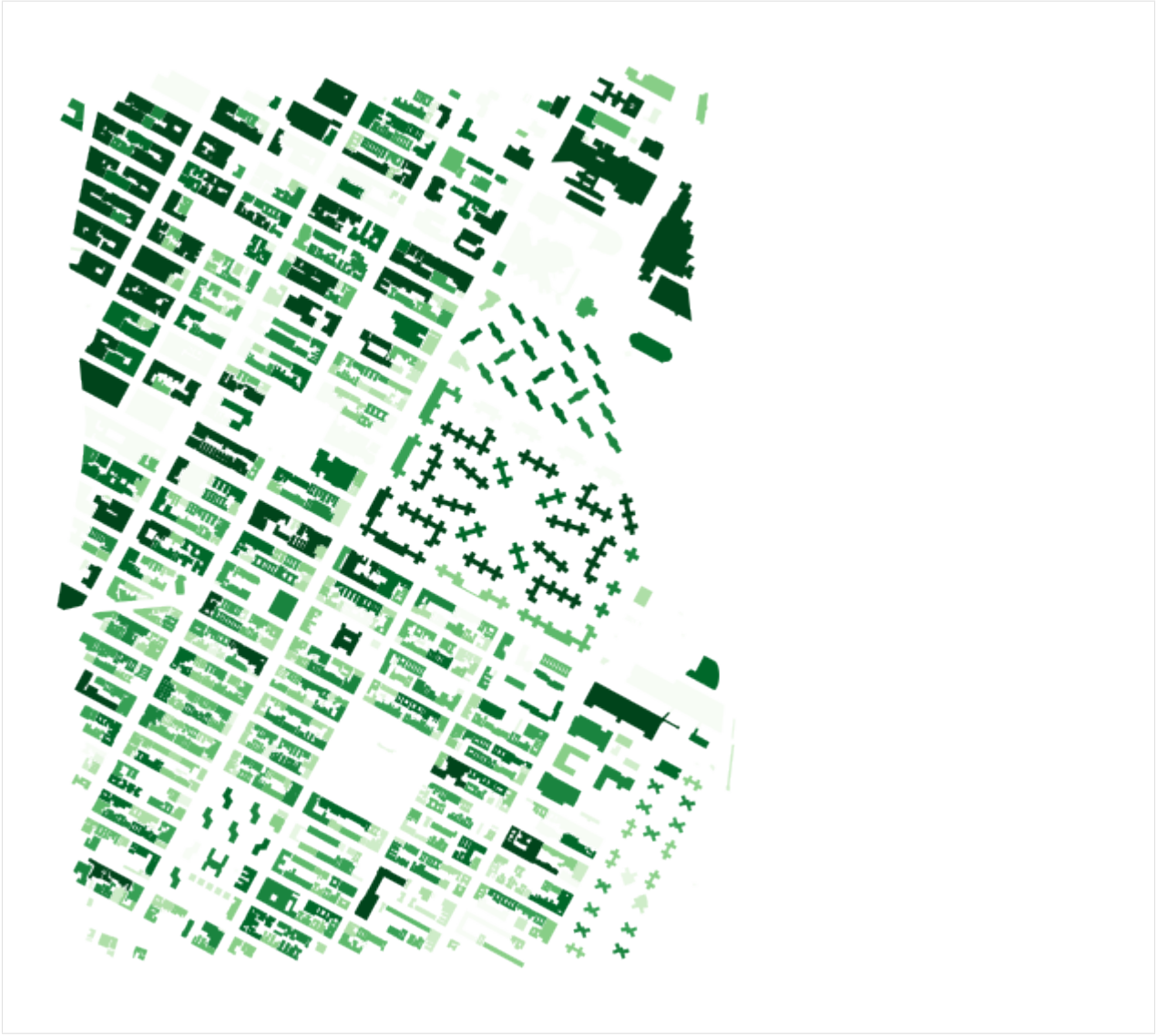
[8]:
```
buildings['height'] = buildings['height'].fillna(0).astype(float)
```

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='height', scheme='quantiles', k=10, legend=True, cmap=
↪'Blues')
ax.set_axis_off()
plt.show()
```

There are not many simple characters we can do with height, but `Volume` is possible. Unlike before, you have to pass the name of the column, `np.array`, or `pd.Series` where is stored height value. We have a column already.

[10]:
```
blg_volume = momepy.Volume(buildings, heights='height')
buildings['volume'] = blg_volume.series
```

[11]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='volume', legend=False, scheme='quantiles', k=10, cmap=
↪'Greens')
ax.set_axis_off()
plt.show()
```



Overview of all characters is available in API, with additional examples of usage. Some characters make sense to calculate only in specific cases. Prime example is `CourtyarArea` - there are many places where all buildings are courtyard-less, resulting in a `Series` full of zeros.

Shape

While the majority of `momepy` functions require the interaction of more `GeoDataFrames` or using spatial weights matrix, there are some which are calculated on single `GeoDataFrame` assessing the dimensions or shapes of features. This notebook illustrates how to measure simple shape characters.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
import numpy as np
```

We will again use `osmnx` to get the data for our example and after preprocessing of building layer will generate tessellation.

[4]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tess = momepy.Tessellation(buildings, unique_id='uID', limit=limit)
tessellation = tess.tessellation
```

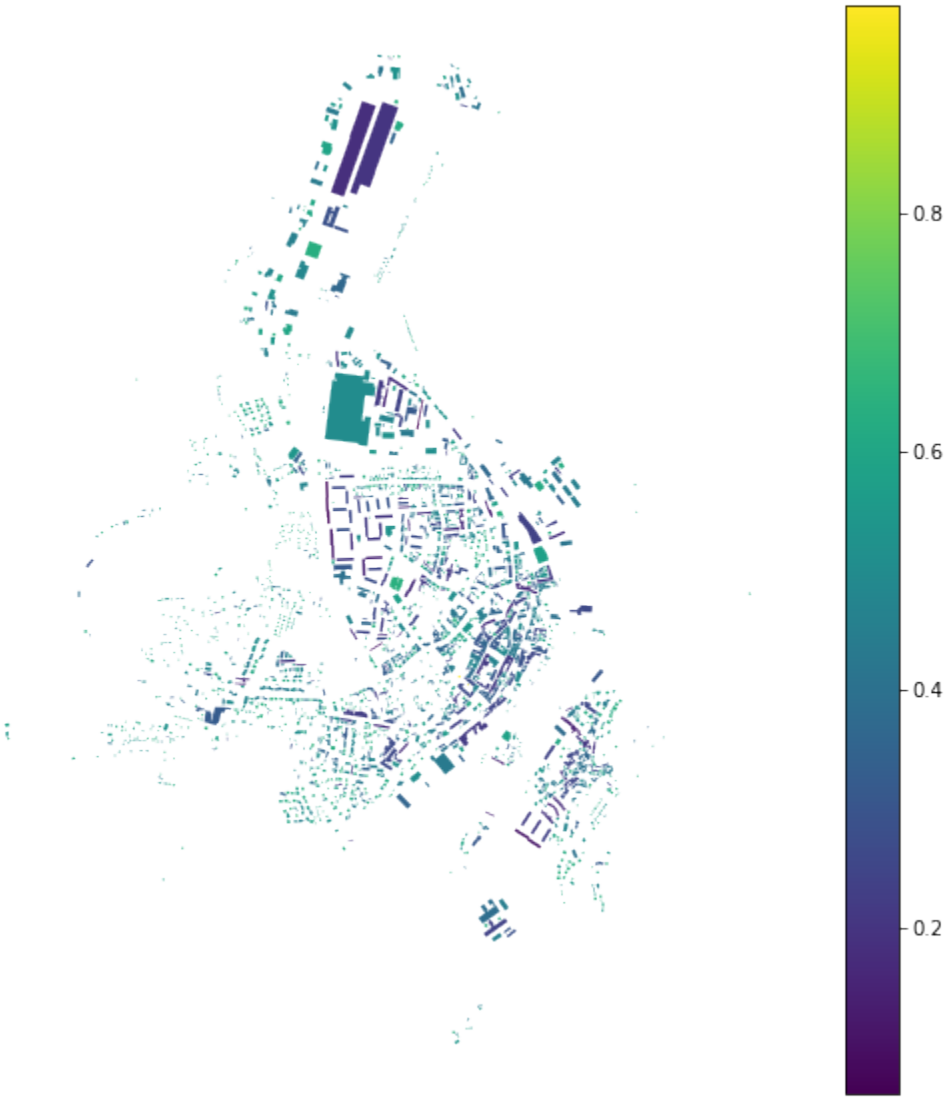
[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

Building shapes

Few examples of measuring building shapes. Circular compactness measures the ratio of object area to the area of its smallest circumsribed circle:

[4]:
```
blg_cc = momepy.CircularCompactness(buildings)
buildings['circular_com'] = blg_cc.series
```

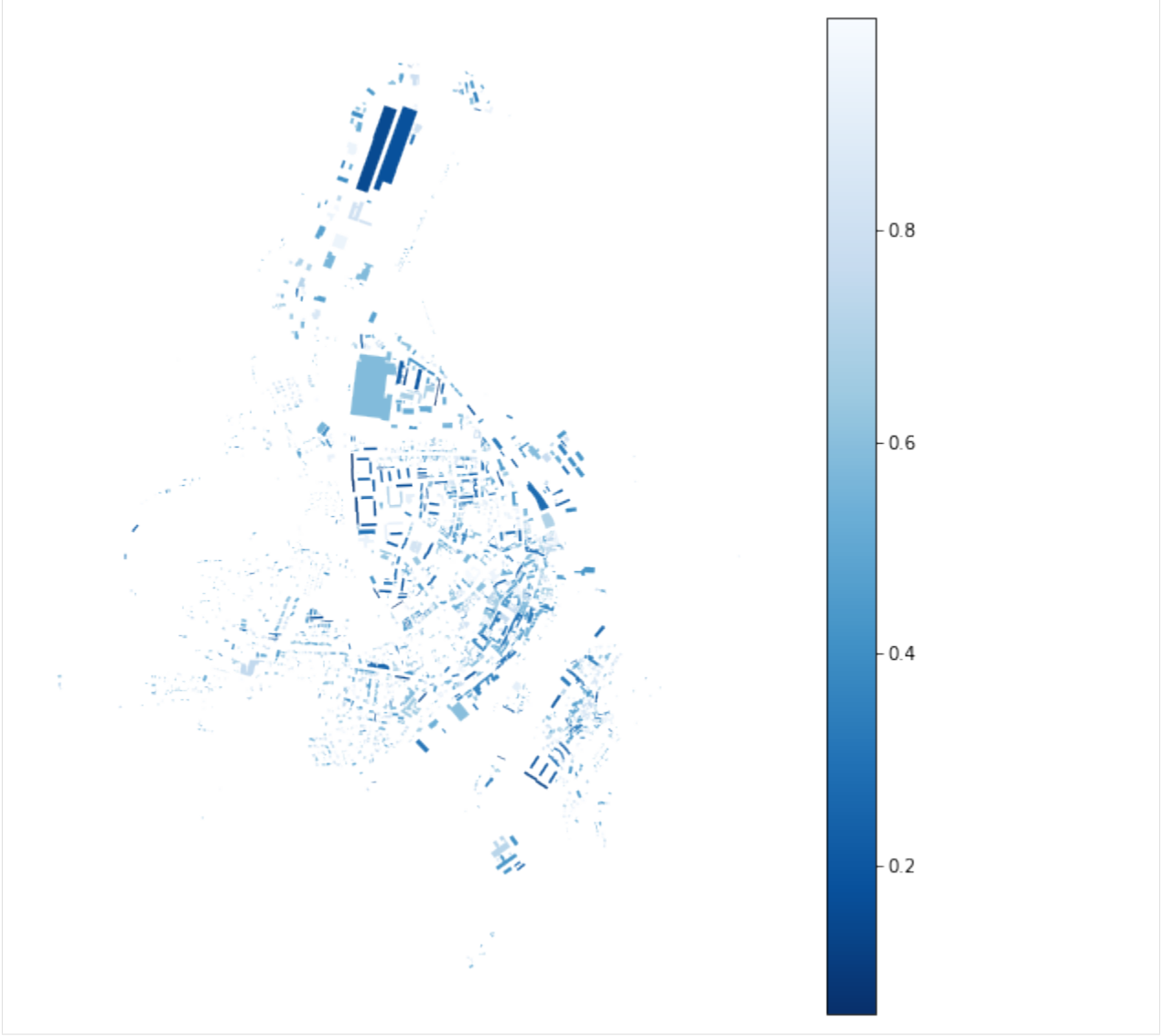
[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='circular_com', legend=True, cmap='viridis')
ax.set_axis_off()
plt.show()
```



While elongation is seen as elongation of its minimum bounding rectangle:

[6]:
```
blg_elongation = momepy.Elongation(buildings)
buildings['elongation'] = blg_elongation.series
```

[7]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='elongation', legend=True, cmap='Blues_r')
ax.set_axis_off()
plt.show()
```

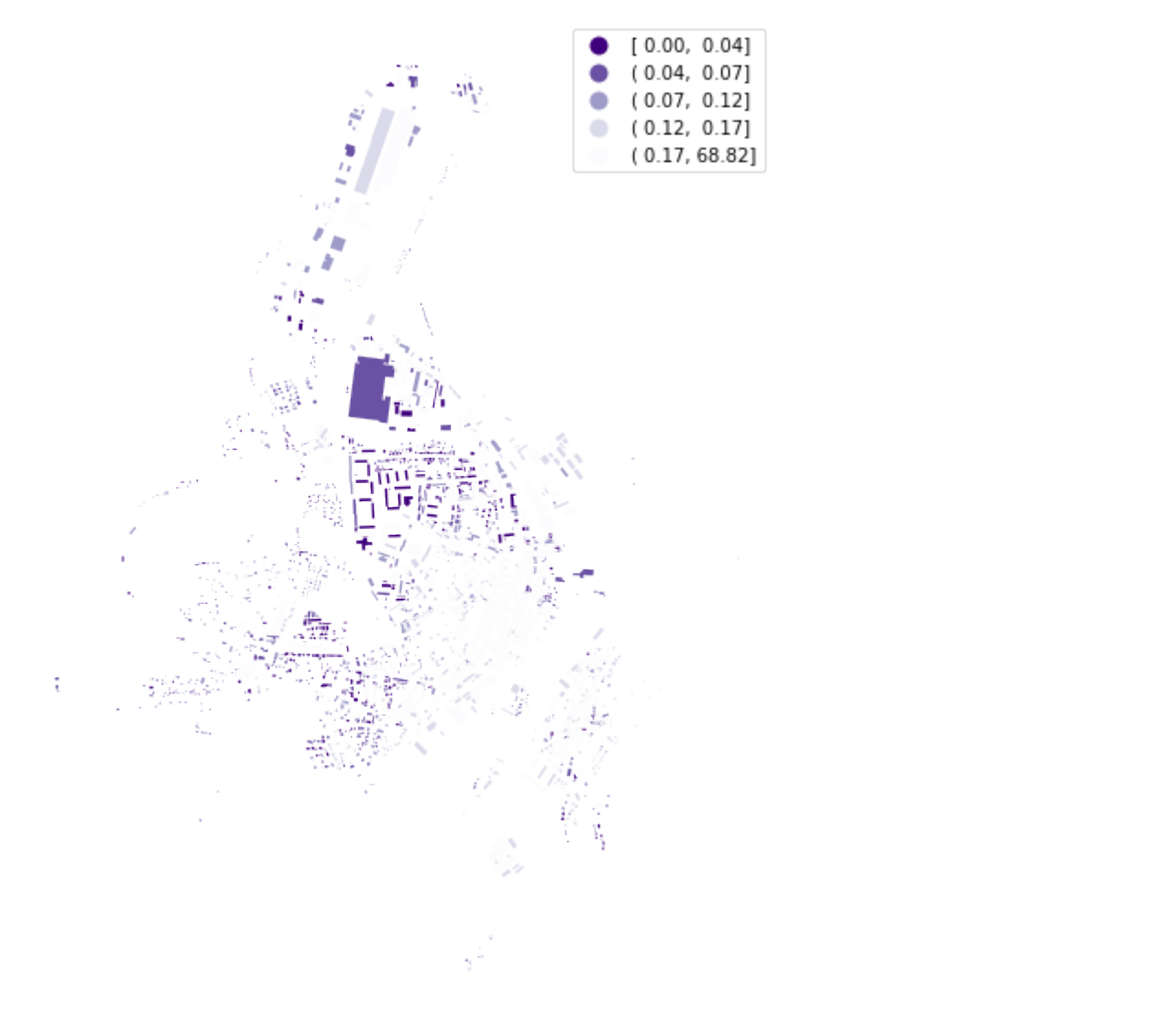


And squareness measures mean deviation of all corners from 90 degrees:

[8]:
```
blg_squareness = momepy.Squareness(buildings)
buildings['squareness'] = blg_squareness.series
```

```
100%|| 2005/2005 [00:00<00:00, 3019.57it/s]
```

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='squareness', legend=True, scheme='quantiles', cmap=
↪'Purples_r')
ax.set_axis_off()
plt.show()
```

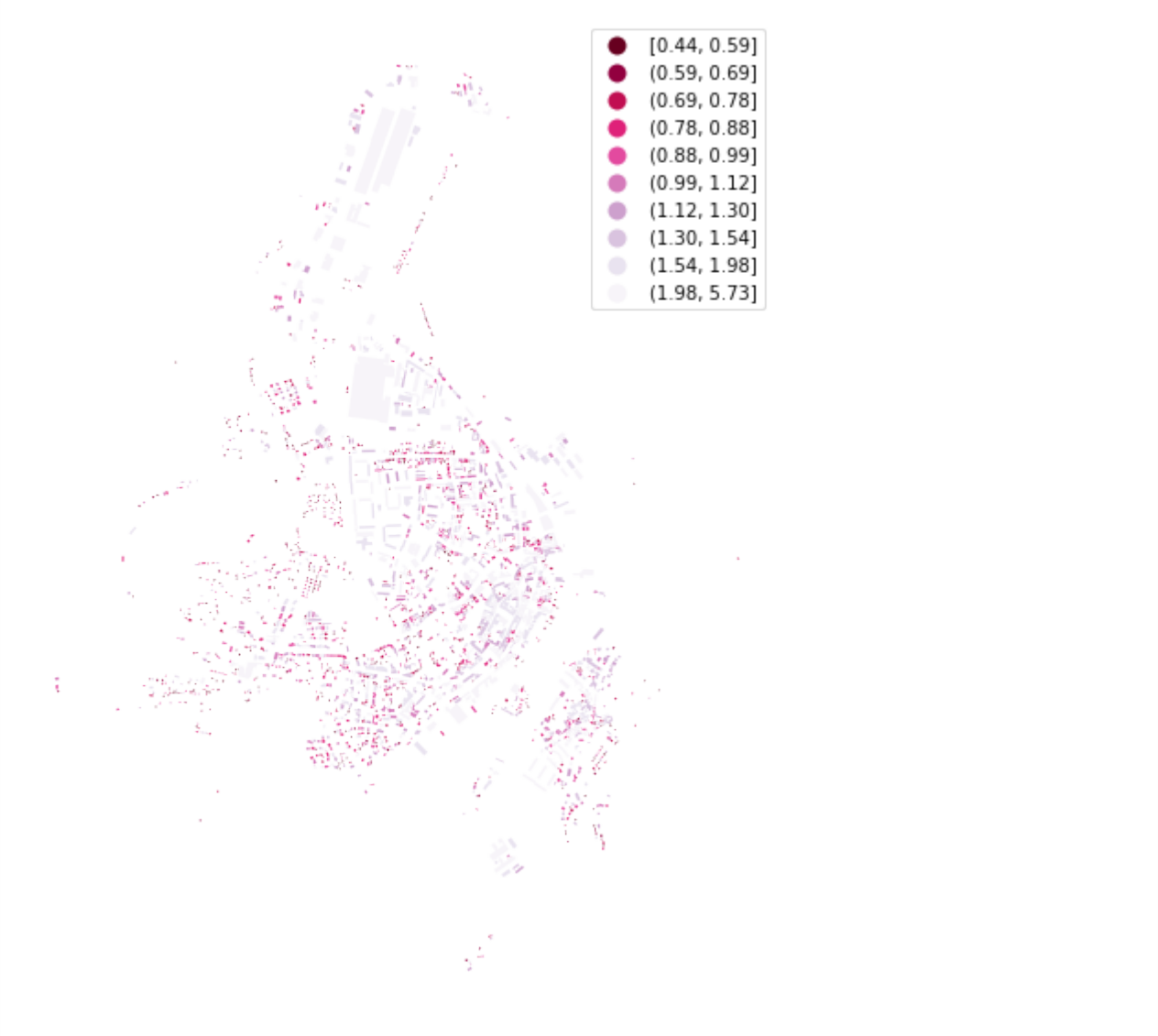


For the form factor, we need to know the volume of each building. While we do not have building height data for Kahla, we will generate them randomly and pass a `Series` containing volume values to `FormFactor`.

Note: For the majority of parameters you can pass values as the name of the column, `np.array`, `pd.Series` or any other list-like object.

[10]:
```
blg_volume = momepy.Volume(buildings, np.random.randint(4, 20, size=len(buildings)))
buildings['formfactor'] = momepy.FormFactor(buildings, volumes=blg_volume.series).
↪series
```

[11]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='formfactor', legend=True, scheme='quantiles', k=10,␣
↪cmap='PuRd_r')
ax.set_axis_off()
plt.show()
```

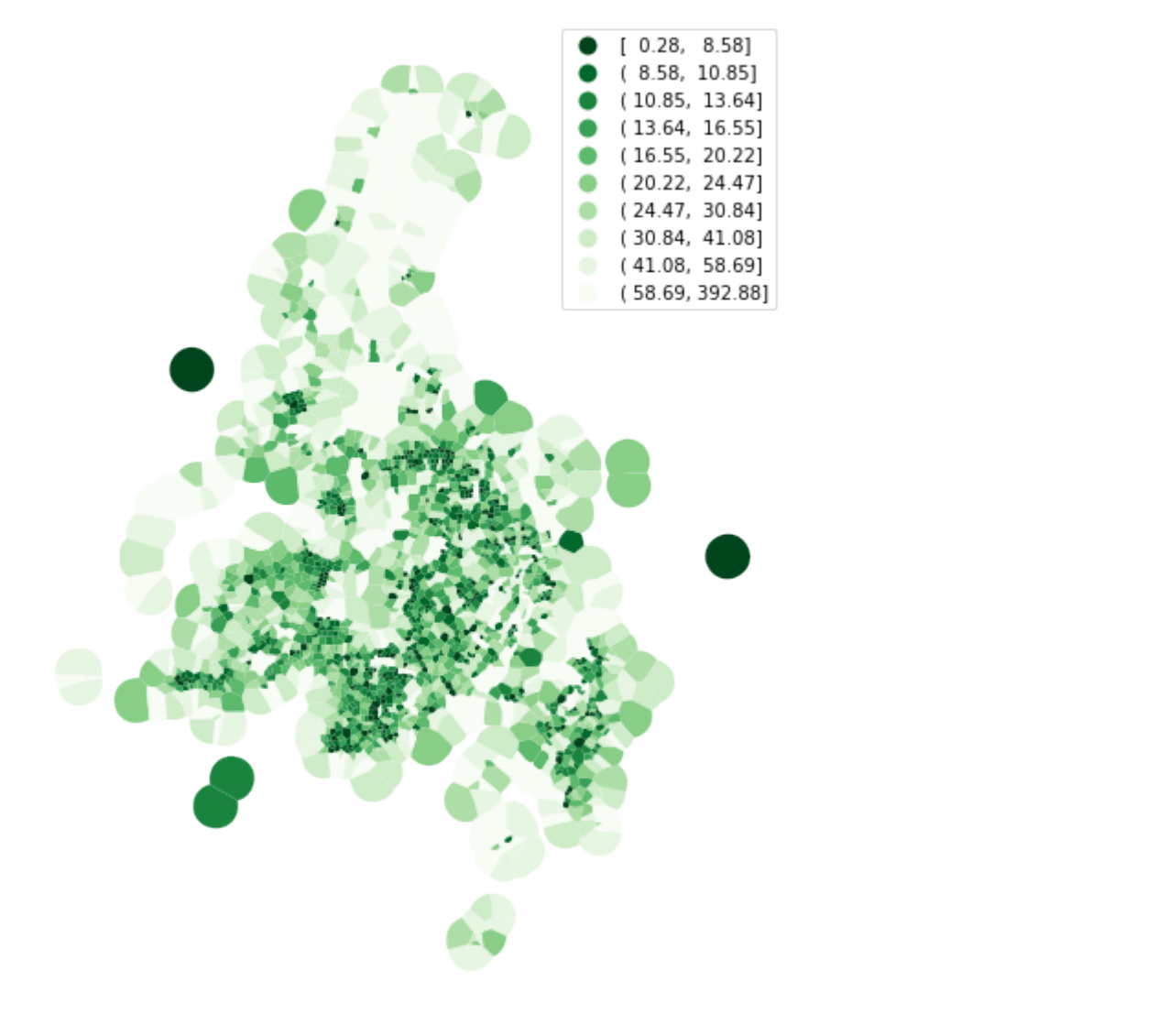


Cell shapes

In theory, you can measure most of the 2D characters on all elements, including tessellation or blocks:

[12]:
```
tes_cwa = momepy.CompactnessWeightedAxis(tessellation)
tessellation['cwa'] = tes_cwa.series
```

[13]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='cwa', legend=True, scheme='quantiles', k=10, cmap=
↪'Greens_r')
ax.set_axis_off()
plt.show()
```



Street network shapes

There are some characters which requires street network as an input. We can again use `osmnx` to retrieve it from OSM.

[14]:
```
streets_graph = ox.graph_from_place('Kahla, Germany', network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
```

`osmnx` returns networkx Graph. While momepy works with graph in some cases, for this one we need `GeoDataFrame`. To get it, we can use `ox.graph_to_gdfs`.

Note: `momepy.nx_to_gdf` might work as well, but OSM network needs to be complete in that case. `osmnx` takes care of it.

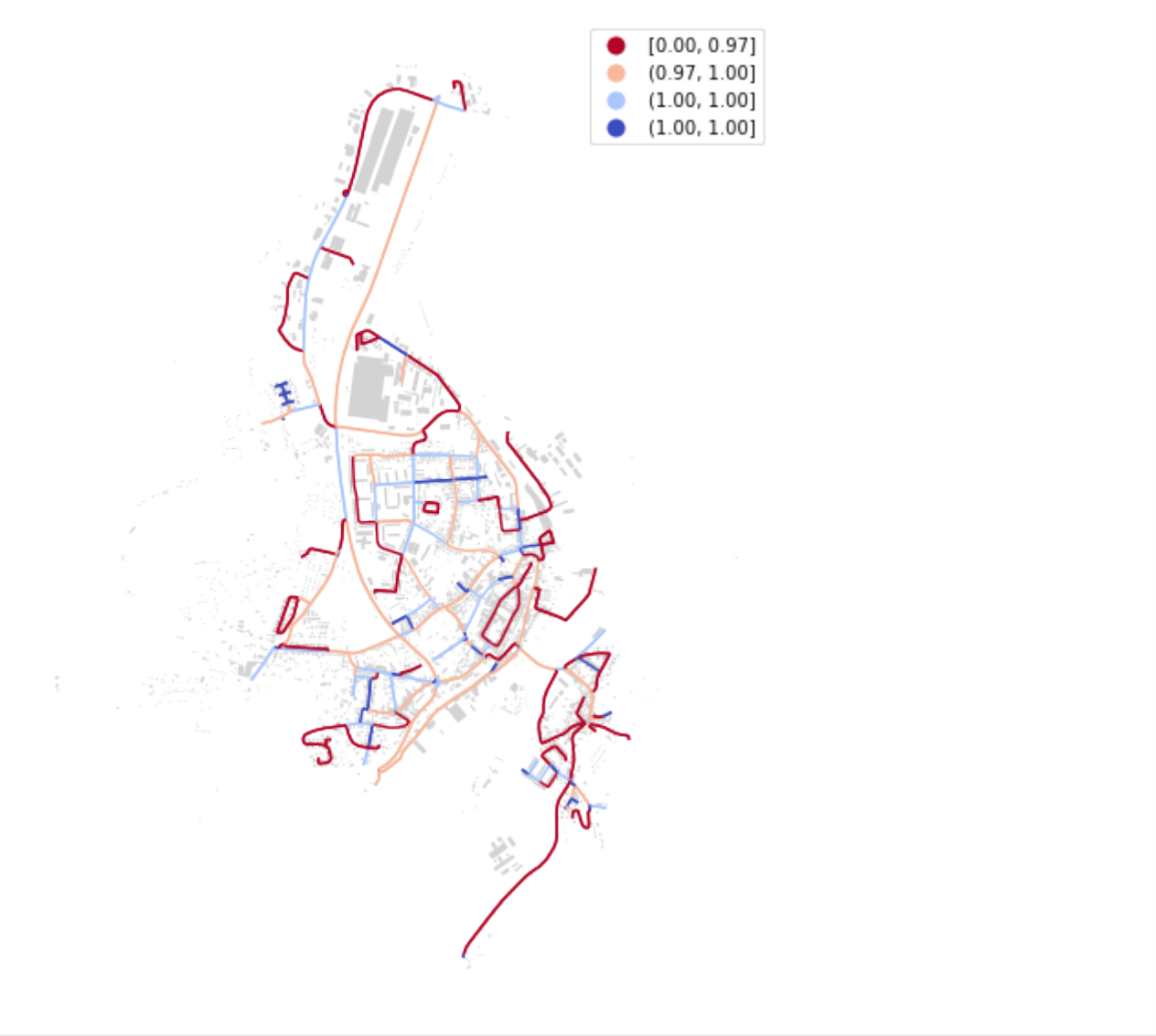
[15]:
```
edges = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

[16]:
```
f, ax = plt.subplots(figsize=(10, 10))
edges.plot(ax=ax, color='pink')
buildings.plot(ax=ax, color='lightgrey')
ax.set_axis_off()
plt.show()
```


Now we can calculate linearity of each segment:

[17]:
```
edg_lin = momepy.Linearity(edges)
edges['linearity'] = edg_lin.series
```

[18]:
```
f, ax = plt.subplots(figsize=(10, 10))
edges.plot(ax=ax, column='linearity', legend=True, cmap='coolwarm_r', scheme=
↪'quantiles', k=4)
buildings.plot(ax=ax, color='lightgrey')
ax.set_axis_off()
plt.show()
```



8.2.5 Characters based on multiple GeoDataFrames

One of the ways how to analyze urban patterns is to study relationships between different elements of urban form. `momepy` does exactly that. Following notebooks will show how to use `momepy` to measure characters of urban form based on two or more `GeoDataFrames`.

This section covers:

Measuring spatial distribution

Spatial distribution can be captured many ways. This notebook show couple of them, based on orientation and street corridor.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

[2]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building': True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 2939/2939 [00:00<00:00, 6779.09it/s]
Changing geometry: 100%|| 629/629 [00:04<00:00, 146.34it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 2122/2122 [00:00<00:00, 15458.70it/s]
Changing geometry: 100%|| 172/172 [00:00<00:00, 176.41it/s]
```

```
Inward offset...
Generating input point array...
Generating Voronoi diagram...
Generating GeoDataFrame...
Dissolving Voronoi polygons...
```

[3]:
```
streets_graph = ox.graph_from_place('Kahla, Germany', network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
edges = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

Alignment

We can measure alignment of different elements to their neighbours (for which `spatial_weights` are needed) or to different elements. We will explore cell alignment (difference of orientation of buildings and cells) and street alignment (difference of orientation of buildings and street segments).

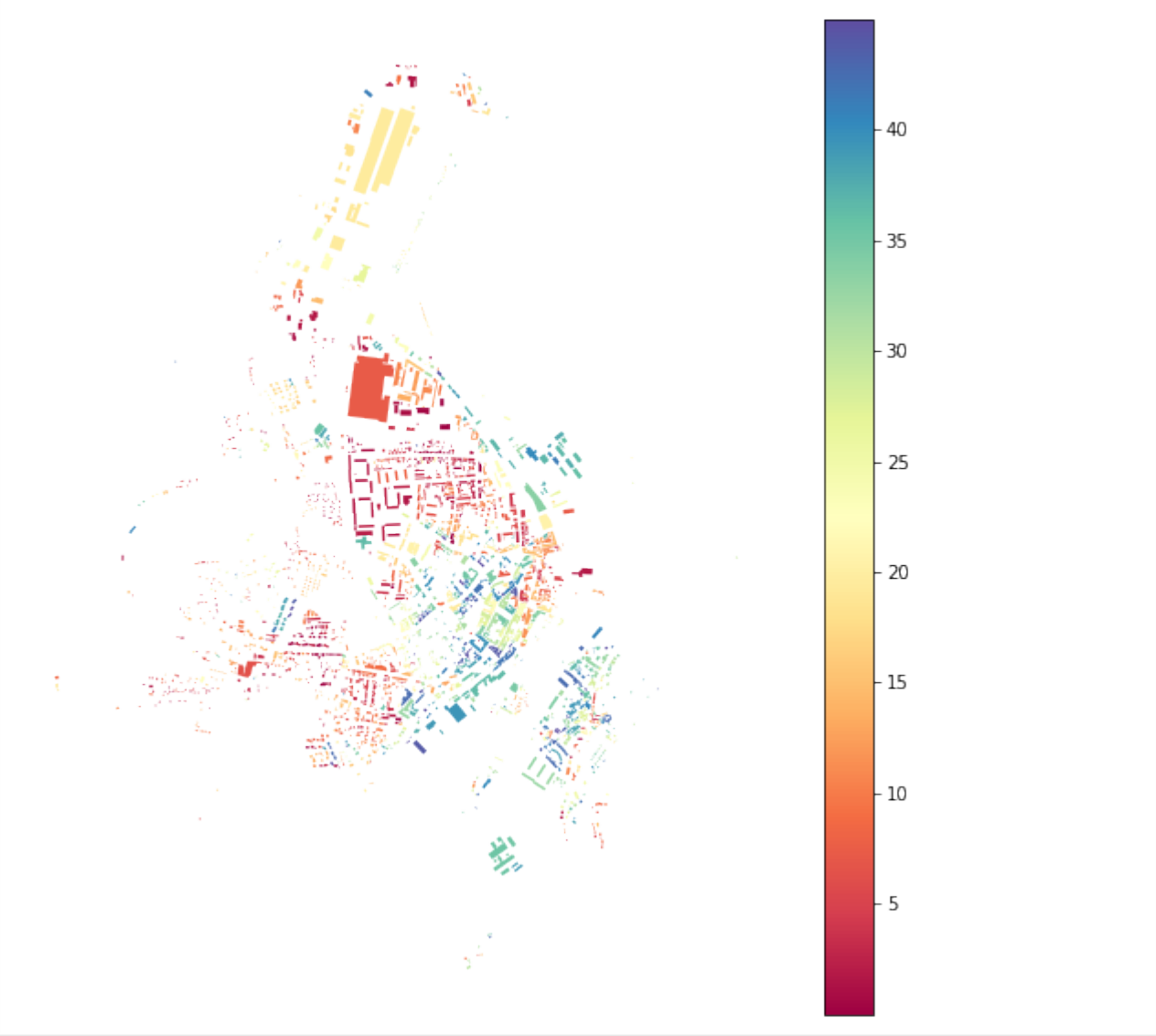
Cell alignment

For `CellAlignment` we need to know orientations, so let's calculate them first. Orientation is defined as an orientation of the longext axis of bounding rectangle in range [0,45). It captures the deviation of orientation from cardinal directions:

[4]:
```
buildings['orientation'] = momepy.Orientation(buildings).series
tessellation['orientation'] = momepy.Orientation(tessellation).series
```

```
100%|| 2011/2011 [00:00<00:00, 2470.29it/s]
100%|| 2011/2011 [00:07<00:00, 273.61it/s]
```

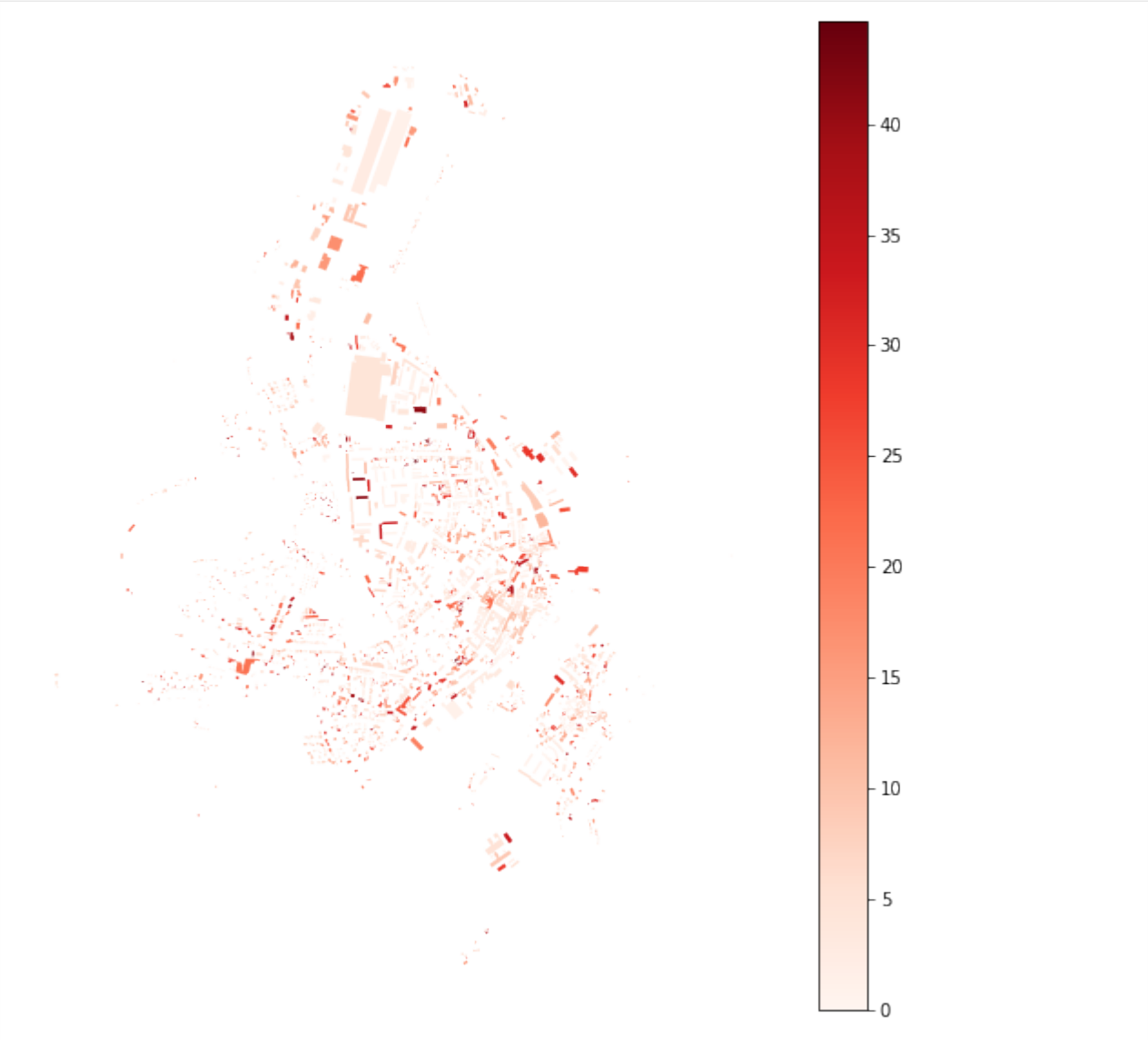
[5]:
```
buildings.plot(column='orientation', legend=True, cmap='Spectral',
               figsize=(10, 10)).set_axis_off()
```



`CellAlignment` requires both gdfs, orientation values for left and right gdf and unique ID linking both gdfs as is in left and right gdf:

[6]:
```
blg_cell_align = momepy.CellAlignment(buildings, tessellation,
                                      'orientation', 'orientation',
                                      'uID', 'uID')
buildings['cell_align'] = blg_cell_align.series
```

[7]:
```
buildings.plot(column='cell_align', legend=True, cmap='Reds',
               figsize=(10, 10)).set_axis_off()
```



No really clear pattern is visible in this case, but it might be in other, especially comparing building orientation with plots.

Street alignment

Street alignment works on the same principle as cell alignment. What we do not have at this moment is network ID, so we have to generate it and link it to buildings:

[8]:
```
edges['networkID'] = momepy.unique_id(edges)
buildings['networkID'] = momepy.get_network_id(buildings, edges,
                                               'networkID')
```

```
Snapping:  10%|         | 192/2011 [00:00<00:00, 1910.30it/s]
```

```
Generating centroids...
Generating rtree...
```

```
Snapping: 100%|| 2011/2011 [00:01<00:00, 1929.06it/s]
/Users/martin/Git/momepy/momepy/elements.py:806: UserWarning: Some objects were not␣
↪attached to the network. Set larger min_size. 239 affected elements
  "Set larger min_size. {} affected elements".format(sum(series.isnull()))
```

Note: UserWarning tells us, that not all buildings were linked to the network. Keep in mind that it may cause issues with missing values later.

OSM network is not ideal in our case, it is missing in part of the study area. Some objects were not attached to the network with `min_size` defaulting to 100 metres. We can either use larger distance or drop unlinked buildings.

[9]:
```
buildings_net = buildings.loc[buildings.networkID >= 0]
```

[10]:
```
buildings_net.plot(figsize=(10, 10)).set_axis_off()
```

`StreetAlignment` will take care of street orientation (saved under `orientations` attribute):

[11]:
```
str_align = momepy.StreetAlignment(buildings_net, edges,
```

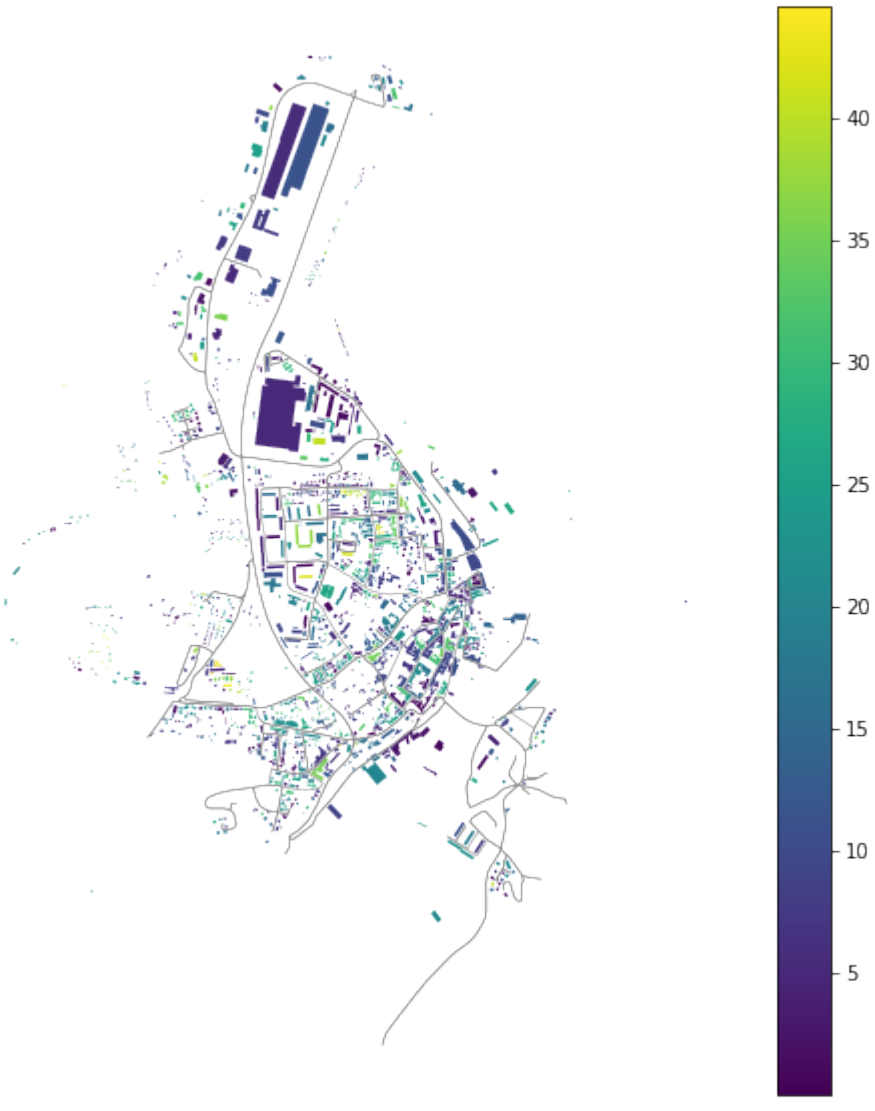
(continues on next page)

(continued from previous page)

```
                                    'orientation', 'networkID',
                                    'networkID')
buildings_net['str_align'] = str_align.series
```

```
/Users/martin/Git/geopandas/martinfleis/geopandas/geodataframe.py:1109:␣
↪SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/
↪user_guide/indexing.html#returning-a-view-versus-a-copy
  super(GeoDataFrame, self).__setitem__(key, value)
```

[12]:
```
ax = edges.plot(color='grey', linewidth=0.5, figsize=(10, 10))
buildings_net.plot(ax=ax, column='str_align', legend=True)
ax.set_axis_off()
```



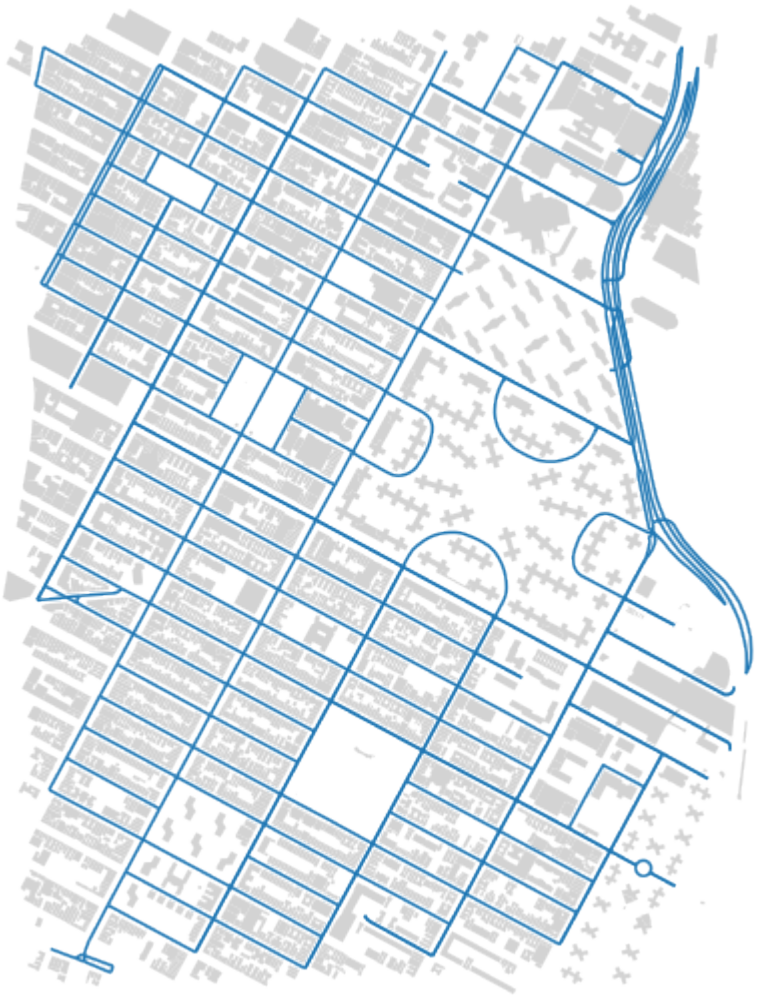
Street profile

`StreetProfile` captures several characters at the same time. It generates a series of perpendicular ticks of set length and set spacing and returns mean widths of street profile, their standard deviation, mean height and its standard deviation, profile as a ratio of widht and height and degree of openness. If heights are not passed, it will not return them and profile. We will use Manhattan example to illustrate how it works. Building `height` column is converted to float and buildings are exploded to avoid multipolygons.

[13]:
```
point = (40.731603, -73.977857)
dist = 1000
gdf = ox.geometries.geometries_from_point(point, dist=dist, tags={'building':True})
buildings = ox.projection.project_gdf(gdf)
buildings = buildings[buildings.geom_type.isin(['Polygon', 'MultiPolygon'])]
buildings['height'] = buildings['height'].fillna(0).astype(float)
buildings = buildings.explode()
buildings.reset_index(inplace=True, drop=True)
```

[14]:
```
streets_graph = ox.graph_from_point(point, dist, network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
edges = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

[15]:
```
ax = buildings.plot(figsize=(10, 10), color='lightgrey')
edges.plot(ax=ax)
ax.set_axis_off()
```



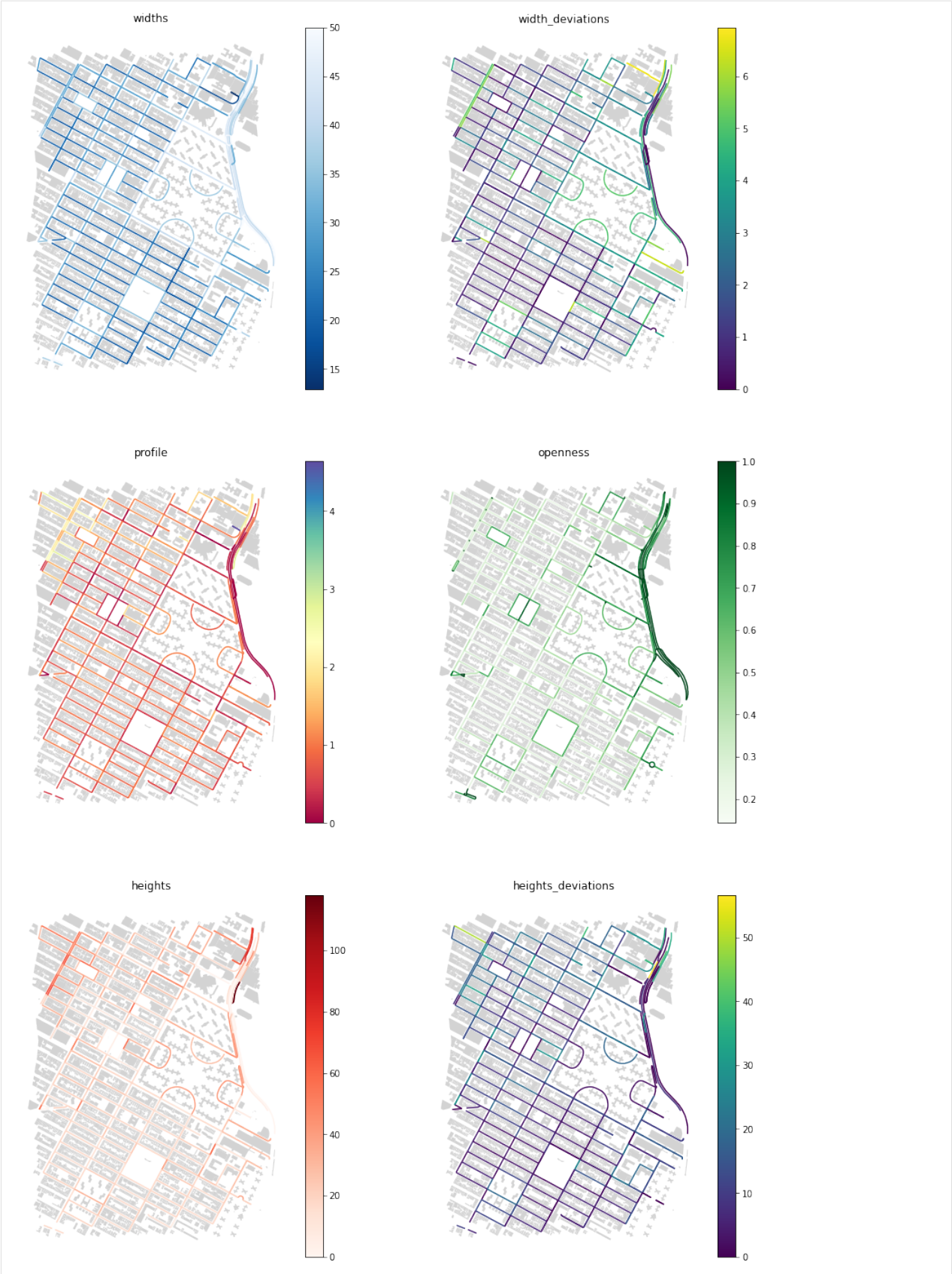
[16]:
```
profile = momepy.StreetProfile(edges, buildings, heights='height')
```

```
/opt/miniconda3/envs/geo_dev/lib/python3.7/site-packages/numpy/lib/nanfunctions.py:
↪1667: RuntimeWarning: Degrees of freedom <= 0 for slice.
  keepdims=keepdims)
/Users/martin/Git/momepy/momepy/dimension.py:625: RuntimeWarning: invalid value␣
↪encountered in long_scalars
  openness.append(np.isnan(s).sum() / (f).sum())
```

We can assign measrued characters as columns of edges gdf:

[17]:
```
edges['widths'] = profile.w
edges['width_deviations'] = profile.wd
edges['openness'] = profile.o
edges['heights'] = profile.h
edges['heights_deviations'] = profile.hd
edges['profile'] = profile.p
```

[18]:
```
f, axes = plt.subplots(figsize=(15, 25), ncols=2, nrows=3)
edges.plot(ax=axes[0][0], column='widths', legend=True, cmap='Blues_r')
buildings.plot(ax=axes[0][0], color='lightgrey')
edges.plot(ax=axes[0][1], column='width_deviations', legend=True)
buildings.plot(ax=axes[0][1], color='lightgrey')
axes[0][0].set_axis_off()
axes[0][0].set_title('widths')
axes[0][1].set_axis_off()
axes[0][1].set_title('width_deviations')
edges.plot(ax=axes[1][0], column='profile', legend=True, cmap='Spectral')
buildings.plot(ax=axes[1][0], color='lightgrey')
edges.plot(ax=axes[1][1], column='openness', legend=True, cmap='Greens')
buildings.plot(ax=axes[1][1], color='lightgrey')
axes[1][0].set_axis_off()
axes[1][0].set_title('profile')
axes[1][1].set_axis_off()
axes[1][1].set_title('openness')
edges.plot(ax=axes[2][0], column='heights', legend=True, cmap='Reds')
buildings.plot(ax=axes[2][0], color='lightgrey')
edges.plot(ax=axes[2][1], column='heights_deviations', legend=True)
buildings.plot(ax=axes[2][1], color='lightgrey')
axes[2][0].set_axis_off()
axes[2][0].set_title('heights')
axes[2][1].set_axis_off()
axes[2][1].set_title('heights_deviations')
plt.show()
```


widths
50
45
40
35
30
25
20
15
width_deviations
6
5
4
3
2
1
0
profile
4
3
2
1
0
openness
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
heights
100
80
60
40
20
0
heights_deviations
50
40
30
20
10
0

Measuring density

Measuring density is a typical exercise in urban analytics. momepy allows to measure different types (see API/Intensity); this notebook will outline the main principles.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use osmnx to get the data for our example and after preprocessing of building layer will generate tessellation layer.

[2]:
```
import osmnx as ox

point = (40.731603, -73.977857)
dist = 1000
gdf = ox.geometries.geometries_from_point(point, dist=dist, tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)
gdf_projected = gdf_projected[gdf_projected.geom_type.isin(['Polygon', 'MultiPolygon
↪'])]

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 3200/3200 [00:02<00:00, 1481.66it/s]
Changing geometry: 100%|| 2157/2157 [00:17<00:00, 120.11it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 1459/1459 [00:00<00:00, 2104.02it/s]
Changing geometry: 100%|| 863/863 [00:05<00:00, 171.39it/s]
```

```
Inward offset...
Discretization...
```

```
  9%|         | 71/833 [00:00<00:01, 706.33it/s]
```

```
Generating input point array...
```

```
100%|| 833/833 [00:01<00:00, 587.68it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 356220/356220 [00:06<00:00, 51892.45it/s]
```

```
Dissolving Voronoi polygons...
```

```
100%|| 3/3 [00:00<00:00, 716.73it/s]
```

```
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

We have some edge effect here as we are using the buffer as a limit for tessellation in the middle of the urban fabric, but for these examples, we can work with it anyway. Keep in mind that values on the edge of this area will be skewed.

Covered Area Ratio

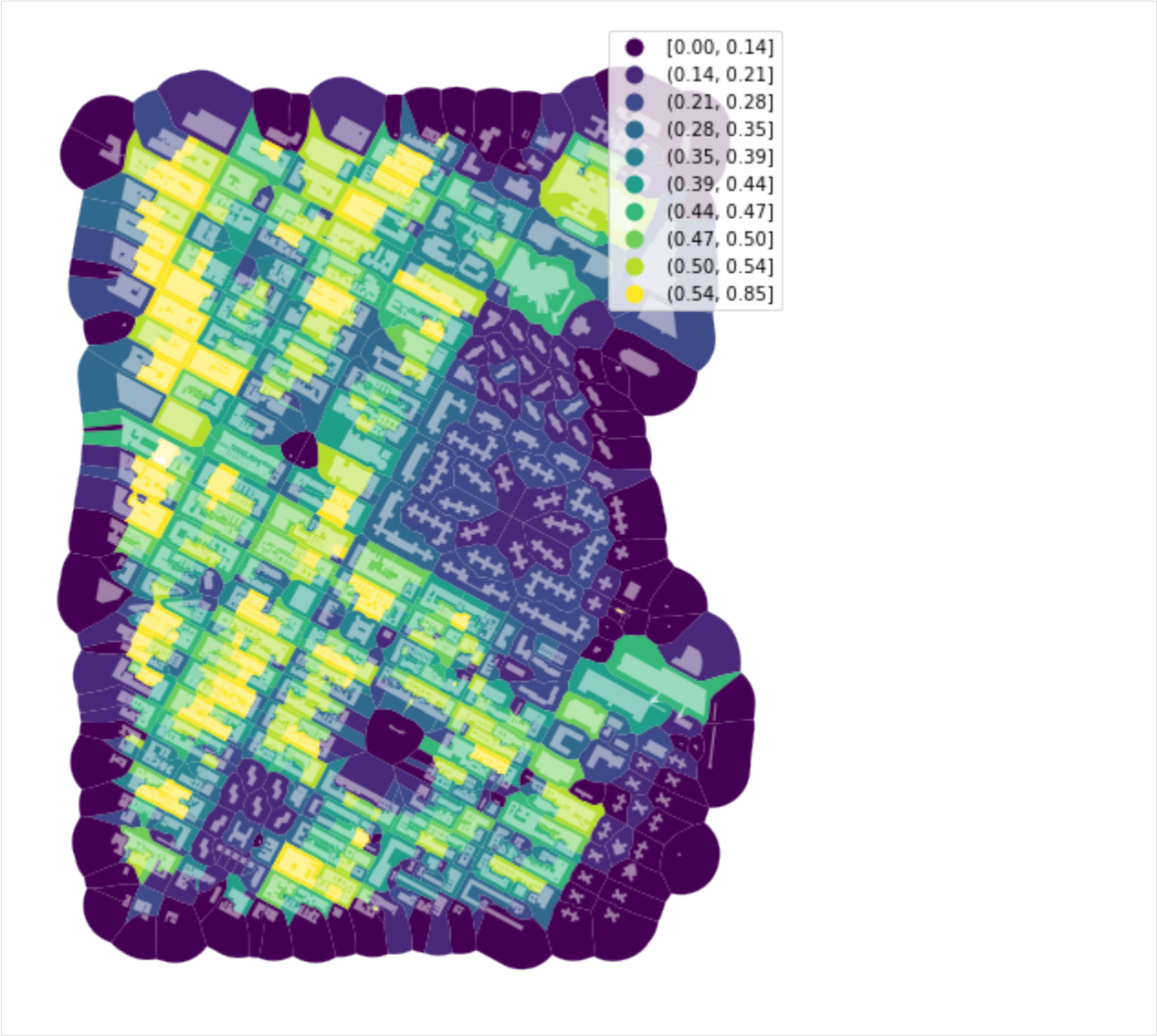
Covered area ratio, in our case measured on tessellation cells, requires `GeoDataFrame` containing spatial unit (cell, plot), and `GeoDataFrame` containing covering objects (buildings). On top of that, it currently requires passed areas for both gdfs and unique ID which links together spatial units and objects on them. We can either calculate areas before:

[4]:
```
tessellation['area'] = momepy.Area(tessellation).series
buildings['area'] = momepy.Area(buildings).series
tess_car = momepy.AreaRatio(tessellation, buildings, 'area', 'area', 'uID')
tessellation['CAR'] = tess_car.series
```

Or we can pass `momepy.Area().series` directly:

[5]:
```
tess_car = momepy.AreaRatio(tessellation, buildings,
                            momepy.Area(tessellation).series,
                            momepy.Area(buildings).series, 'uID')
tessellation['CAR'] = tess_car.series
```

[6]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='CAR', legend=True, scheme='quantiles', k=10, cmap=
↪'viridis')
buildings.plot(ax=ax, color='white', alpha=0.5)
ax.set_axis_off()
plt.show()
```



Floor Area Ratio

Because we know building heights for our `buildings` gdf, we can also calculate FAR. This part of New York has height data, only stored as strings, so we have to convert them to `floats` (or `int`) and fill `NaN` values with zero.

FAR requires floor areas for building gdf instead of covered area.

[7]:
```
buildings['height'] = buildings['height'].fillna(0).astype(float)

buildings['floor_area'] = momepy.FloorArea(buildings, 'height').series
```

[8]:
```
tessellation['FAR'] = momepy.AreaRatio(tessellation, buildings,
                                       'area', 'floor_area', 'uID').series
```

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='FAR', legend=True, scheme='quantiles', k=10, cmap=
↪'viridis')
```

(continues on next page)

(continued from previous page)

```
buildings.plot(ax=ax, color='white', alpha=0.5)
ax.set_axis_off()
plt.show()
```

Location-based density is described in *examples using spatial weights*.

8.2.6 Using spatial weights matrix

To capture patterns in urban form, we have to analyze morphometric characters within a spatial context, taking into the account neighbouring elements. A simple example could be a mean height within 100 metres (measured as a location-based character for each building). `momepy` is using `libpysal` spatial weights to capture the relationship between elements, in this case, buildings, in the form of a binary matrix (1 = neighbours, 0 = not neighbours). Spatial weights are an essential part of `momepy` and are used on many occasions and in various forms. However, they should always be based on the unique ID of element, not index (there are a few exceptions documented in API).

This section covers:

Generating spatial weights

momepy is using libpysal to handle spatial weights, but also builds on top of it. This notebook will show how to use different weights.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use osmnx to get the data for our example and after preprocessing of building layer will generate tessellation layer.

[2]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 2932/2932 [00:00<00:00, 6196.10it/s]
Changing geometry: 100%|| 630/630 [00:04<00:00, 143.29it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 2115/2115 [00:00<00:00, 16000.91it/s]
Changing geometry: 100%|| 172/172 [00:00<00:00, 195.82it/s]
```

```
Inward offset...
Discretization...
```

```
  6%|         | 112/2008 [00:00<00:01, 1119.47it/s]
```

```
Generating input point array...
```

```
100%|| 2008/2008 [00:01<00:00, 1840.40it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 249298/249298 [00:05<00:00, 48790.35it/s]
```

```
Dissolving Voronoi polygons...
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

```
0it [00:00, ?it/s]
```

Queen contiguity

Morphological tessellation allows using contiguity-based weights matrix. While `libpysal.weights.contiguity.Queen` will do the standard Queen contiguity matrix of the first order; it might not be enough to capture proper context. For that reason, we can use `momepy.sw_high` to capture all neighbours within set topological distance `k`. It generates spatial weights of higher orders under the hood and joins them together.

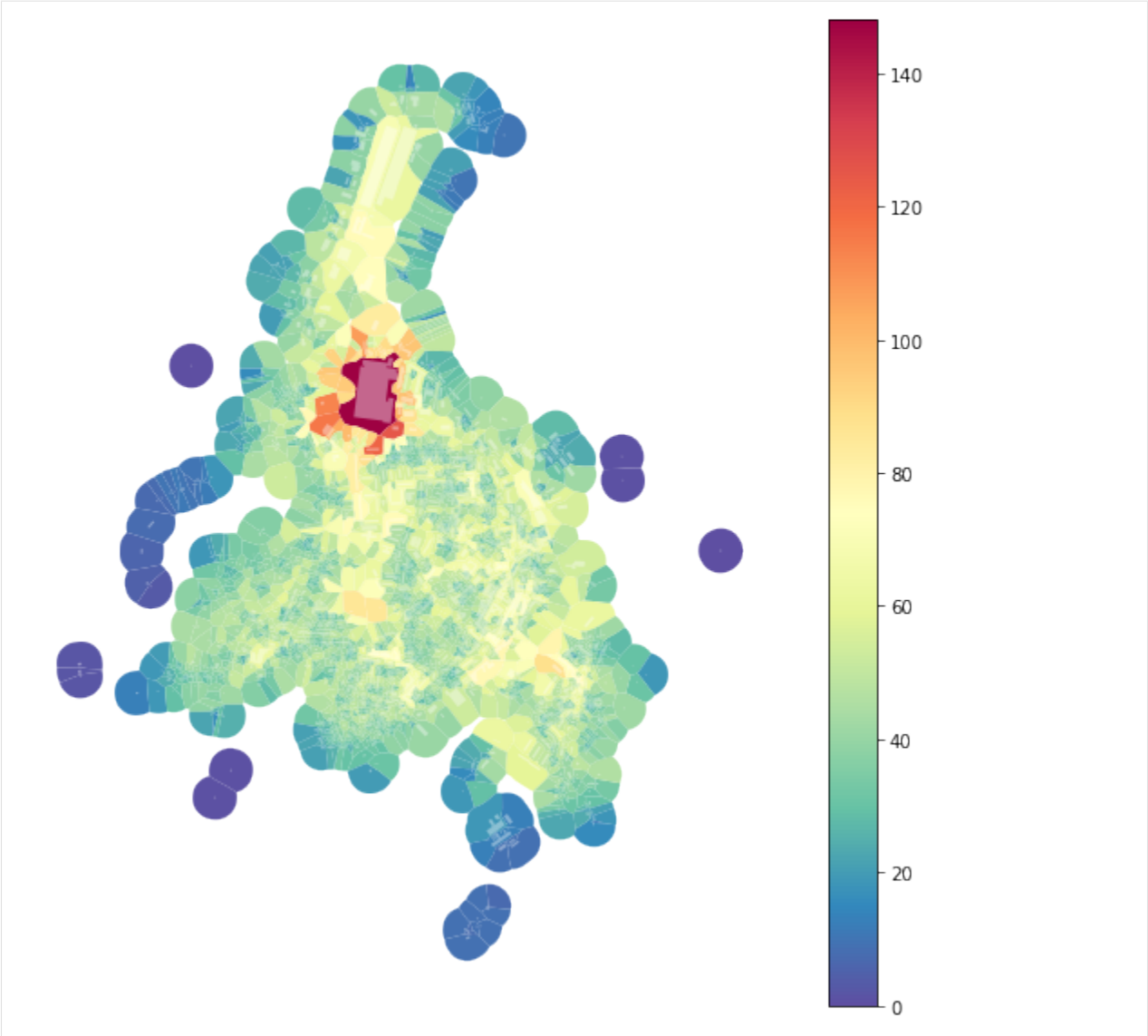
[3]:
```
sw3 = momepy.sw_high(k=3, gdf=tessellation, ids='uID')
```

Queen contiguity of morphological tessellation can capture the comparable level of information across the study area - the number of the neighbour is relatively similar and depends on the morphology of urban form. We can visualize it by counting the number of neighbours (as captured by `sw3`).

[4]:
```
tessellation['neighbours'] = momepy.Neighbors(tessellation, sw3,'uID').series
```

```
100%|| 2005/2005 [00:00<00:00, 94463.12it/s]
```

[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='neighbours', legend=True, cmap='Spectral_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



Distance

Often we want to define the neighbours based on metric distance. We will look at two options - distance band and k-nearest neighbour.

Distance band

We can imagine distance band as a buffer of a set radius around each object, for example, 400 meters. For that, we can use `libpysal.weights.DistanceBand`:

[6]:
```
import libpysal
dist400 = libpysal.weights.DistanceBand.from_dataframe(buildings, 400,
                                                       ids='uID')
```

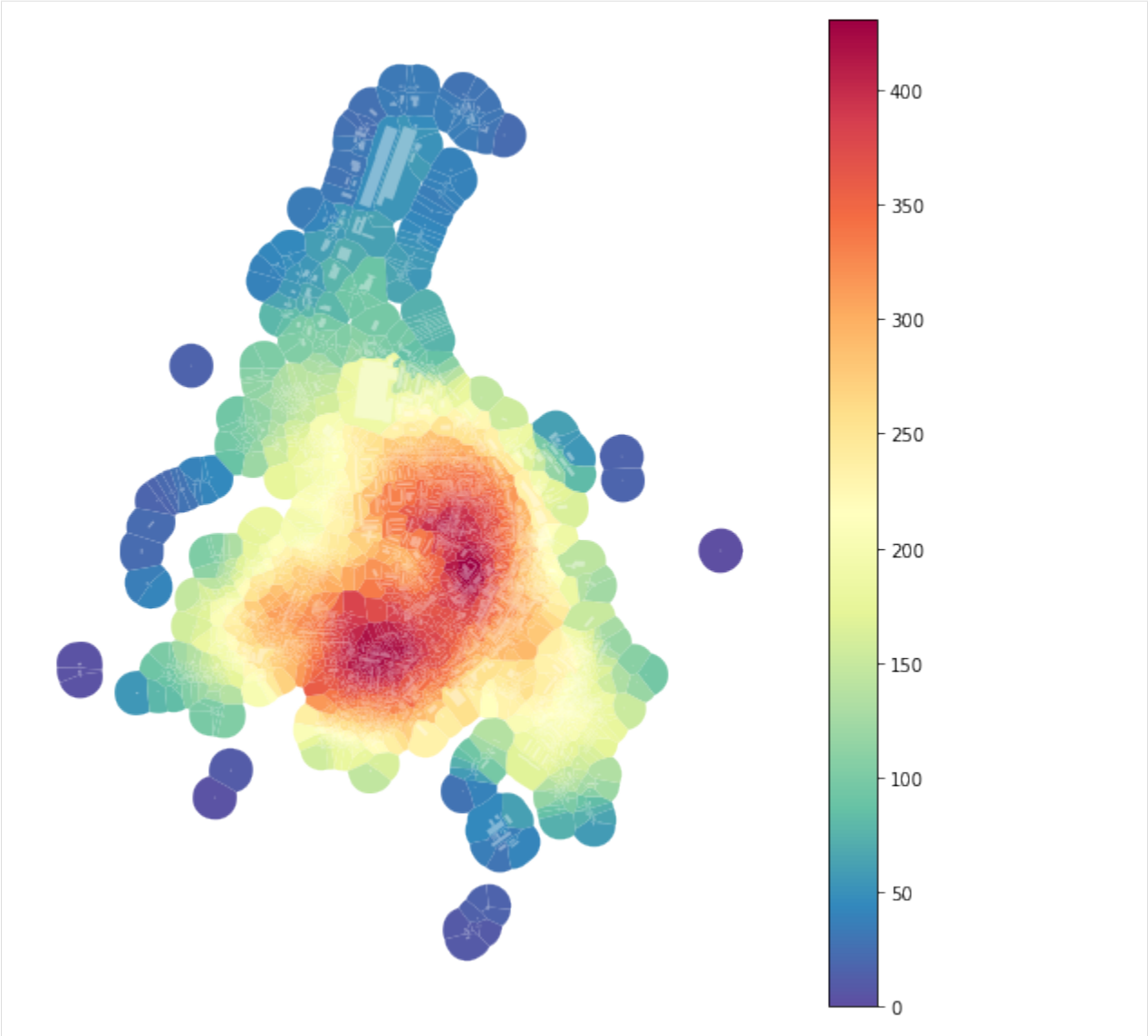
```
/opt/miniconda3/envs/momepy_guide/lib/python3.8/site-packages/libpysal/weights/
↪weights.py:172: UserWarning: The weights matrix is not fully connected:
 There are 2 disconnected components.
 There is 1 island with id: 212.
  warnings.warn(message)
```

Because we have defined spatial weights using uID, we can use `dist400` generated on buildings and use it on tessellation:

[7]:
```
tessellation['neighbours400'] = momepy.Neighbors(tessellation, dist400, 'uID').series
```

```
100%|| 2005/2005 [00:00<00:00, 92447.50it/s]
```

[8]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='neighbours400', legend=True, cmap='Spectral_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



K nearest neighbor

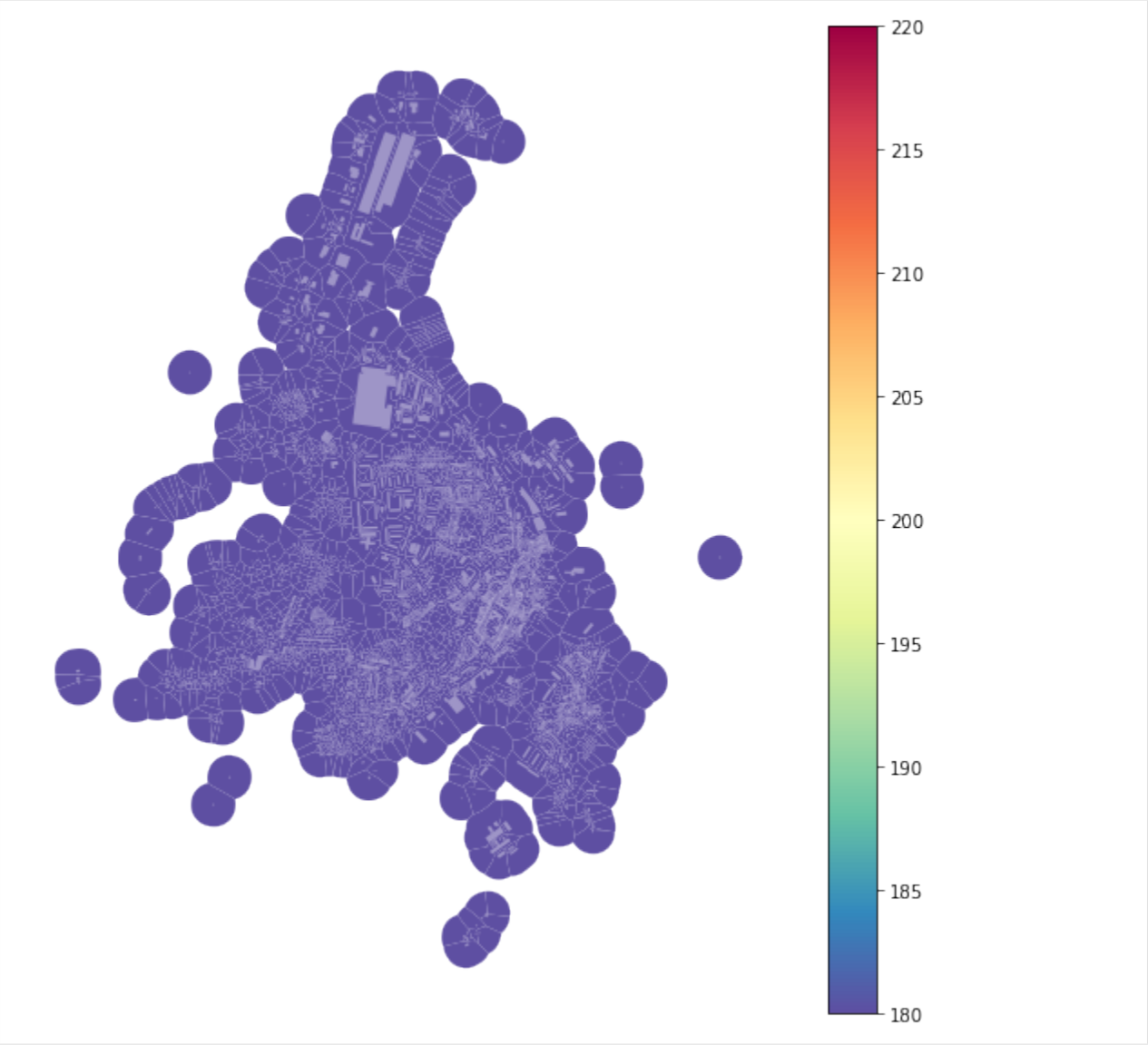
If we want fixed number of neighbours, we can use `libpysal.weights.KNN`:

[9]:
```
knn = libpysal.weights.KNN.from_dataframe(buildings, k=200, ids='uID')
tessellation['neighboursKNN'] = momepy.Neighbors(tessellation, knn,'uID').series
```

```
100%|| 2005/2005 [00:00<00:00, 46032.72it/s]
```

Note: As all tessellation cells have the same number of neighbours (due to KNN), they all have the same colour.

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='neighboursKNN', legend=True, cmap='Spectral_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



All of them can be used within morphometric analysis. Theoretical and practical differences are discussed in Fleischmann, Romice and Porta (2019).

For the other options on generating spatial weights see lipysal API.

Examples of usage

Spatial weights are used across `momepy`. This notebook will illustrate its use on three examples.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use `osmnx` to get the data for our example and after preprocessing of building layer will generate tessellation layer.

[2]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 2932/2932 [00:00<00:00, 7214.32it/s]
Changing geometry: 100%|| 630/630 [00:04<00:00, 148.53it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 2115/2115 [00:00<00:00, 10114.93it/s]
Changing geometry: 100%|| 172/172 [00:00<00:00, 185.18it/s]
```

```
Inward offset...
Discretization...
```

```
  6%|         | 114/2008 [00:00<00:01, 1133.81it/s]
```

```
Generating input point array...
```

```
100%|| 2008/2008 [00:01<00:00, 1907.59it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 249298/249298 [00:04<00:00, 53635.00it/s]
```

```
Dissolving Voronoi polygons...
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

```
0it [00:00, ?it/s]
```

First order contiguity

Distance to neighbours

To calculate the mean distance to neighbouring buildings, we need queen contiguity weights of the first order capturing the relationship between immediate neighbours. Relationship between buildings is here represented by relationships between their tessellation cells.

[3]:
```
sw1 = momepy.sw_high(k=1, gdf=tessellation, ids='uID')
```

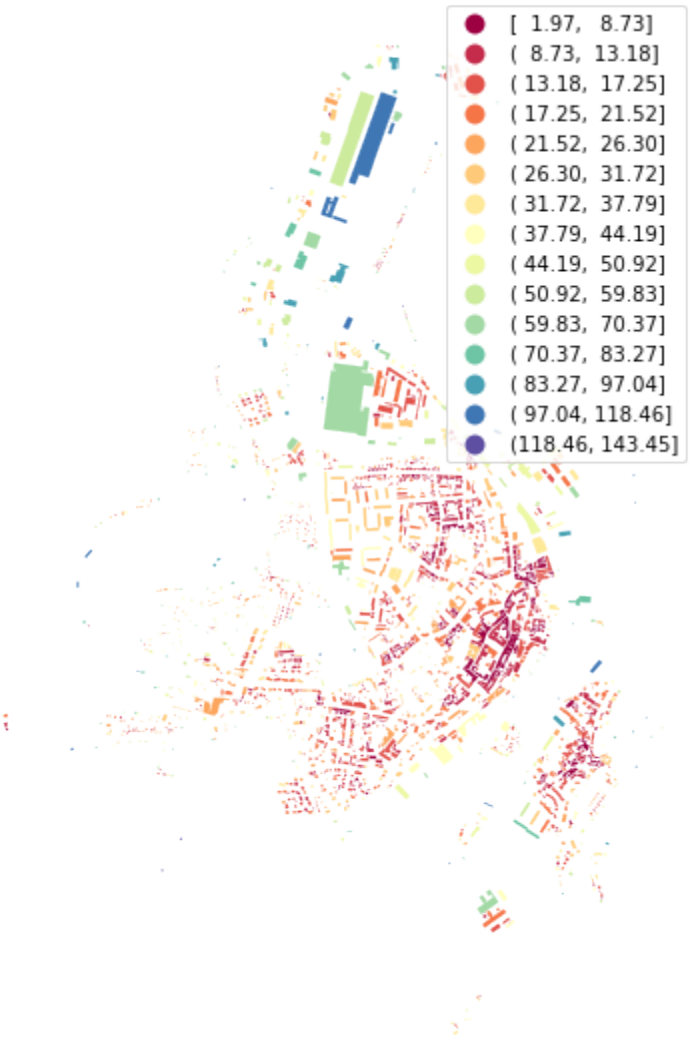
[4]:
```
buildings['neighbour_dist'] = momepy.NeighborDistance(buildings, sw1, 'uID').series
```

```
100%|| 2005/2005 [00:01<00:00, 1761.93it/s]
```

Note: If there is no neighbour for a building denoted in `spatial_weights`, the value is set to `np.nan`. In GeoPandas older than 0.7.0, rows with NaN have to be removed before plotting with natural breaks scheme.

[5]:
```
buildings = buildings.dropna(subset=['neighbour_dist'])
```

[6]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='neighbour_dist', scheme='naturalbreaks', k=15,␣
↪legend=True, cmap='Spectral')
ax.set_axis_off()
plt.show()
```



Higher order / distance

However, typical usage of spatial weights is to capture the vicinity of each feature. As illustrated in the *previous notebook*, there are multiple options on how to capture it. In this example, we will use queen contiguity of the higher order (3) based on morphological tessellation.

[7]:
```
sw3 = momepy.sw_high(k=3, gdf=tessellation, ids='uID')
```

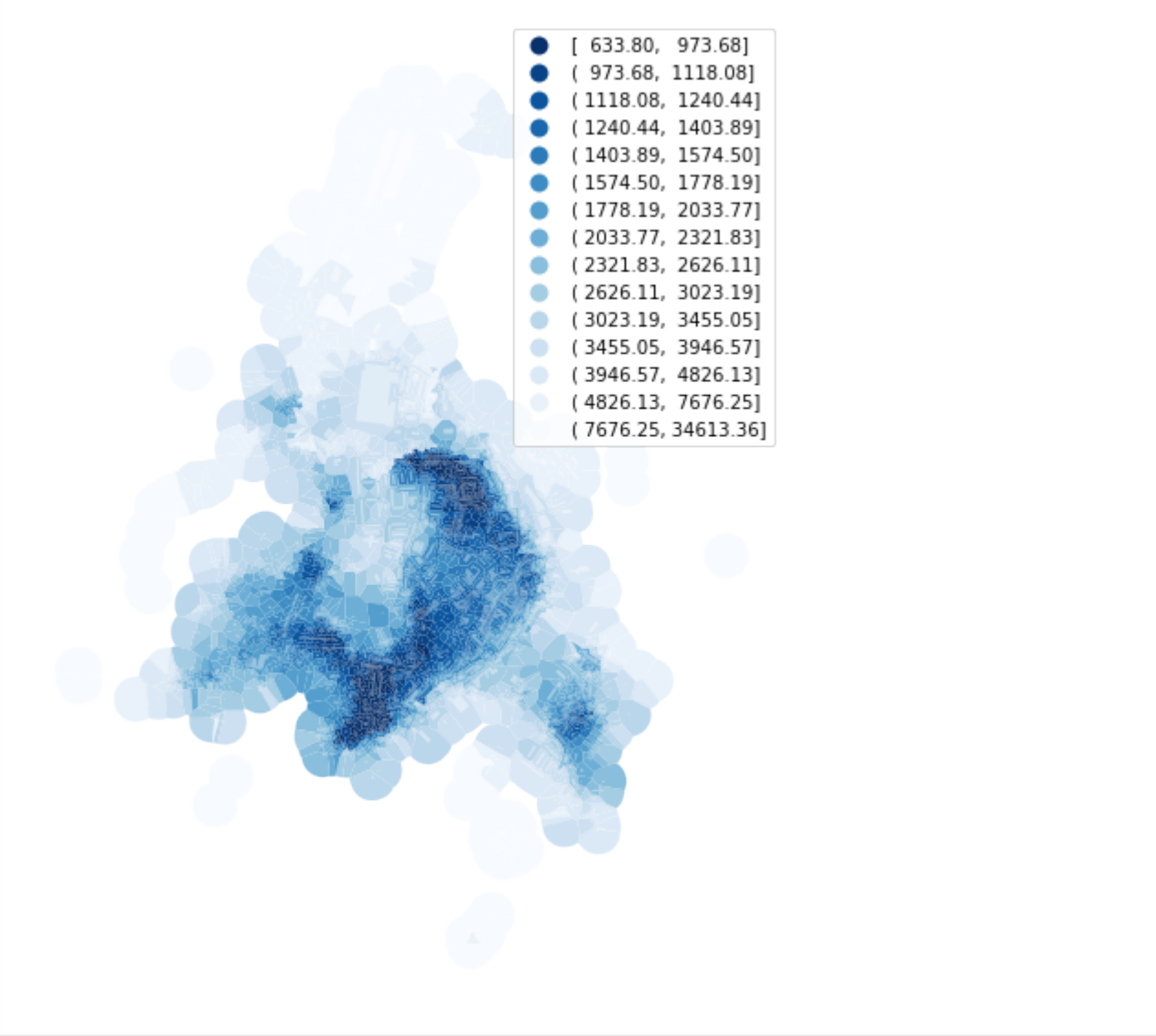
Average character

Mean value of selected character within a vicinity of each cell (or building, plot) is a simple example. `AverageCharacter` can measure mean, median or mode and defaults to `all`. Each of them can be accessed using `.mean`, `.median` or `.mode`. `.series` will return mean.

[8]:
```
areas = momepy.Area(tessellation).series
mean_area = momepy.AverageCharacter(
    tessellation, values=areas, spatial_weights=sw3, unique_id='uID')
tessellation['mean_area'] = mean_area.mean
```

```
100%|| 2005/2005 [00:01<00:00, 1901.77it/s]
```

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='mean_area', legend=True, scheme='quantiles', k=15,␣
↪cmap='Blues_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```

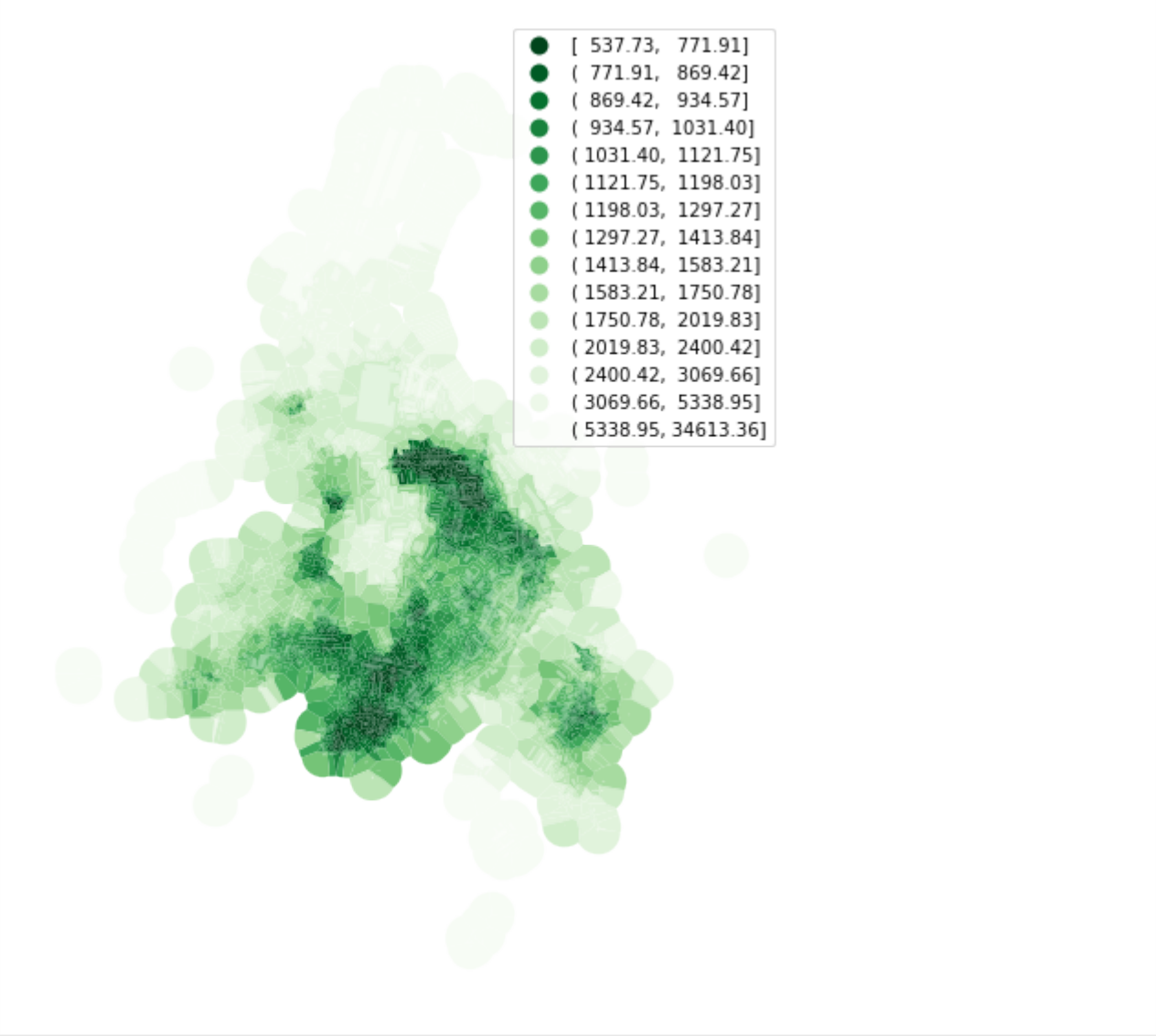


In some cases, we might want to eliminate the effect of outliers. To do so, we can specify the range on which should `AverageCharacter` calculate mean. Below we will measure only interquartile mean.

[10]:
```
tessellation['mean_area_iq'] = momepy.AverageCharacter(
    tessellation, areas, sw3, 'uID', rng=(25, 75)).mean
```

```
100%|| 2005/2005 [00:01<00:00, 1603.13it/s]
```

[11]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='mean_area_iq', legend=True, scheme='quantiles', k=15,
↪ cmap='Greens_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```

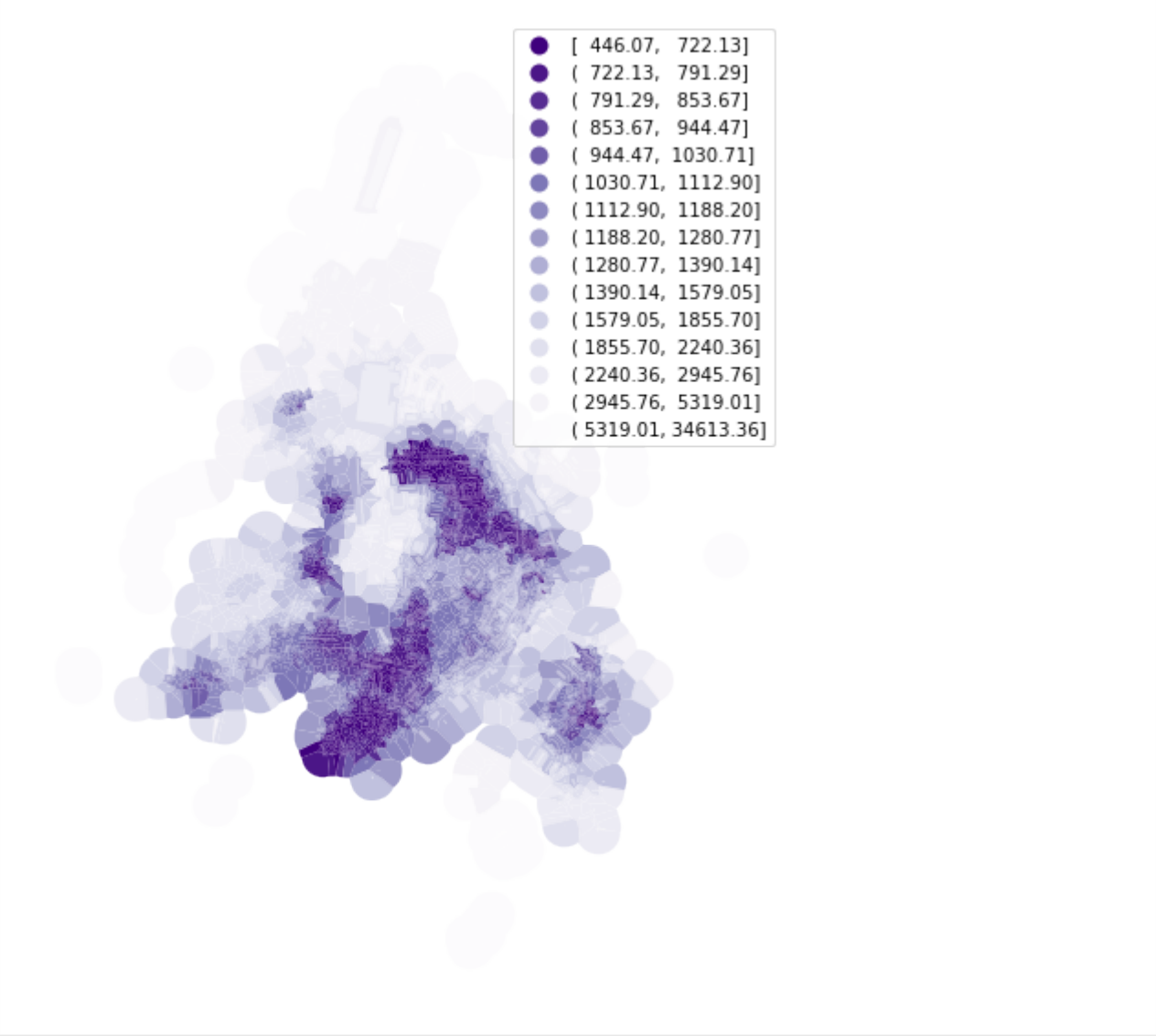


Another option would be to calculate median only:

[12]:
```
tessellation['med_area'] = momepy.AverageCharacter(
    tessellation, areas, sw3, 'uID', mode='median').median
```

```
100%|| 2005/2005 [00:00<00:00, 3063.14it/s]
```

[13]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='med_area', legend=True, scheme='quantiles', k=15,␣
↪cmap='Purples_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



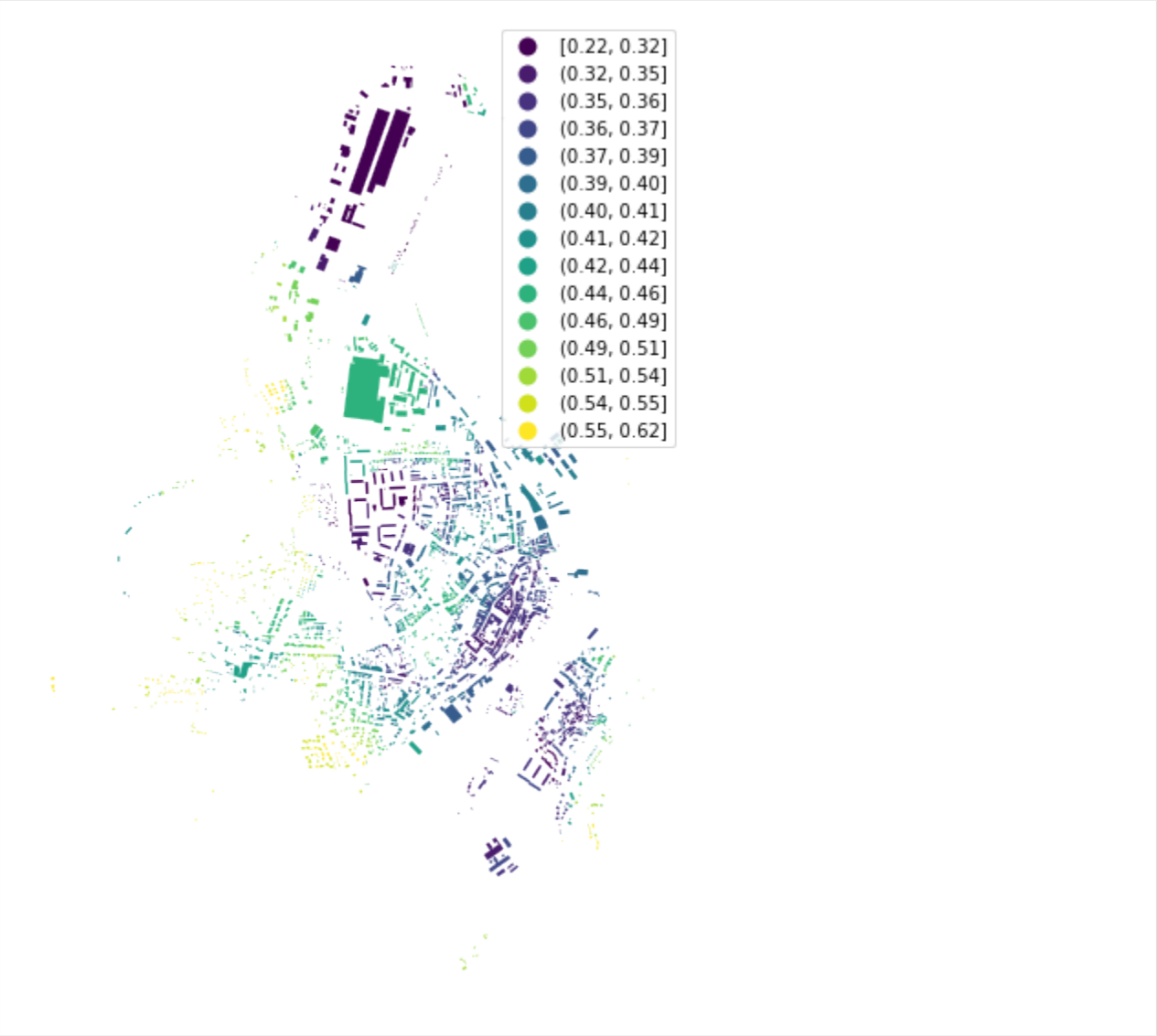
Weighted character

The weighted average is another example using the same spatial weights. For illustration, we can try area-weighted circular compactness:

[14]:
```
circular_compactness = momepy.CircularCompactness(buildings)
buildings['weighted_circom'] = momepy.WeightedCharacter(
    buildings, circular_compactness.series, sw3, 'uID', momepy.Area(buildings).
↪series).series
```

```
100%|| 2003/2003 [00:00<00:00, 2473.83it/s]
```

[15]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='weighted_circom', legend=True, scheme='quantiles', k=15,
↪ cmap='viridis')
ax.set_axis_off()
plt.show()
```



Density

We will again use our Manhattan case study to illustrate `Density`.

[16]:
```
point = (40.731603, -73.977857)
dist = 1000
gdf = ox.geometries.geometries_from_point(point, dist=dist, tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)
gdf_projected = gdf_projected[gdf_projected.geom_type.isin(['Polygon', 'MultiPolygon
↪'])]

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

(continues on next page)

(continued from previous page)

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 3200/3200 [00:01<00:00, 2176.30it/s]
Changing geometry: 100%|| 2157/2157 [00:18<00:00, 115.38it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 1459/1459 [00:00<00:00, 2216.25it/s]
Changing geometry: 100%|| 863/863 [00:05<00:00, 164.14it/s]
```

```
Inward offset...
Discretization...
```

```
  8%|         | 70/833 [00:00<00:01, 692.05it/s]
```

```
Generating input point array...
```

```
100%|| 833/833 [00:01<00:00, 583.48it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 356220/356220 [00:06<00:00, 54018.76it/s]
```

```
Dissolving Voronoi polygons...
```

```
100%|| 3/3 [00:00<00:00, 681.04it/s]
```

```
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

[17]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```

To get gross density, we need to know floor areas:

[18]:
```
buildings['height'] = buildings['height'].fillna(0).astype(float)
buildings['floor_area'] = momepy.FloorArea(buildings, 'height').series
```

Now we merge floor areas to tessellation based on shared unique ID and generate spatial weights.

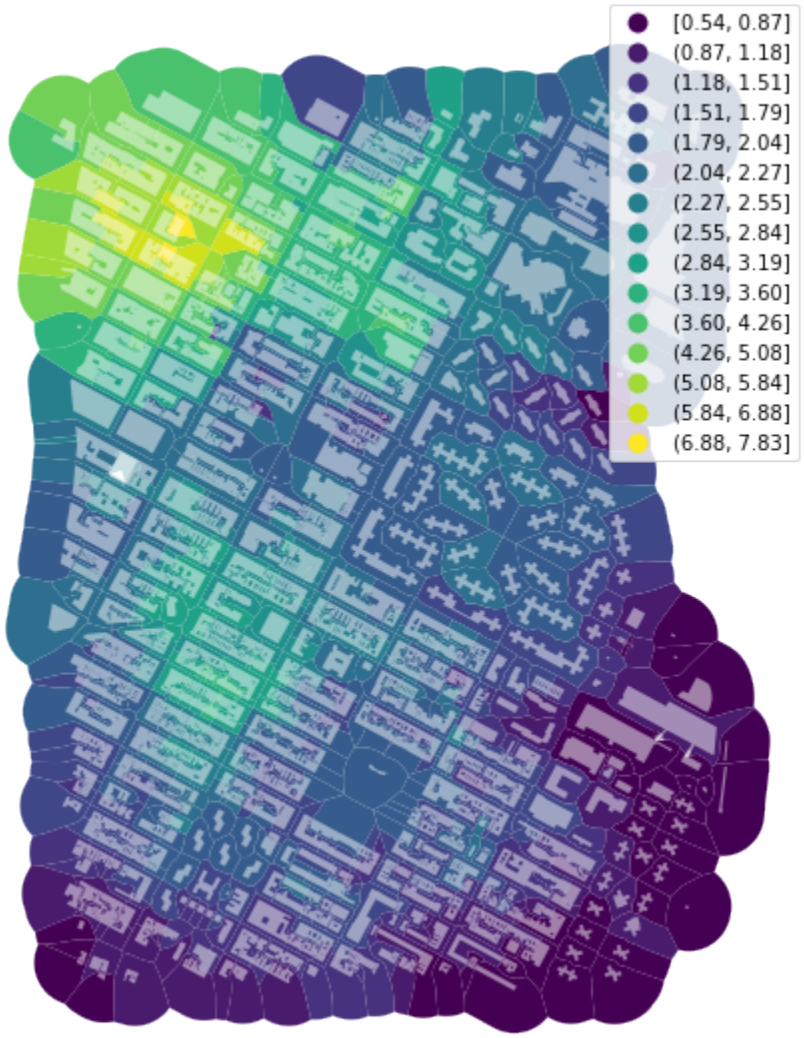
[19]:
```
tessellation = tessellation.merge(buildings[['uID', 'floor_area']])
sw = momepy.sw_high(k=3, gdf=tessellation, ids='uID')
```

`Density` is then following the same principle as illustrated above.

[20]:
```
gross = momepy.Density(
    tessellation, values='floor_area', spatial_weights=sw, unique_id='uID')
tessellation['gross_density'] = gross.series
```

```
100%|| 833/833 [00:00<00:00, 1148.61it/s]
```

[21]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='gross_density', legend=True, scheme='naturalbreaks',␣
↪k=15)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```



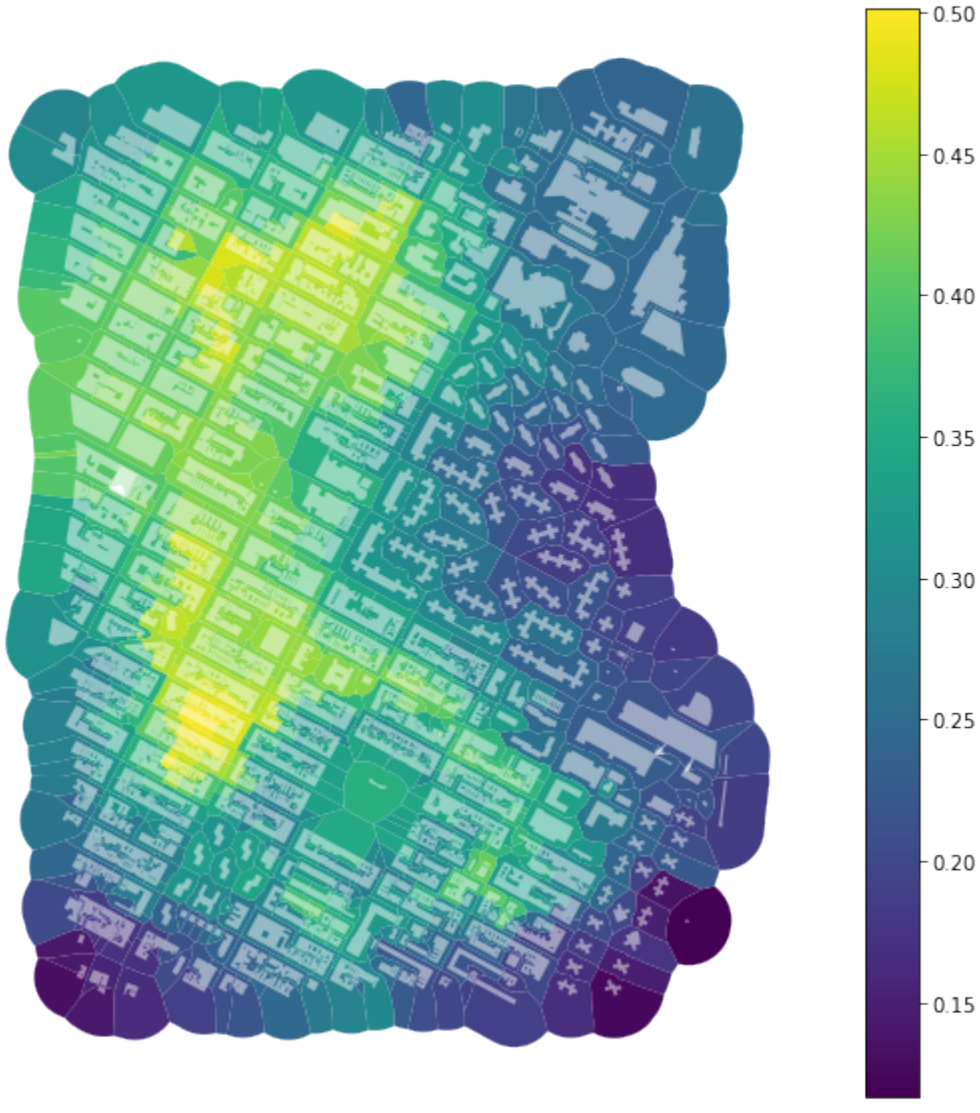
In a similar way can be done gross coverage.

[22]:
```
buildings['area'] = momepy.Area(buildings).series
tessellation = tessellation.merge(buildings[['uID', 'area']])
```

[23]:
```
coverage = momepy.Density(
    tessellation, values='area', spatial_weights=sw, unique_id='uID')
tessellation['gross_coverage'] = coverage.series
```

```
100%|| 833/833 [00:00<00:00, 1022.98it/s]
```

[24]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='gross_coverage', legend=True)
buildings.plot(ax=ax, color='white', alpha=.5)
ax.set_axis_off()
plt.show()
```



Measuring diversity

As important diversity is for cities, as complicated is to capture it. `momepy` offers several options on how to do that using urban morphometrics. Generally, we can distinguish three types of diversity characters, based on:

1. Absolute values
2. Relative values
3. Categorization (binning)

This notebook provides examples from each of them.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use `osmnx` to get the data for our example and after preprocessing of building layer will generate tessellation.

[2]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit).
↪tessellation
```

```
Loop 1 out of 2.
```

```
Identifying changes: 100%|| 2932/2932 [00:00<00:00, 7106.34it/s]
Changing geometry: 100%|| 630/630 [00:04<00:00, 152.46it/s]
```

```
Loop 2 out of 2.
```

```
Identifying changes: 100%|| 2115/2115 [00:00<00:00, 16914.35it/s]
Changing geometry: 100%|| 172/172 [00:01<00:00, 170.48it/s]
```

```
Inward offset...
Discretization...
```

```
  6%|         | 111/2008 [00:00<00:01, 1109.29it/s]
```

```
Generating input point array...
```

```
100%|| 2008/2008 [00:01<00:00, 1824.22it/s]
```

```
Generating Voronoi diagram...
Generating GeoDataFrame...
```

```
Vertices to Polygons: 100%|| 249298/249298 [00:04<00:00, 54255.22it/s]
```

```
Dissolving Voronoi polygons...
Preparing limit for edge resolving...
Building R-tree...
Identifying edge cells...
Cutting...
```

```
0it [00:00, ?it/s]
```

Queen contiguity

Morphological tessellation allows using contiguity-based weights matrix. While `libpysal.weights.contiguity.Queen` will do a classic Queen contiguity matrix; it might not be enough to capture proper context. For that reason, we can use `momepy.sw_high` to capture all neighbours within set topological distance `k`. More in *Generating spatial weights*.

[3]:
```
sw3 = momepy.sw_high(k=3, gdf=tessellation, ids='uID')
```

To have a character whose diversity can be measured, we can use the area of tessellation.

[4]:
```
tessellation['area'] = momepy.Area(tessellation).series
```

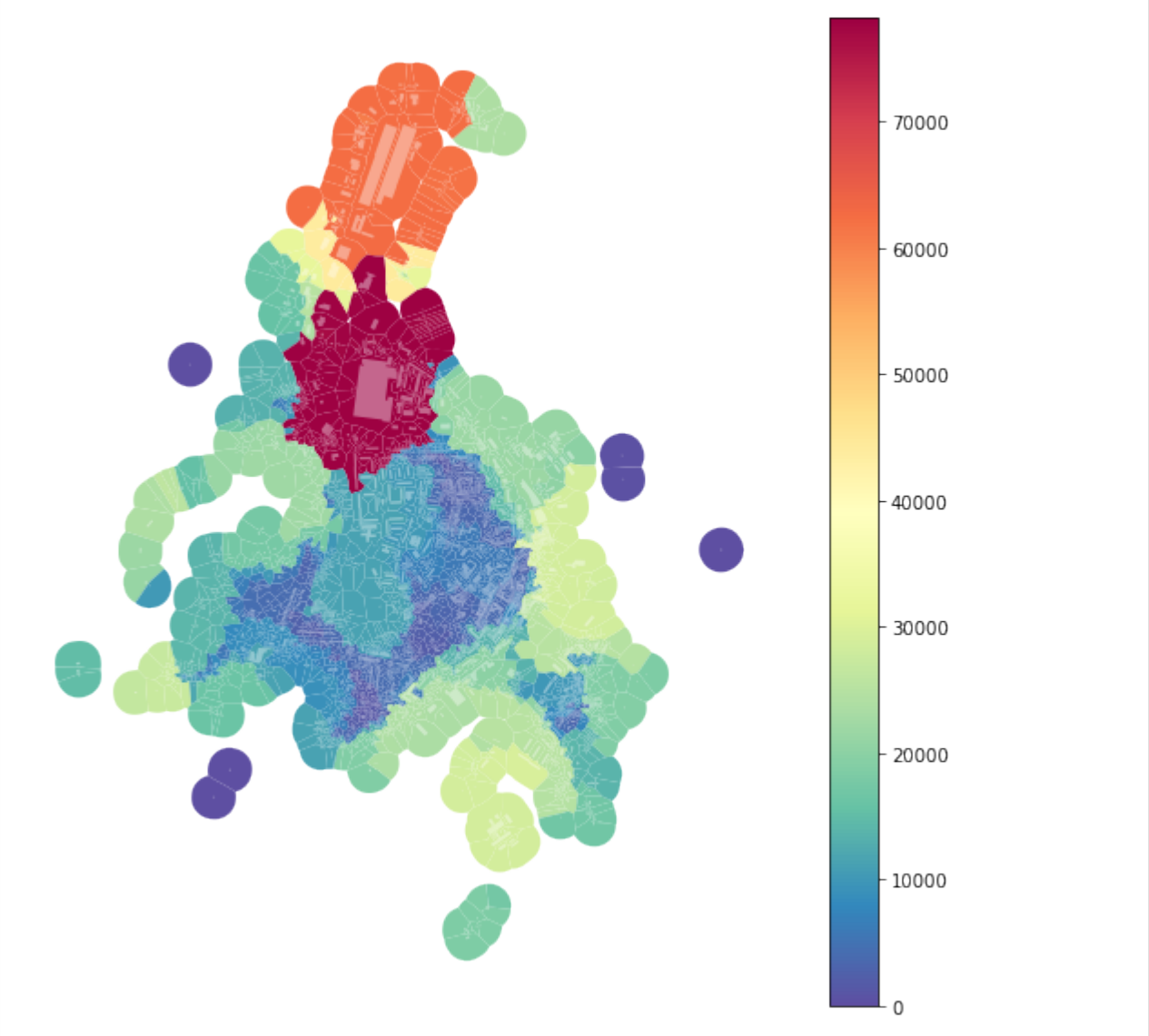
Range

The range is as simple as it sounds; it measures the range of the values withing all neighbours as captured by `spatial_weights`.

[5]:
```
area_rng = momepy.Range(tessellation, values='area',
                                        spatial_weights=sw3, unique_id='uID')
tessellation['area_rng'] = area_rng.series
```

```
100%|| 2005/2005 [00:00<00:00, 2617.70it/s]
```

[6]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_rng', legend=True, cmap='Spectral_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```

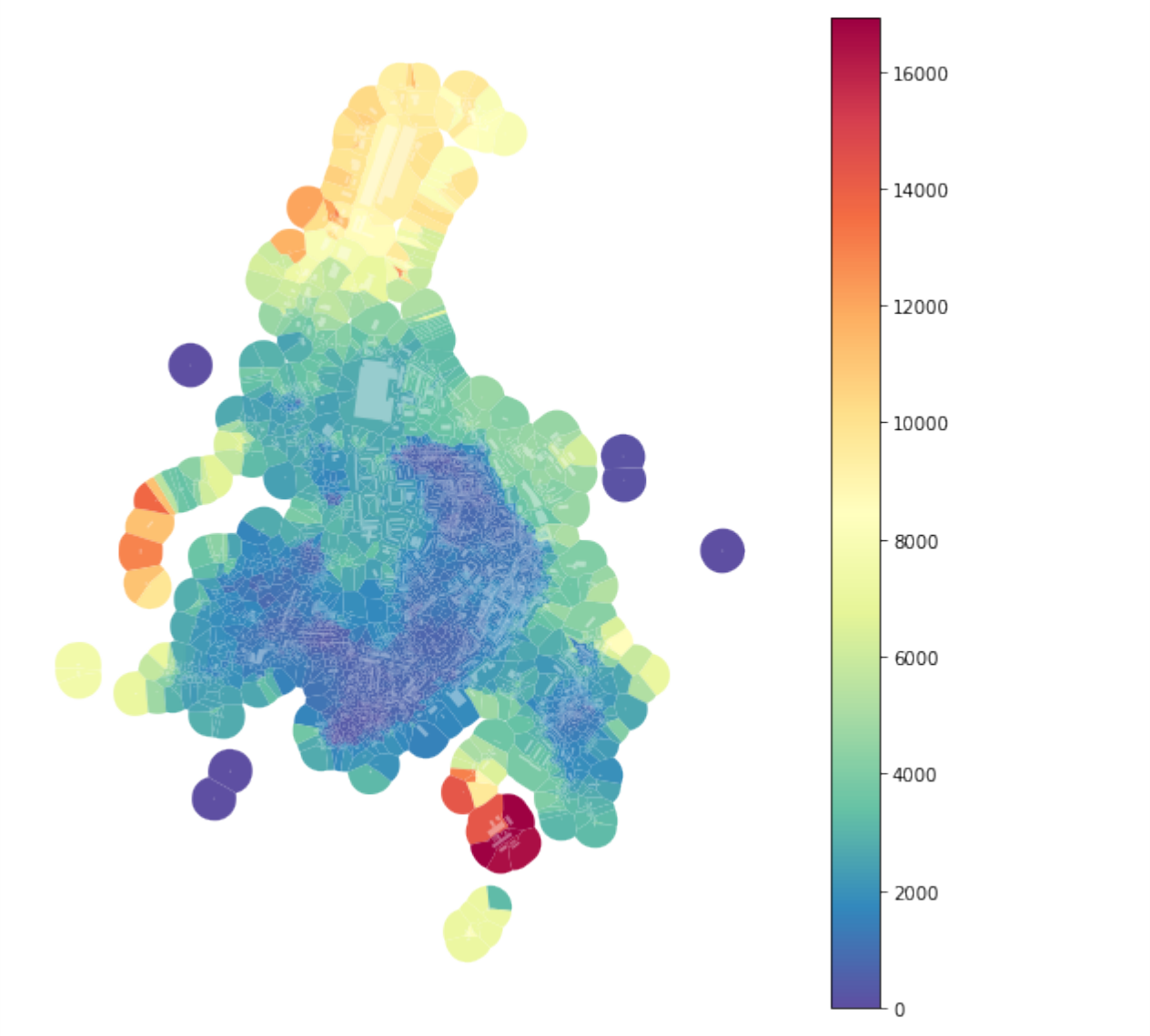


However, as we can see from the plot above, there is a massive effect of large-scale buildings, which can be seen as outliers. For that reason, we can define `rng` keyword argument to limit the range taken into account. To get the interquartile range:

[7]:
```
area_iqr = momepy.Range(tessellation, values='area',
                        spatial_weights=sw3, unique_id='uID',
                        rng=(25, 75))
tessellation['area_IQR'] = area_iqr.series
```

```
100%|| 2005/2005 [00:00<00:00, 2467.33it/s]
```

[8]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_IQR', legend=True, cmap='Spectral_r')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



The effect of outliers has been successfully eliminated.

Theil index

Theil index is a measure of inequality (as Gini index is). `momepy` is using `pysal`'s implementation of the Theil index to do the calculation.

[9]:
```
area_theil = momepy.Theil(tessellation, values='area',
                          spatial_weights=sw3,
                          unique_id='uID')
tessellation['area_Theil'] = area_theil.series
```

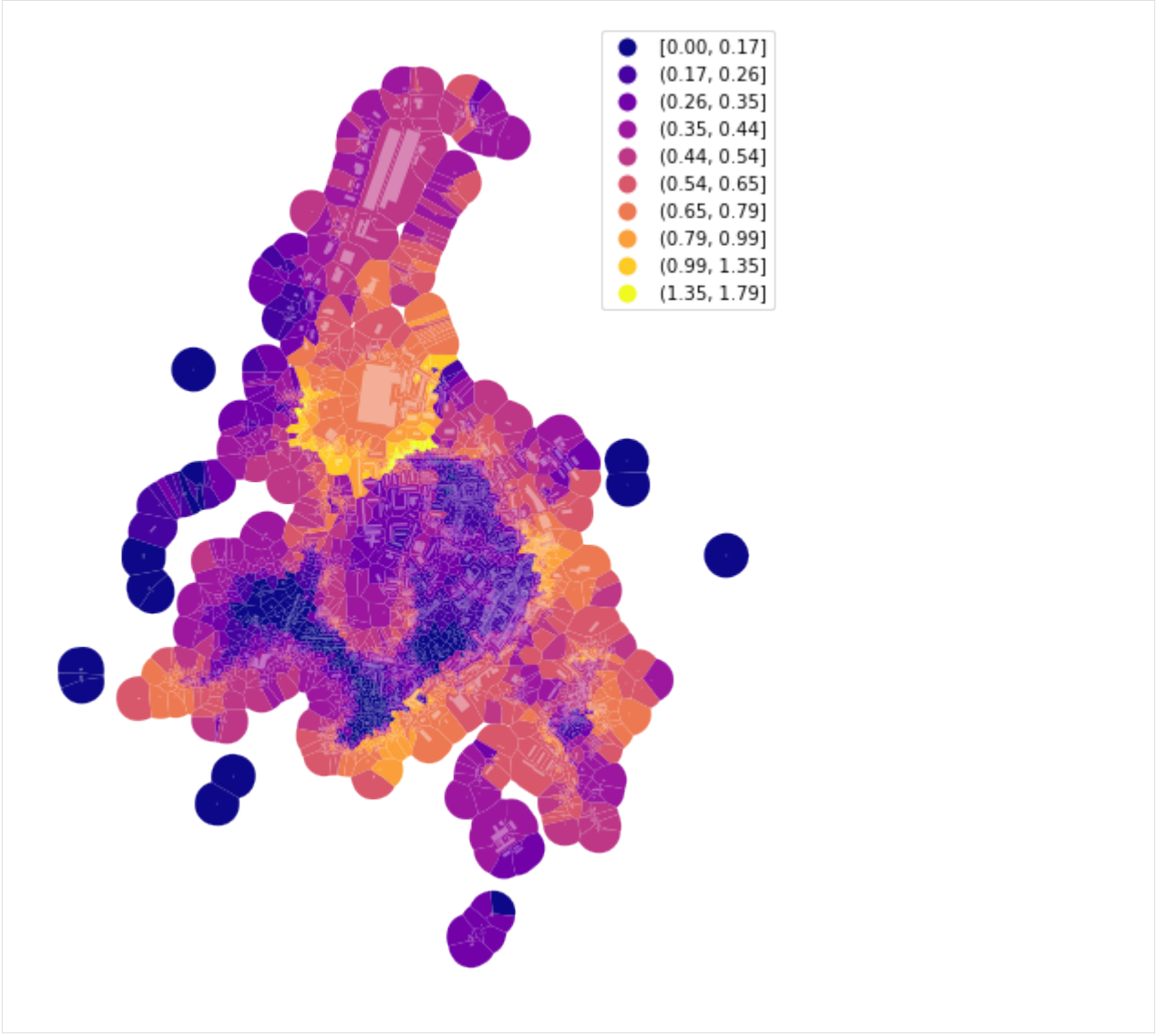
```
100%|| 2005/2005 [00:01<00:00, 1099.79it/s]
```

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_Theil', scheme='fisherjenks', k=10, legend=True,
↪ cmap='plasma')
```

(continues on next page)

(continued from previous page)

```
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



Again, the outlier effect is present. We can use the same keyword as above to limit it and measure the Theil index on the inter-decile range.

[11]:
```
area_id_theil = momepy.Theil(tessellation, values='area',
                             spatial_weights=sw3,
                             unique_id='uID',
                             rng=(10, 90))
tessellation['area_Theil_ID'] = area_id_theil.series
```

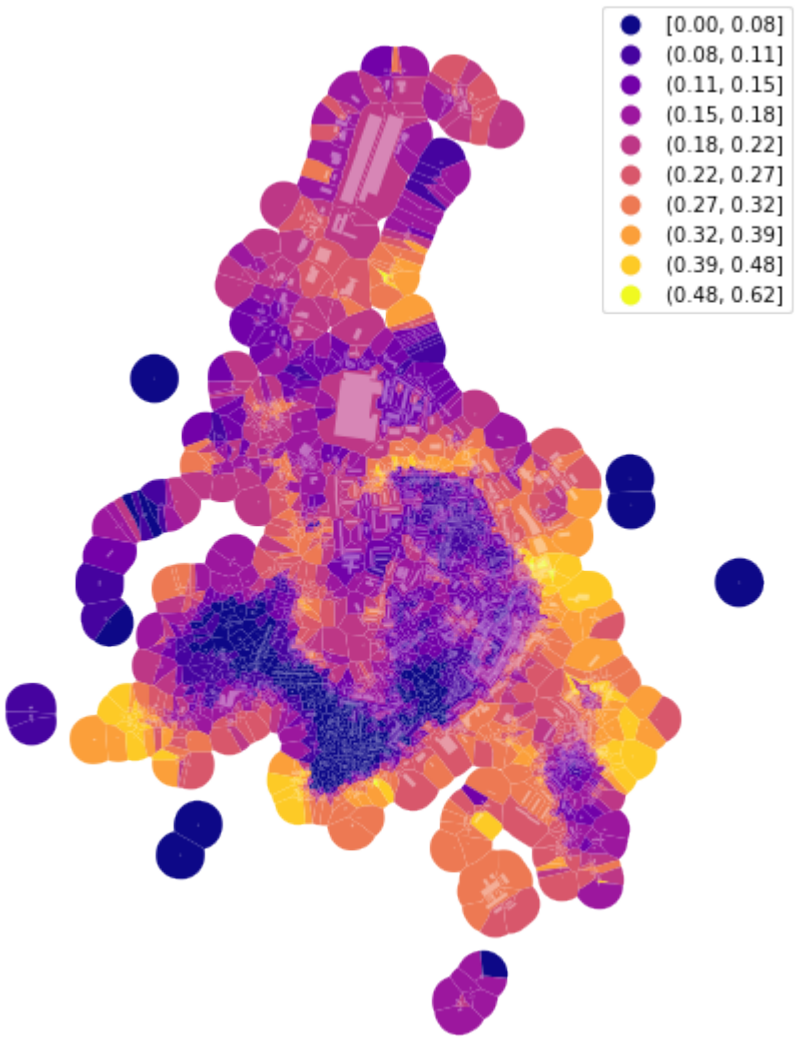
```
100%|| 2005/2005 [00:00<00:00, 2071.59it/s]
```

[12]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_Theil_ID', scheme='fisherjenks', k=10,␣
↪legend=True, cmap='plasma')
```

(continues on next page)

(continued from previous page)

```
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



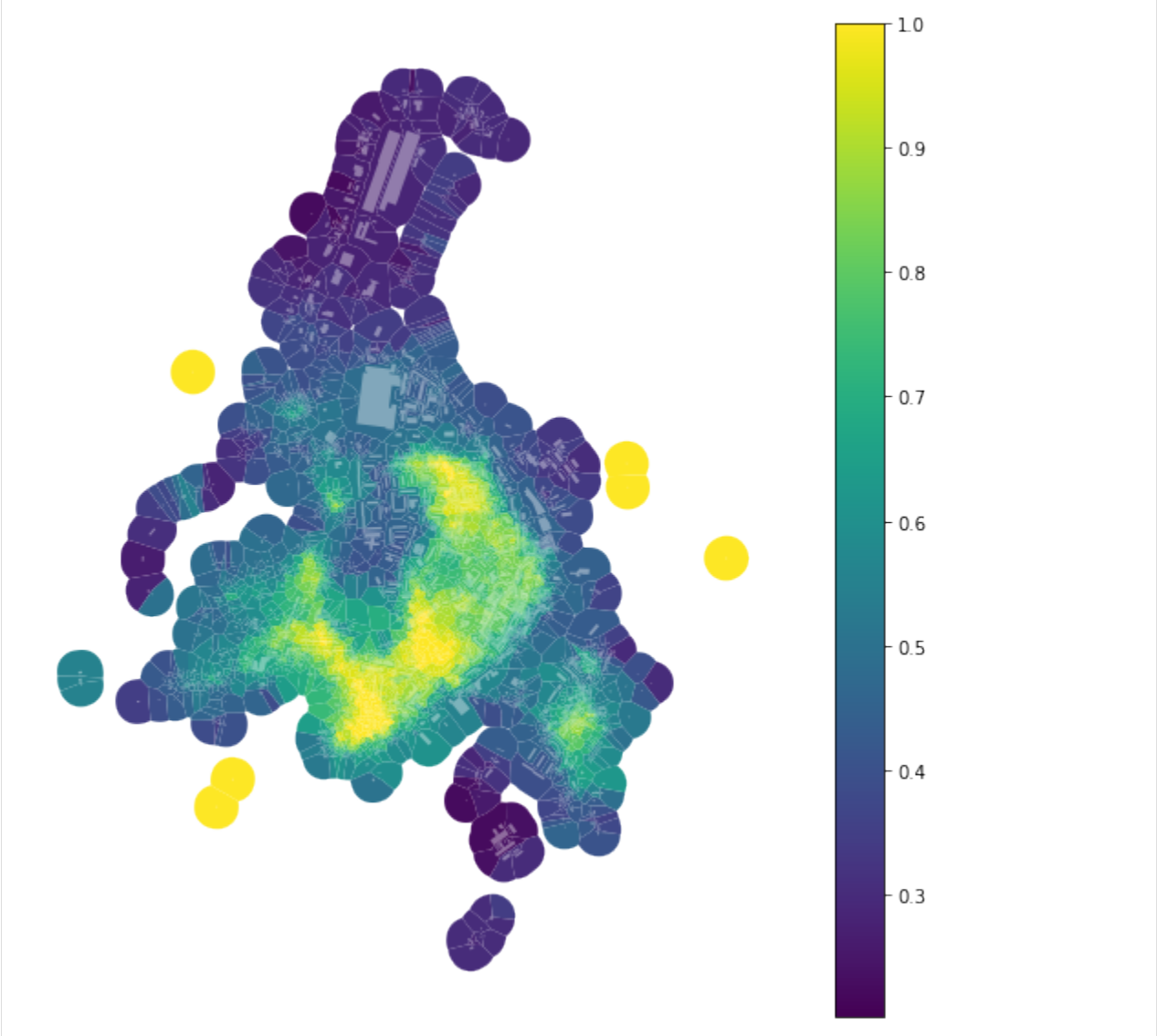
Simpson's diversity index

Simpson's diversity index is one of the most used indices capturing diversity. However, we need to be careful using it for continuous values, as it depends on the binning of these values to categories. The effect of different binning could be significant. `momepy` uses Head/tail Breaks as a large number of morphometric characters follows power-law distribution (for which Head/tail Breaks are designed). However, you can use any binning provided by `mapclassify` (including user-defined). The default Head/tail Breaks:

[13]:
```
area_simpson = momepy.Simpson(tessellation, values='area',
                              spatial_weights=sw3,
                              unique_id='uID')
tessellation['area_simpson'] = area_simpson.series
```

```
 10%|         | 208/2005 [00:00<00:03, 488.39it/s]/opt/miniconda3/envs/momepy_guide/
↪lib/python3.8/site-packages/mapclassify/classifiers.py:888: RuntimeWarning: invalid␣
↪value encountered in double_scalars
  gadf = 1 - self.adcm / adam
100%|| 2005/2005 [00:02<00:00, 932.39it/s]
```

[14]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_simpson', legend=True, cmap='viridis')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```

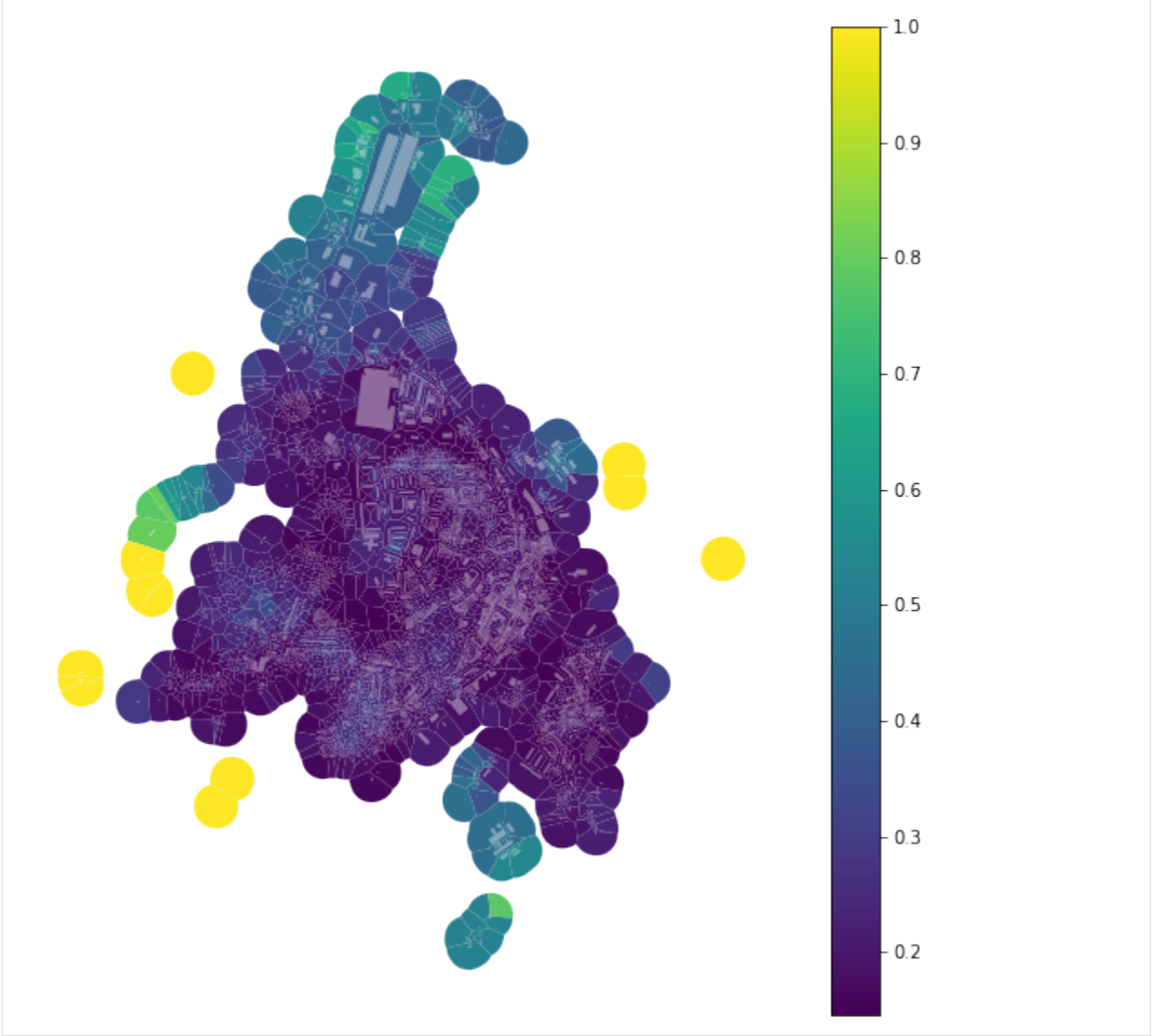


And binning based on quantiles (into 7 bins of equal size):

[15]:
```
tessellation['area_simpson_q7'] = momepy.Simpson(tessellation, values='area',
                                                 spatial_weights=sw3,
                                                 unique_id='uID',
                                                 binning='quantiles', k=7).series
```

```
  7%|         | 131/2005 [00:00<00:01, 1302.73it/s]/opt/miniconda3/envs/momepy_guide/
↪lib/python3.8/site-packages/mapclassify/classifiers.py:888: RuntimeWarning: invalid␣
↪value encountered in double_scalars
  gadf = 1 - self.adcm / adam
100%|| 2005/2005 [00:01<00:00, 1157.35it/s]
```

[16]:
```
f, ax = plt.subplots(figsize=(10, 10))
tessellation.plot(ax=ax, column='area_simpson_q7', legend=True, cmap='viridis')
buildings.plot(ax=ax, color="white", alpha=0.4)
ax.set_axis_off()
plt.show()
```



Always consider whether your binning is the optimal one.

Using two spatial weights matrices

Some functions are using spatial weights for two different purposes. Therefore two matrices have to be passed. We will illustrate this case measuring building adjacency and mean interbuilding distance.

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
```

We will again use `osmnx` to get the data for our example and after preprocessing of building layer will generate tessellation.

[2]:
```
import osmnx as ox

gdf = ox.geometries.geometries_from_place('Kahla, Germany', tags={'building':True})
gdf_projected = ox.projection.project_gdf(gdf)

buildings = momepy.preprocess(gdf_projected, size=30,
                              compactness=True, islands=True, verbose=False)
buildings['uID'] = momepy.unique_id(buildings)
limit = momepy.buffered_limit(buildings)
tessellation = momepy.Tessellation(buildings, unique_id='uID', limit=limit,
                                   verbose=False).tessellation
```

Building adjacency

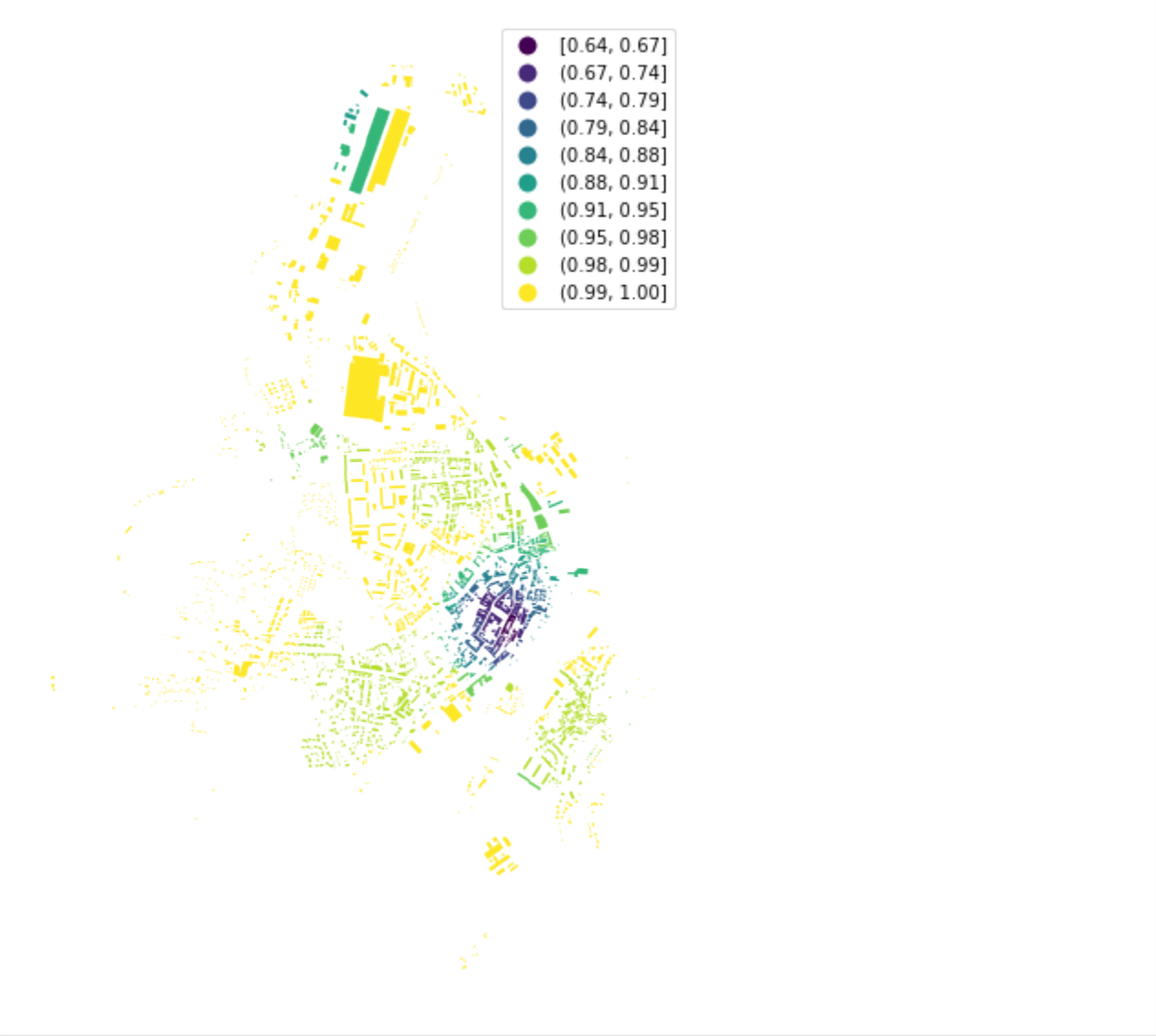
Building adjacency is using `spatial_weights_higher` to denote the area within which the calculation occurs (required) and `spatial_weights` to denote adjacency of buildings (optional, the function can do it for us). We can use distance band of 200 meters to define `spatial_weights_higher`.

[3]:
```
import libpysal
dist200 = libpysal.weights.DistanceBand.from_dataframe(buildings, 200,
                                                       ids='uID')
```

[4]:
```
adjac = momepy.BuildingAdjacency(
    buildings, spatial_weights_higher=dist200, unique_id='uID')
buildings['adjacency'] = adjac.series
```

```
Calculating adjacency: 100%|| 2005/2005 [00:00<00:00, 98379.52it/s]Calculating␣
↪spatial weights...
Spatial weights ready...
```

[5]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='adjacency', legend=True, cmap='viridis', scheme=
↪'naturalbreaks', k=10)
ax.set_axis_off()
plt.show()
```



If we want to specify or reuse `spatial_weights`, we can generate them as Queen contiguity weights. Using `libpysal` or `momepy` (momepy will use the same libpysal method, but you don't need to import libpysal directly):

[6]:
```
queen = libpysal.weights.Queen.from_dataframe(buildings,
                                              silence_warnings=True,
                                              ids='uID')
queen = momepy.sw_high(k=1, gdf=buildings, ids='uID', contiguity='queen')
```

[7]:
```
buildings['adj2'] = momepy.BuildingAdjacency(buildings,
                                             spatial_weights_higher=dist200,
                                             unique_id='uID',
                                             spatial_weights=queen).series
```

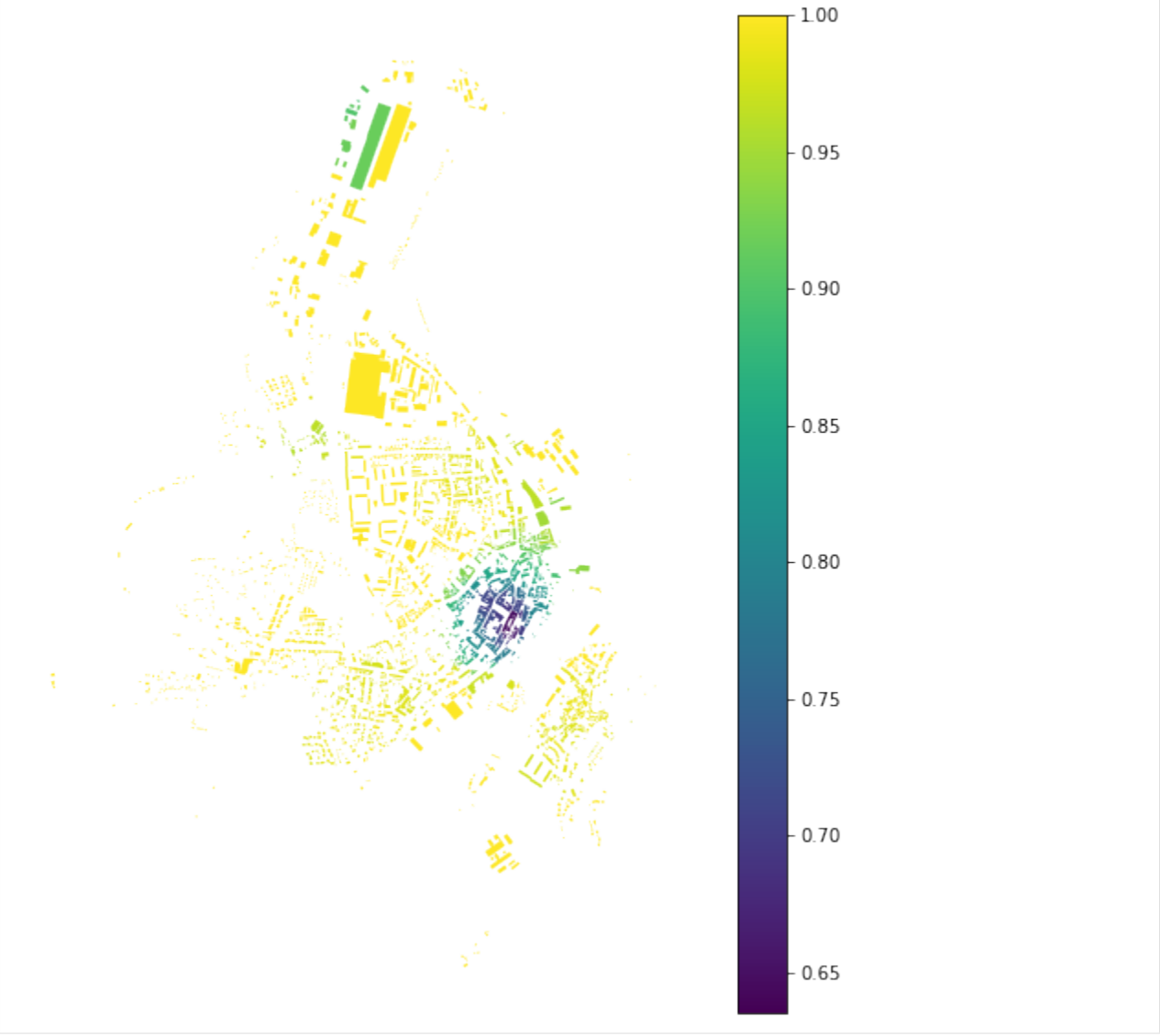
```
Calculating adjacency: 100%|| 2005/2005 [00:00<00:00, 86549.47it/s]
```

[8]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='adj2', legend=True, cmap='viridis')
ax.set_axis_off()
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Mean interbuilding distance

Mean interbuilding distance is similar to `neighbour_distance`, but it is calculated within vicinity defined in `spatial_weights_higher`, while `spatial_weights` captures immediate neighbours.

[9]:
```
sw1 = momepy.sw_high(k=1, gdf=tessellation, ids='uID')
sw3 = momepy.sw_high(k=3, gdf=tessellation, ids='uID')
```

[10]:
```
interblg_distance = momepy.MeanInterbuildingDistance(
    buildings, sw1, 'uID', spatial_weights_higher=sw3)
buildings['mean_ib_dist'] = interblg_distance.series
```

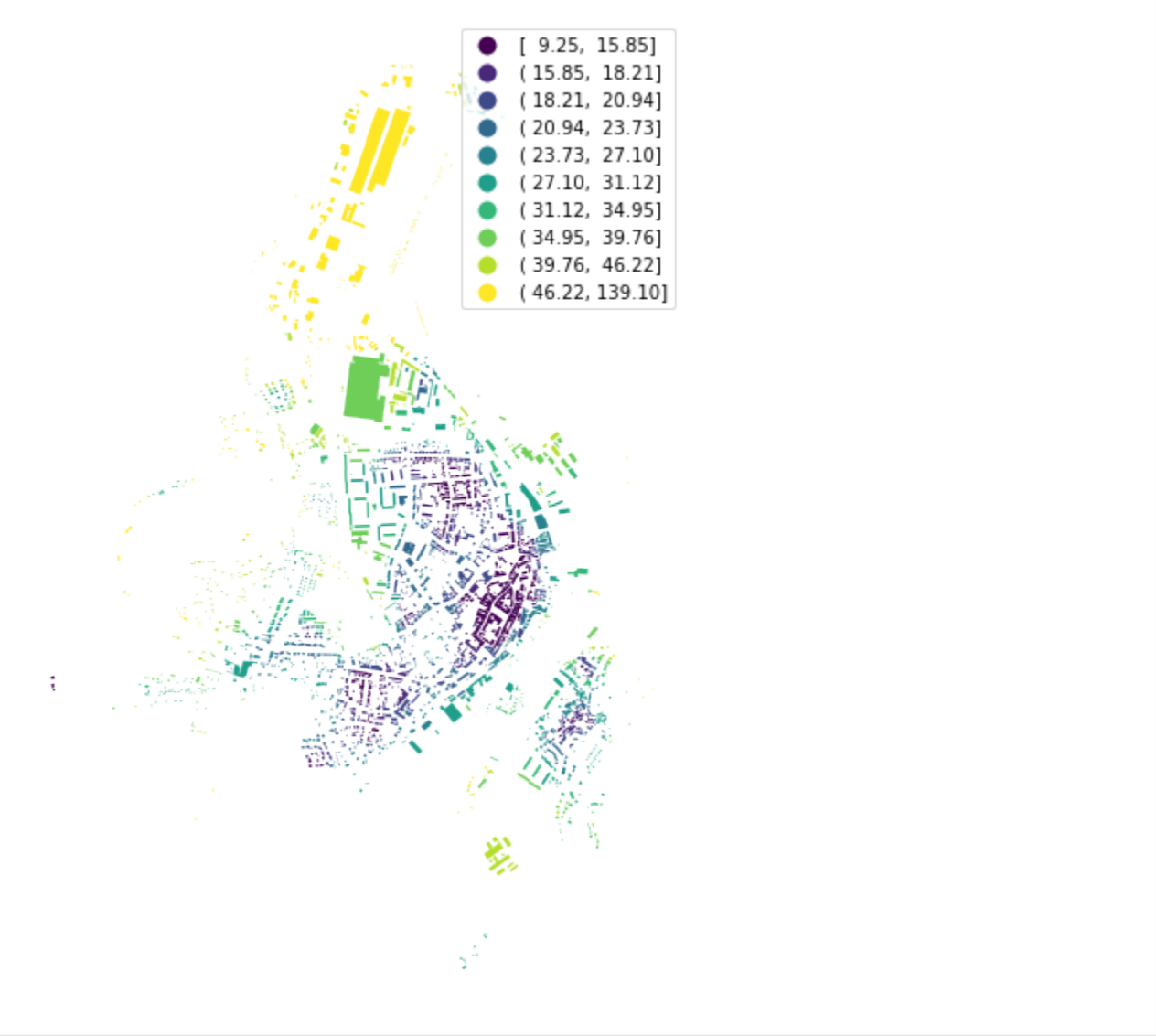
```
7%|         | 140/2005 [00:00<00:02, 717.84it/s]Computing mean interbuilding␣
↪distances...
100%|| 2005/2005 [00:02<00:00, 782.58it/s]
```

`spatial_weights_higher` is optional and can be derived from `spatial_weights` as weights of higher order defined in `order`.

[11]:
```
buildings['mean_ib_dist'] = momepy.MeanInterbuildingDistance(
    buildings, sw1, 'uID', order=3).series
```

```
Generating weights matrix (Queen) of 3 topological steps...
  4%|         | 85/2005 [00:00<00:02, 843.97it/s]Computing mean interbuilding␣
↪distances...
100%|| 2005/2005 [00:02<00:00, 820.91it/s]
```

[12]:
```
f, ax = plt.subplots(figsize=(10, 10))
buildings.plot(ax=ax, column='mean_ib_dist', scheme='quantiles', k=10, legend=True,␣
↪cmap='viridis')
ax.set_axis_off()
plt.show()
```



8.2.7 Network analysis

Part of the morphometric analysis is an analysis of street networks. While `momepy` at the moment focuses more on smaller-scale analysis, some of the network-based characters are included. The main difference here is that functions in the `graph` module, allowing this kind of study, are based on `networkx.Graph`, not `geopandas.GeoDataFrame`.

This section covers:

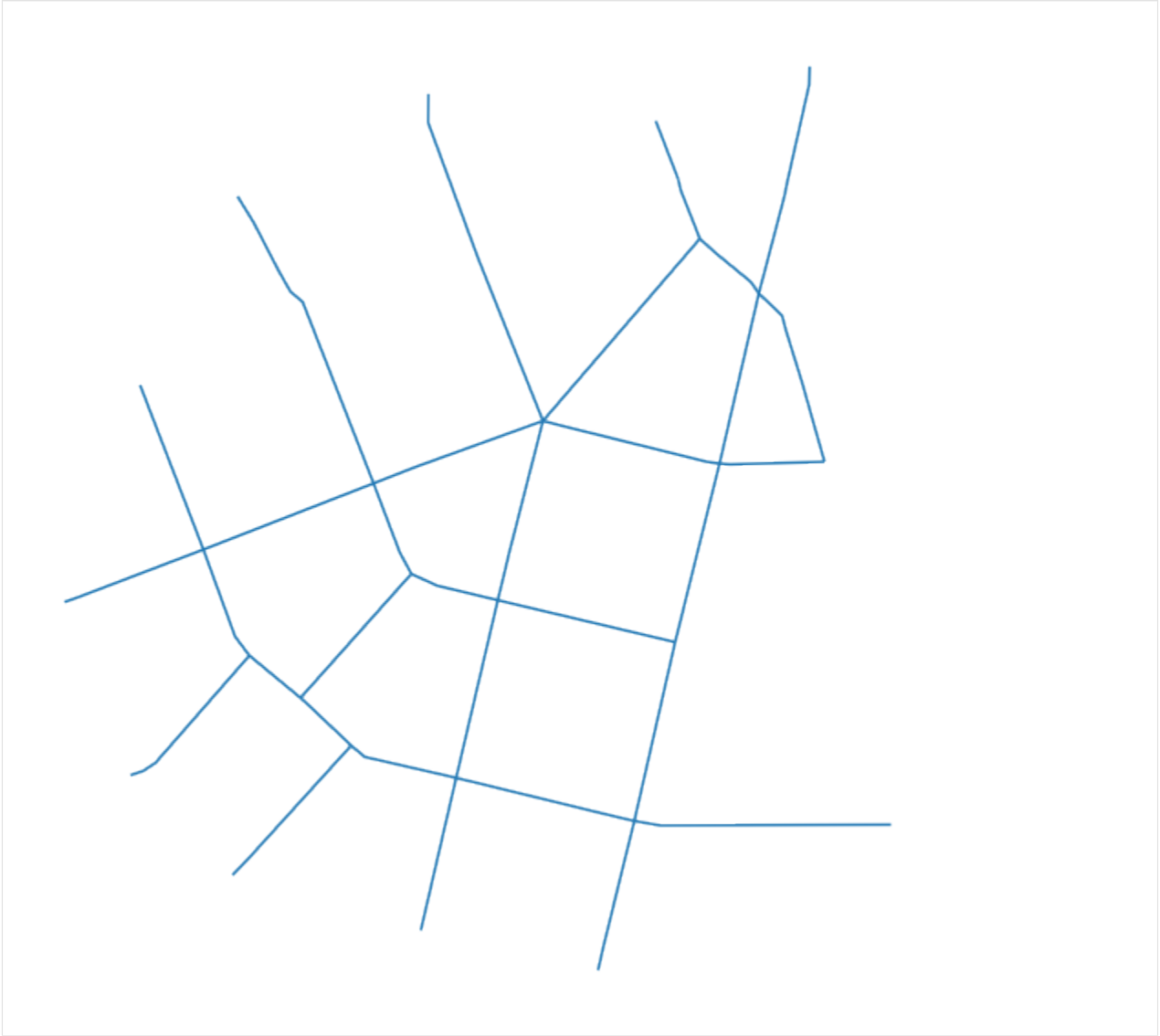
Converting from GeoDataFrame to Graph and back

The model situation expects to have all input data for analysis in `GeoDataFrames`, including street network (e.g. from shapefile).

[1]:
```
import momepy
import geopandas as gpd
import matplotlib.pyplot as plt
import networkx as nx
```

[2]:
```
streets = gpd.read_file(momepy.datasets.get_path('bubenec'), layer='streets')
```

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
streets.plot(ax=ax)
ax.set_axis_off()
plt.show()
```

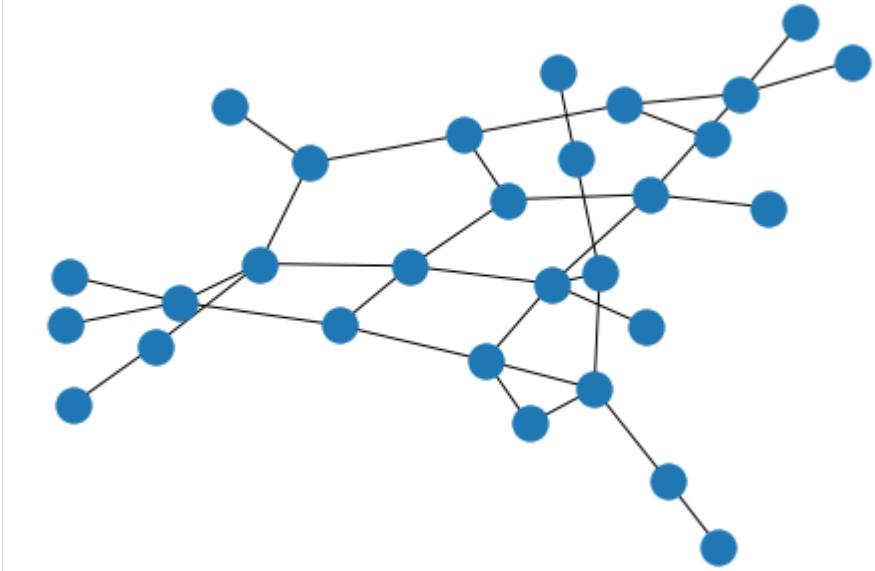


We have to convert this LineString GeoDataFrame to `networkx.Graph`. We use `momepy.gdf_to_nx` and later `momepy.nx_to_gdf` as pair of interconnected functions. `gdf_to_nx` supports both primal and dual graphs.

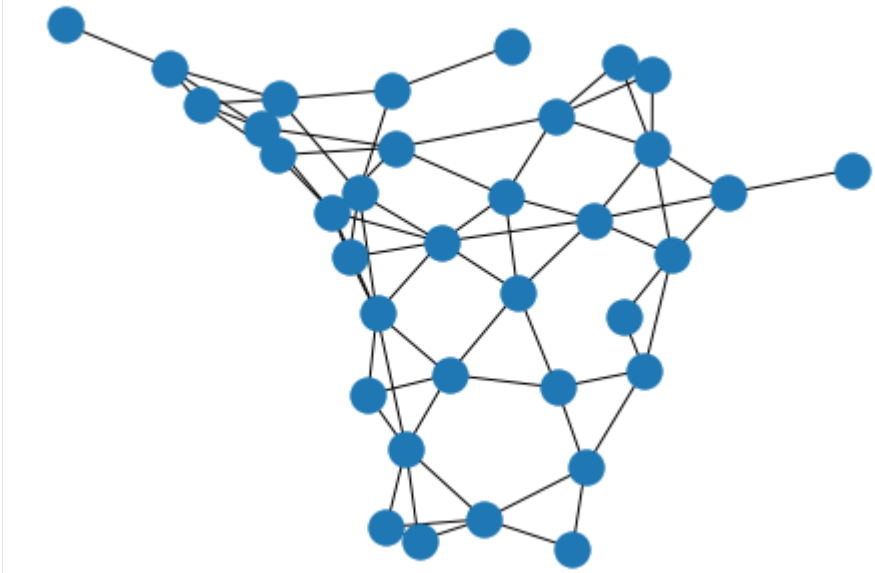
Primal approach will save length of each segment to be used as a weight later, while dual will save the angle between segments (allowing angular centrality).

[4]:
```
graph = momepy.gdf_to_nx(streets, approach='primal')
```

[5]:
```
nx.draw(graph)
```



[6]:
```
dual = momepy.gdf_to_nx(streets, approach='dual')
nx.draw(dual)
```



At this moment (almost) any `networkx` method can be used. For illustration, we will measure the node degree. Using `networkx`, we can do:

[7]:
```
degree = dict(nx.degree(graph))
nx.set_node_attributes(graph, degree, 'degree')
```

However, node degree is implemented in momepy so we can use directly:

[8]:
```
graph = momepy.node_degree(graph, name='degree')
```

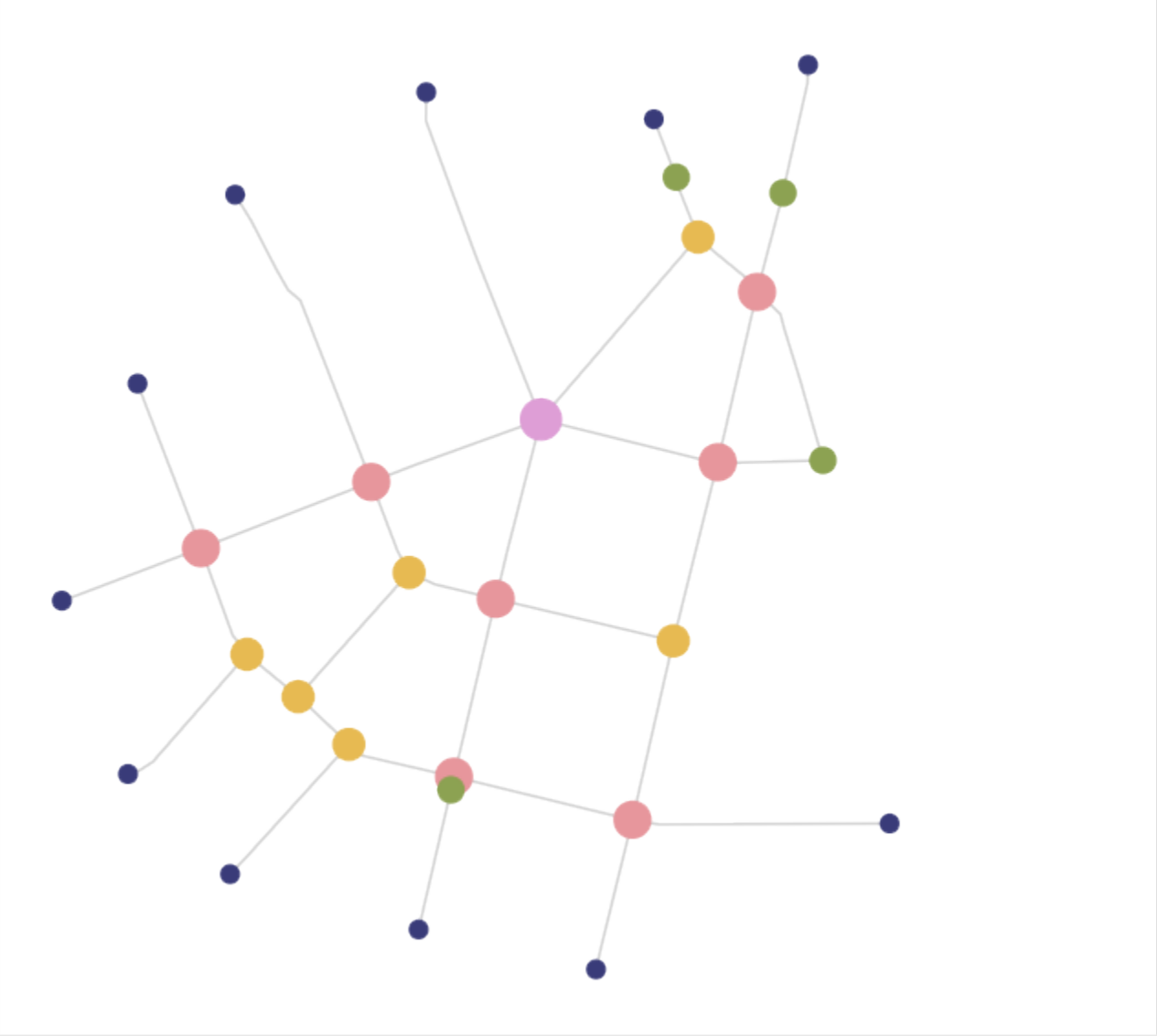
Once we have finished our network-based analysis, we want to convert the graph back to geodataframe. For that, we will use `momepy.nx_to_gdf`, which gives us several options what to export.

- `lines`
  - original LineString geodataframe
- `points`
  - point geometry representing street network intersections (nodes of primal graph)
- `spatial_weights`
  - spatial weights for nodes capturing their relationship within a network

Moreover, `edges` will contain `node_start` and `node_end` columns capturing the ID of both nodes at its ends.

[9]:
```
nodes, edges, sw = momepy.nx_to_gdf(graph, points=True, lines=True,
                                    spatial_weights=True)
```

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
nodes.plot(ax=ax, column='degree', cmap='tab20b', markersize=(nodes['degree'] * 100),␣
↪zorder=2)
edges.plot(ax=ax, color='lightgrey', zorder=1)
ax.set_axis_off()
plt.show()
```



[11]:
```
nodes.head(3)
```

[11]:
```
   degree  nodeID                        geometry
0       3       1  POINT (1603585.640 6464428.774)
1       5       2  POINT (1603413.206 6464228.730)
2       3       3  POINT (1603268.502 6464060.781)
```

[12]:
```
edges.head(3)
```

[12]:
```
                                            geometry      mm_len  node_start  \
0  LINESTRING (1603585.640 6464428.774, 1603413.2...  264.103950           1
1  LINESTRING (1603561.740 6464494.467, 1603564.6...   70.020202           1
2  LINESTRING (1603585.640 6464428.774, 1603603.0...   88.924305           1

   node_end
0         2
1         9
2         7
```

Street network analysis

Graph analysis offers three modes, of which the first two are used within `momepy` (as per v0.2): - node-based - value per node - edge-based - value per edge - network-based - single value per network

[1]:
```
import momepy
import geopandas as gpd
import osmnx as ox
import matplotlib.pyplot as plt
```

In this notebook, we will look at Písek, Czechia. We retrieve its network from OSM and convert it to GeoDataFrame:

[2]:
```
streets_graph = ox.graph_from_place('Pisek, Czechia', network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)

streets = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

Note: See the detailed explanation of these steps in the *centrality notebook*.

[3]:
```
f, ax = plt.subplots(figsize=(10, 10))
streets.plot(ax=ax, linewidth=0.2)
ax.set_axis_off()
plt.show()
```

We can generate networkX.MultiGraph, which is used within momepy for network analysis, using `gdf_to_nx`.

[4]:
```
graph = momepy.gdf_to_nx(streets)
```

Node-based analysis

Once we have the graph, we can use momepy functions, like the one measuring clustering:

[5]:
```
graph = momepy.clustering(graph, name='clustering')
```

Using sub-graph

Momepy includes local characters measured on the network within a certain radius from each node, like meshedness. The function will generate `ego_graph` for each node so that it might take a while for more extensive networks. Radius can be defined topologically:

[6]:
```
graph = momepy.meshedness(graph, radius=5, name='meshedness')
```
```
100%|| 543/543 [00:01<00:00, 449.74it/s]
```

Or metrically, using distance which has been saved as an edge argument by `gdf_to_nx` (or any other weight).

[7]:
```
graph = momepy.meshedness(graph, radius=400, name='meshedness400',
                          distance='mm_len')
```
```
100%|| 543/543 [00:00<00:00, 1324.80it/s]
```

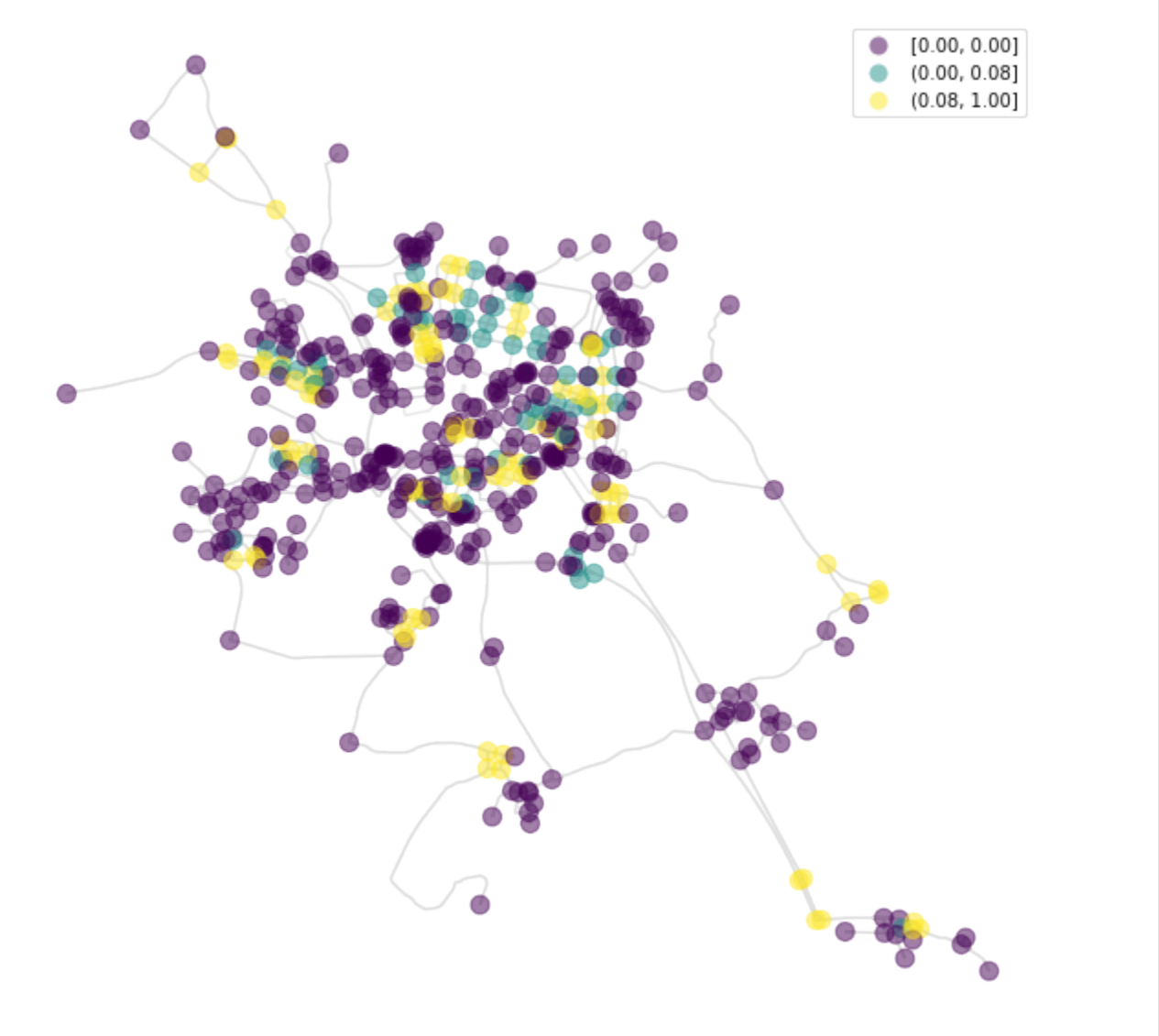
Once we have finished the graph-based analysis, we can go back to `GeoPandas`. In this notebook, we are interested in nodes only:

[8]:
```
nodes = momepy.nx_to_gdf(graph, points=True, lines=False, spatial_weights=False)
```

Now we can plot our results in a standard way, or link them to other elements (using `get_node_id`).

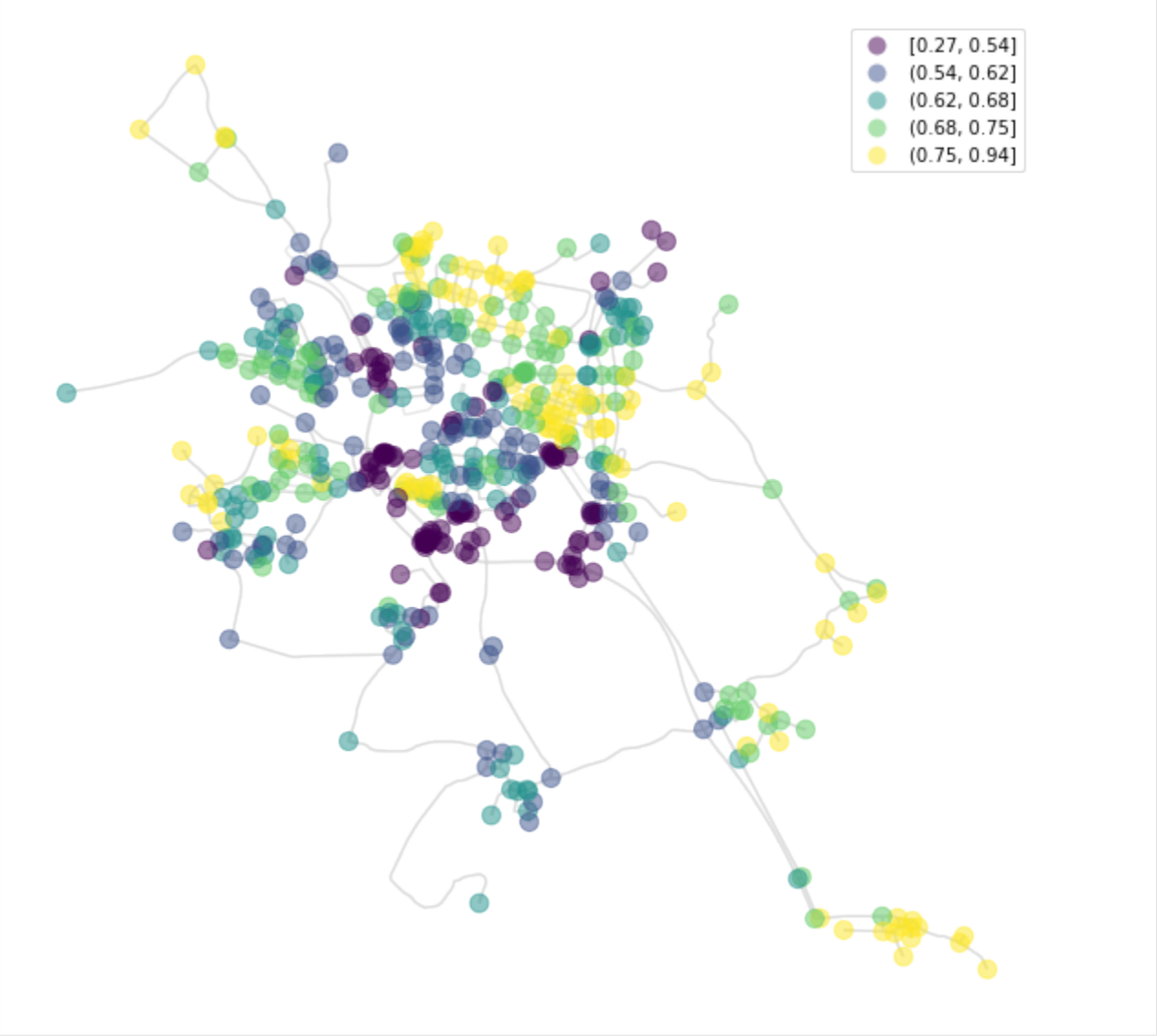
Clustering:

[9]:
```
f, ax = plt.subplots(figsize=(10, 10))
nodes.plot(ax=ax, column='clustering', markersize=100, legend=True, cmap='viridis',
           scheme='quantiles', alpha=0.5, zorder=2)
streets.plot(ax=ax, color='lightgrey', alpha=0.5, zorder=1)
ax.set_axis_off()
plt.show()
```
```
/opt/miniconda3/envs/momepy_guide/lib/python3.8/site-packages/mapclassify/classifiers.
↪py:235: UserWarning: Warning: Not enough unique values in array to form k classes
  Warn(
/opt/miniconda3/envs/momepy_guide/lib/python3.8/site-packages/mapclassify/classifiers.
↪py:238: UserWarning: Warning: setting k to 3
  Warn("Warning: setting k to %d" % k_q, UserWarning)
```



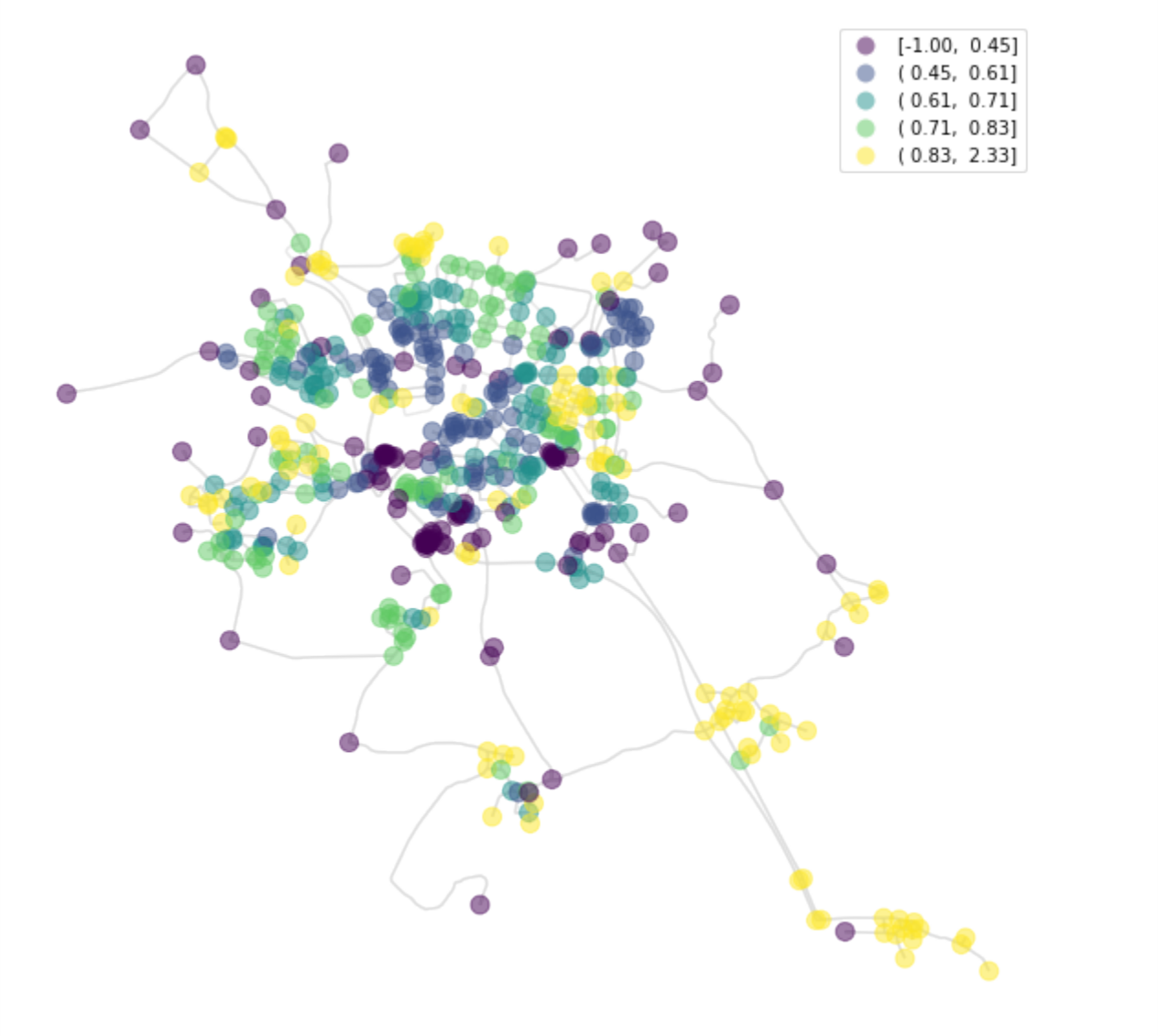
Meshedness based on topological distance:

[10]:
```
f, ax = plt.subplots(figsize=(10, 10))
nodes.plot(ax=ax, column='meshedness', markersize=100, legend=True, cmap='viridis',
           alpha=0.5, zorder=2, scheme='quantiles')
streets.plot(ax=ax, color='lightgrey', alpha=0.5, zorder=1)
ax.set_axis_off()
plt.show()
```



And meshedness based on 400 metres:

[11]:
```
f, ax = plt.subplots(figsize=(10, 10))
nodes.plot(ax=ax, column='meshedness400', markersize=100, legend=True, cmap='viridis',
           alpha=0.5, zorder=2, scheme='quantiles')
streets.plot(ax=ax, color='lightgrey', alpha=0.5, zorder=1)
ax.set_axis_off()
plt.show()
```



Multiple Centrality Assessment

Multiple Centrality Assessment (MCA) is an approach to street network analysis developed by Porta and Latora (2006). It's main aim is to understand the structure of street network of our cities from the perspective of the importance and position of each street/intersection within the whole network as expressed by various centralities. Momepy can do all types of MCA-based centrality analysis as were developed through the years.

The aim of this notebook is to illustrate how to measure different centralities using momepy. For the theoretical background, please refer to the work of Porta et al.

[1]:
```
import geopandas as gpd
import momepy
import osmnx as ox
import matplotlib.pyplot as plt
```

In the ideal case, momepy expects LineString GeoDataFrame containing street network as a starting point. Either we have our own, or we can use osmnx to dowload network from OSM. In this notebook, we will look at Vicenza, Italy.

[2]:
```
streets_graph = ox.graph_from_place('Vicenza, Vicenza, Italy', network_type='drive')
streets_graph = ox.projection.project_graph(streets_graph)
```

Code above dowloaded network from OSM and projected it. At this point, `streets_graph` is networkX Graph object, similar to the one we will use in momepy. In theory, you can use it directly. However, momepy when converting GeoDataFrame to network ensures that all attributes are set and compatible with morphometric functions, so we do recommend saving graph to gdf and let momepy do the conversion back to graph.

[3]:
```
edges = ox.graph_to_gdfs(streets_graph, nodes=False, edges=True,
                                    node_geometry=False, fill_edge_geometry=True)
```

[4]:
```
f, ax = plt.subplots(figsize=(10, 10))
edges.plot(ax=ax, linewidth=0.2)
ax.set_axis_off()
plt.show()
```

To measure centrality, we have to *convert* gdf to graph. For that, we can choose from two options how to represent street network within graph. First and the most straightforward one is the primal approach (Porta et al., 2006) where

street is represented by graph edge and intersection by node.

Primal graph

We can generate networkX.MultiGraph, which is used within momepy, using `gdf_to_nx`.

[5]:
```
primal = momepy.gdf_to_nx(edges, approach='primal')
```

Closeness centrality

Closeness centrality could be simplified as average distance to every other node from each node. As such, it can be measured on the whole network (Global Closeness Centrality) or wihtin certain reach only (Local Closeness Centrality).

Local closeness

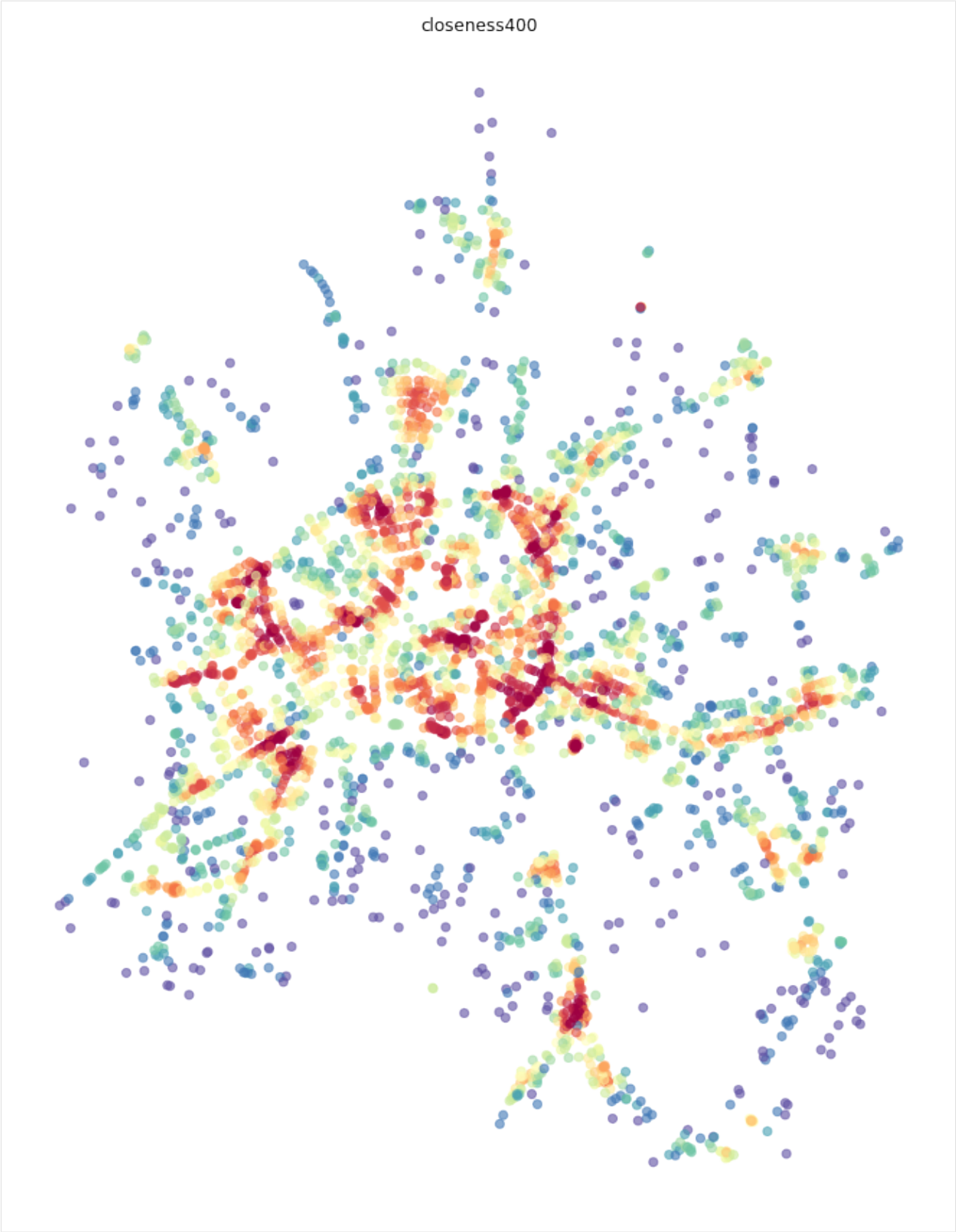
To measure local `closeness_centrality` we need to specify radius (how far we should go from each node). We can use topological distance (e.g. 5 steps, then radius=5) or metric distance (e.g. 400 metres) - then radius=400 and distance= lenght of each segment saved as a parameter of each edge. By default, momepy saves length as `mm_len`.

Weight parameter is used for centrality calculation. Again, we can use metric weight (using the same attribute as above) or no weight (weight=None) at all. Or any other attribute we wish.

[6]:
```
primal = momepy.closeness_centrality(primal, radius=400, name='closeness400',␣
↪distance='mm_len', weight='mm_len')
```

```
100%|| 4116/4116 [00:07<00:00, 562.53it/s]
```

[7]:
```
nodes = momepy.nx_to_gdf(primal, lines=False)
f, ax = plt.subplots(figsize=(15, 15))
nodes.plot(ax=ax, column='closeness400', cmap='Spectral_r', scheme='quantiles', k=15,␣
↪alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness400')
plt.show()
```

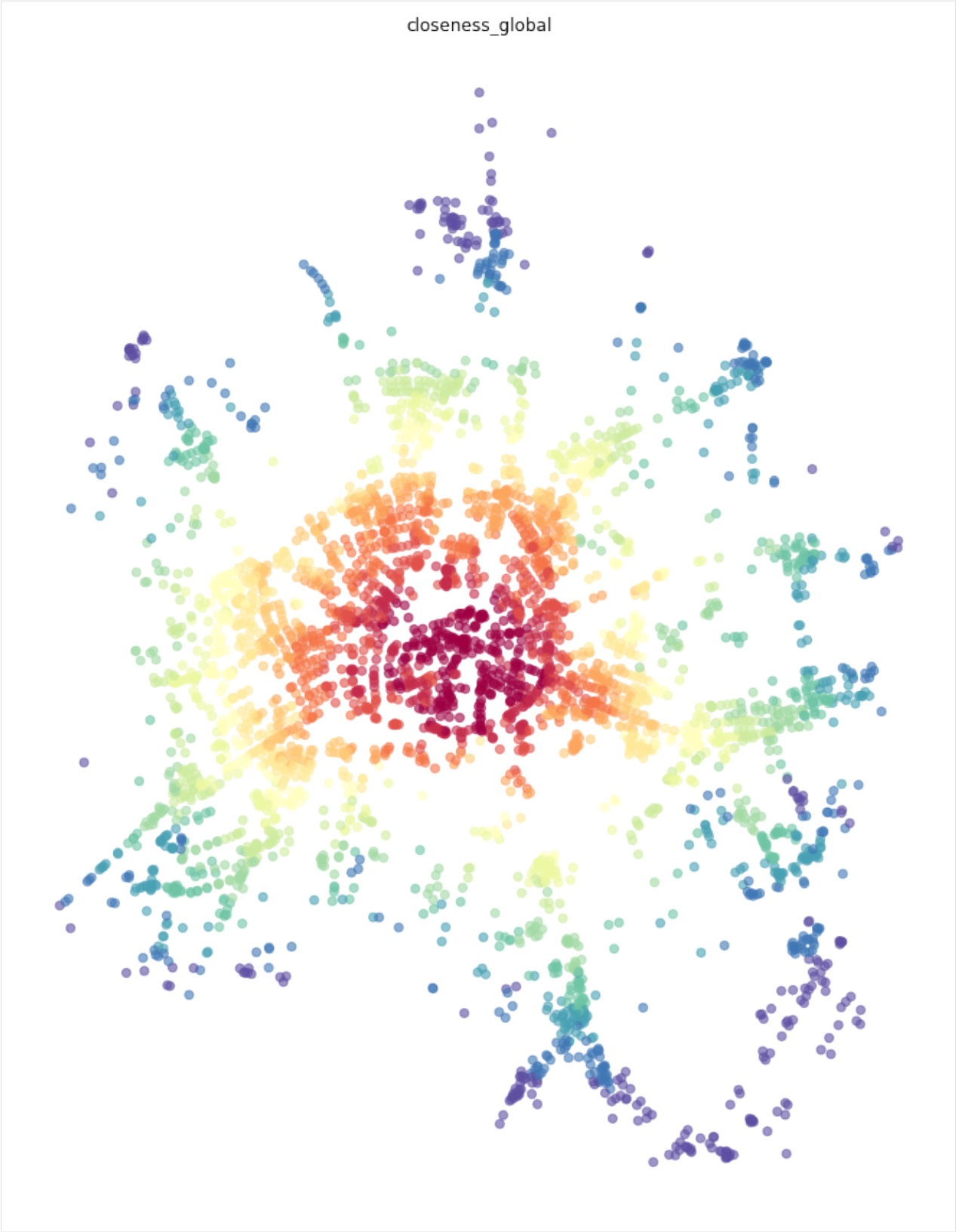

closeness400

Global closeness

Global closeness centrality is a bit simpler as we do not have to specify radius and distance, the rest remains the same.

[8]:
```
primal = momepy.closeness_centrality(primal, name='closeness_global', weight='mm_len')
```

[9]:
```
nodes = momepy.nx_to_gdf(primal, lines=False)
f, ax = plt.subplots(figsize=(15, 15))
nodes.plot(ax=ax, column='closeness_global', cmap='Spectral_r', scheme='quantiles',␣
↪k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness_global')
plt.show()
```


closeness_global

Betweenness

Betweenness centrality measures the importance of each node or edge for the travelling along the network. It measures how many times is each node/edge used if we walk using the shortest paths from each node to every other.

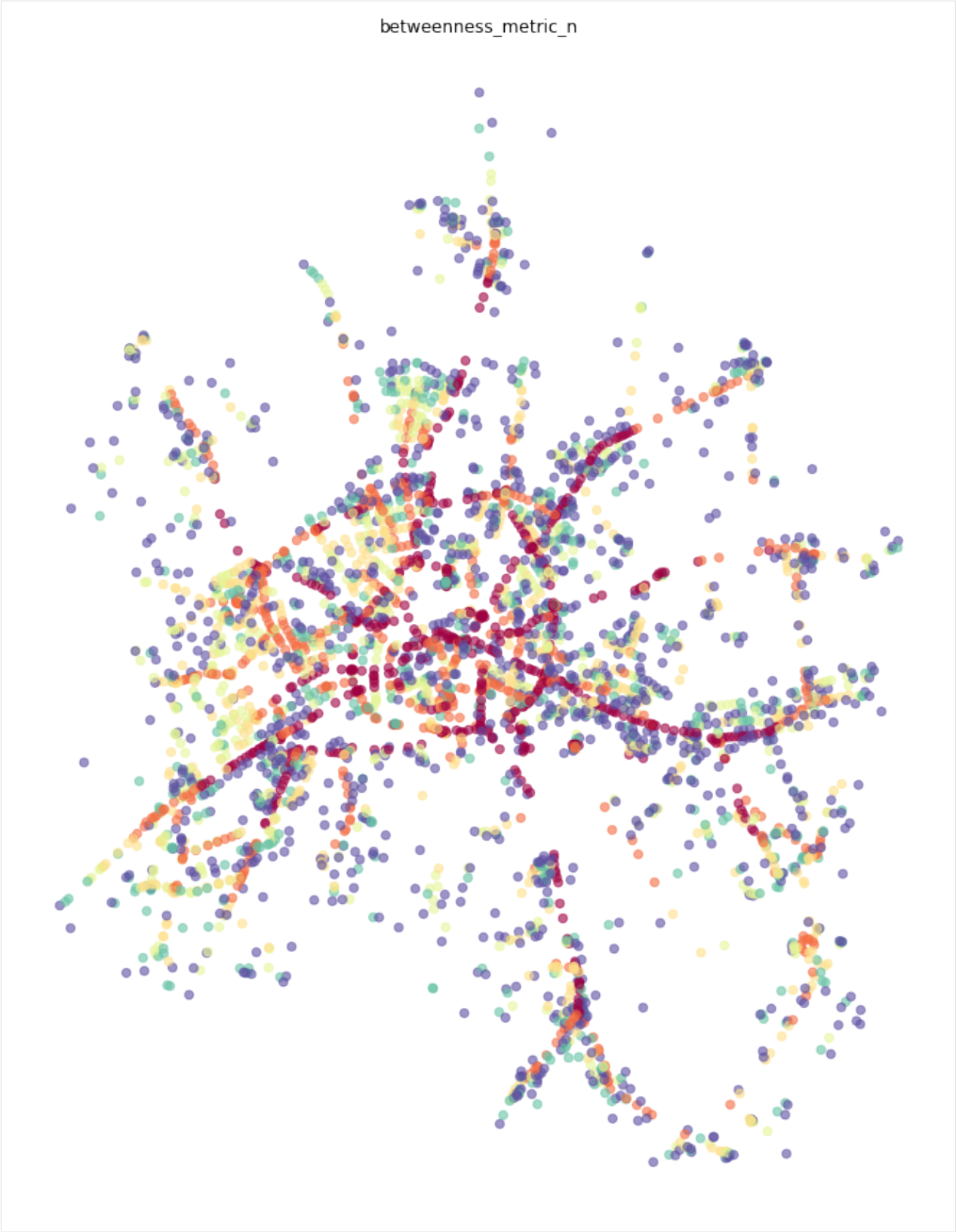
We have two options how to measure betweenness on primal graph - on nodes or on edges.

Node-based

Node-based betweenness, as name suggests, measures betweennes of each node - how many times we would walk through node.

[10]:
```
primal = momepy.betweenness_centrality(primal, name='betweenness_metric_n', mode=
↪'nodes', weight='mm_len')
```

[11]:
```
nodes = momepy.nx_to_gdf(primal, lines=False)
f, ax = plt.subplots(figsize=(15, 15))
nodes.plot(ax=ax, column='betweenness_metric_n', cmap='Spectral_r', scheme='quantiles
↪', k=7, alpha=0.6)
ax.set_axis_off()
ax.set_title('betweenness_metric_n')
plt.show()
```

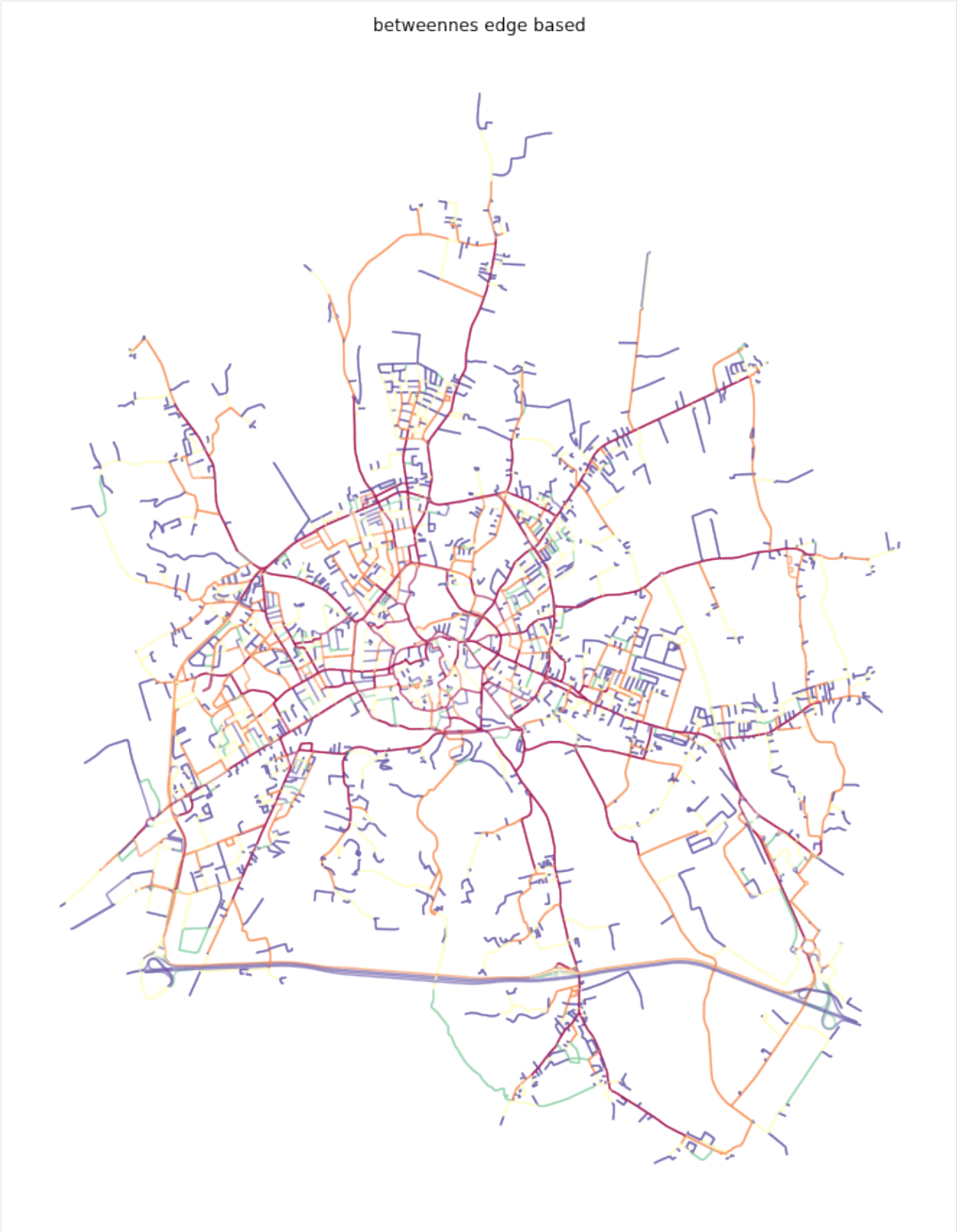

betweenness_metric_n

Edge-based

Edge-based betweenness does the same but for edges. How many times we go through each edge (street).

[12]:
```
primal = momepy.betweenness_centrality(primal, name='betweenness_metric_e', mode=
↪'edges', weight='mm_len')
```

[13]:
```
primal_gdf = momepy.nx_to_gdf(primal, points=False)
f, ax = plt.subplots(figsize=(15, 15))
primal_gdf.plot(ax=ax, column='betweenness_metric_e', cmap='Spectral_r', scheme=
↪'quantiles', alpha=0.6)
ax.set_axis_off()
ax.set_title('betweennes edge based')
plt.show()
```

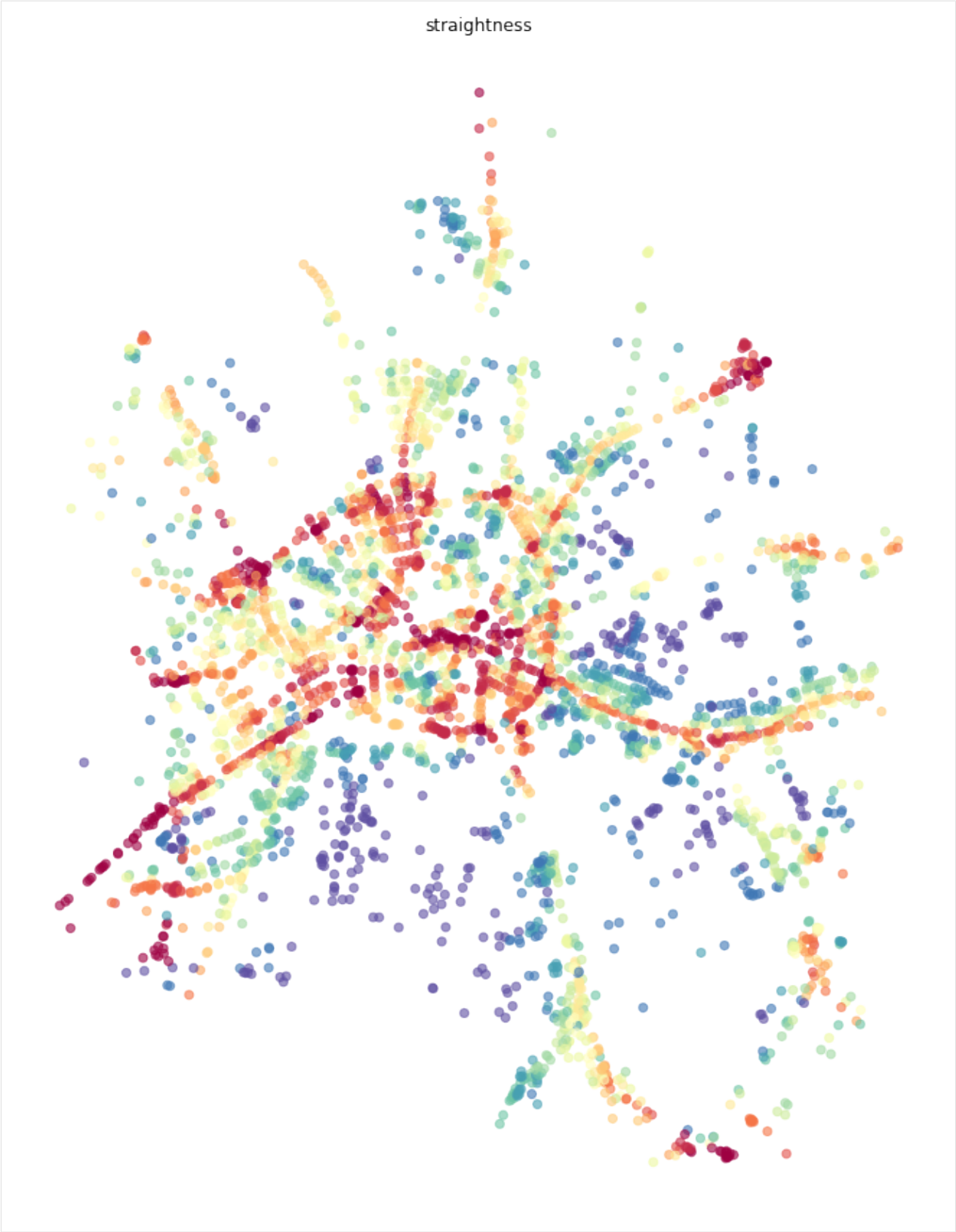

betweennes edge based

Straightness

While both closeness and betweenness are generally used in many applications of network analysis, straightness centrality is specific to street networks as it requires geographical element. It is measured as a ratio between real and Euclidean distance while waking from each node to every other.

[14]:
```
primal = momepy.straightness_centrality(primal)
```

[15]:
```
nodes = momepy.nx_to_gdf(primal, lines=False)
f, ax = plt.subplots(figsize=(15, 15))
nodes.plot(ax=ax, column='straightness', cmap='Spectral_r', scheme='quantiles', k=15,␣
↪alpha=0.6)
ax.set_axis_off()
ax.set_title('straightness')
plt.show()
```

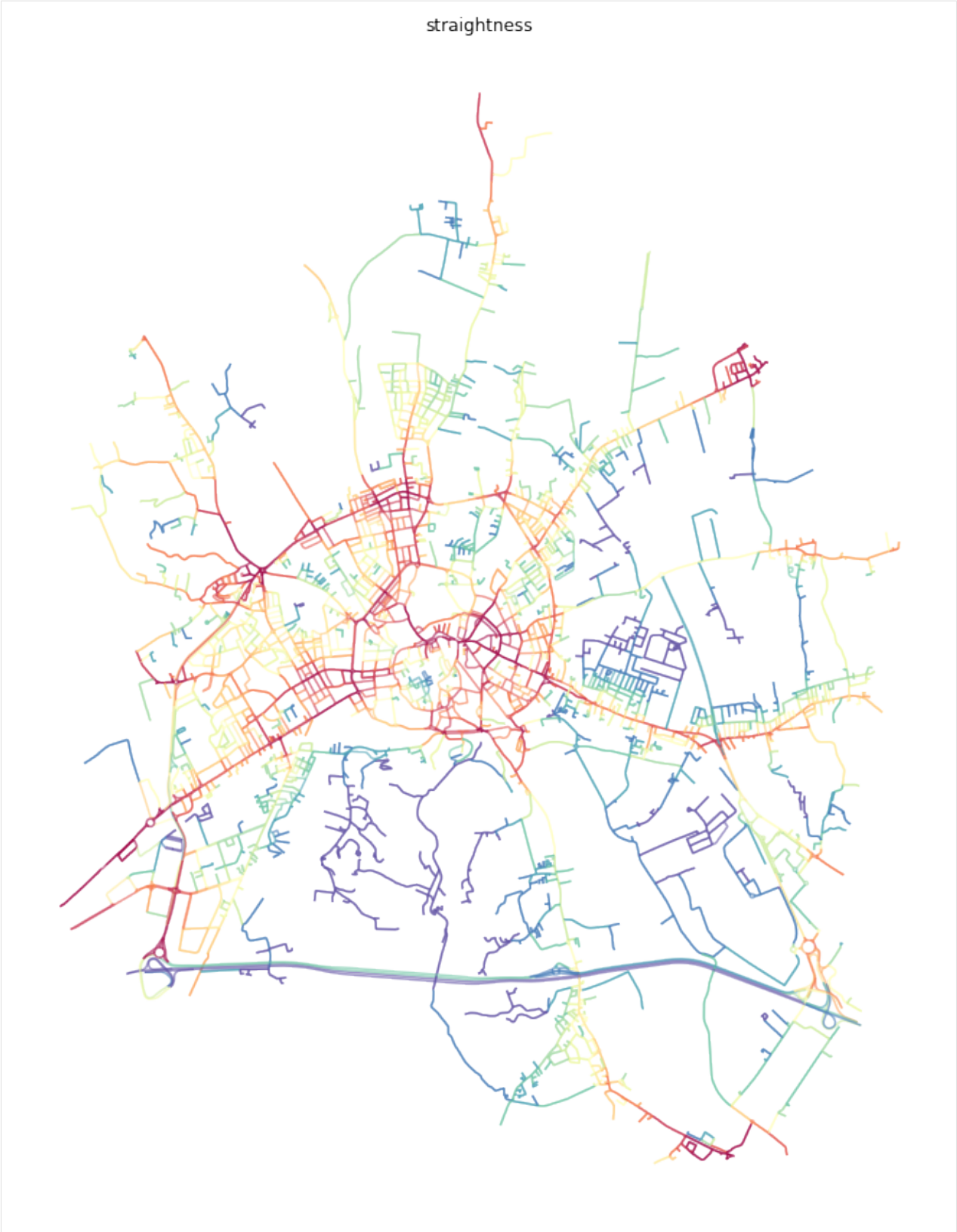

straightness

Node values averaged onto edges

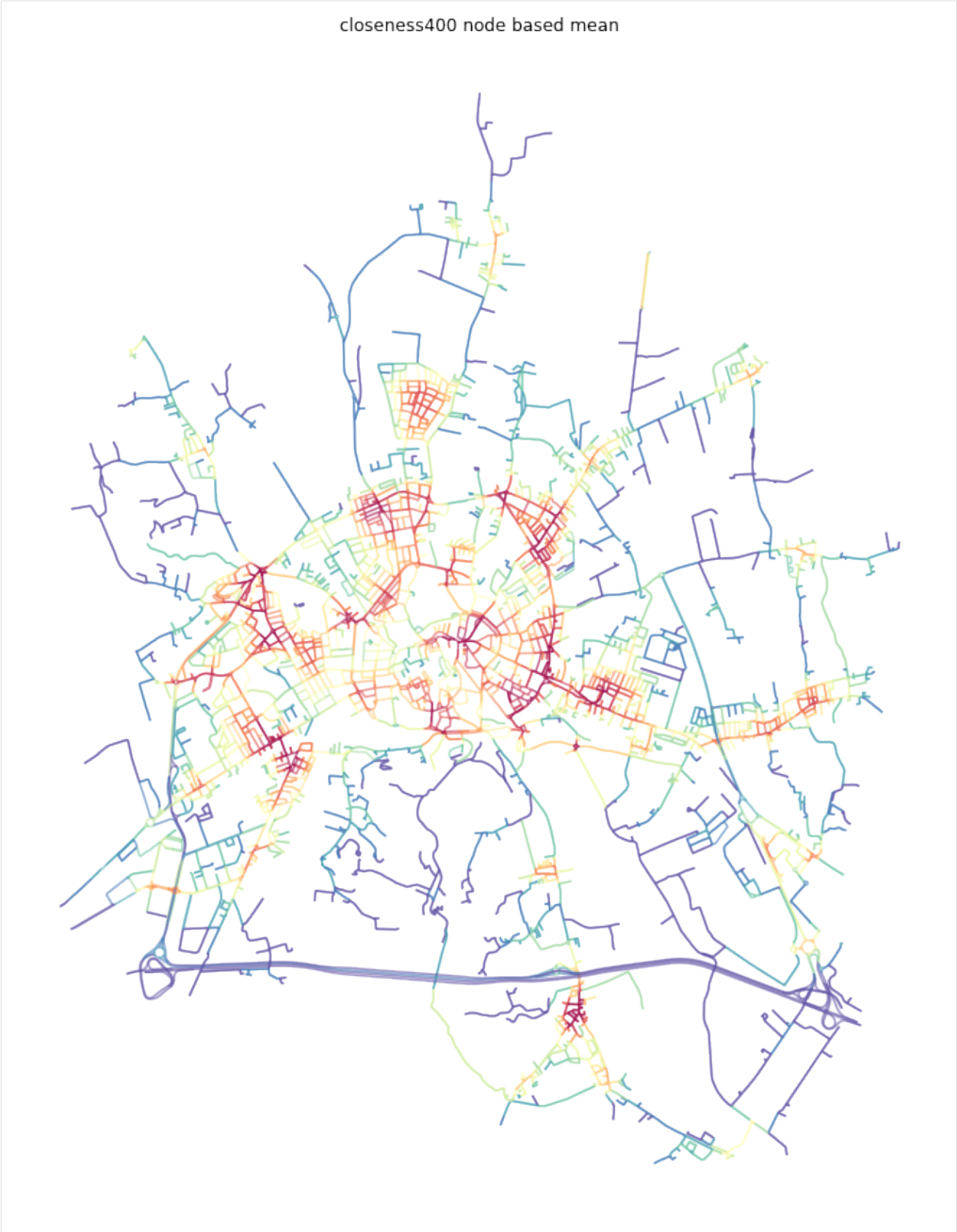
In some cases, it is easier to understand centrality results if they are attached to street segments, rather than intersections. We can do an approximation using the mean value attached to start and end node of each edge.

[16]:
```
momepy.mean_nodes(primal, 'straightness')
momepy.mean_nodes(primal, 'closeness400')
momepy.mean_nodes(primal, 'closeness_global')
momepy.mean_nodes(primal, 'betweenness_metric_n')
```

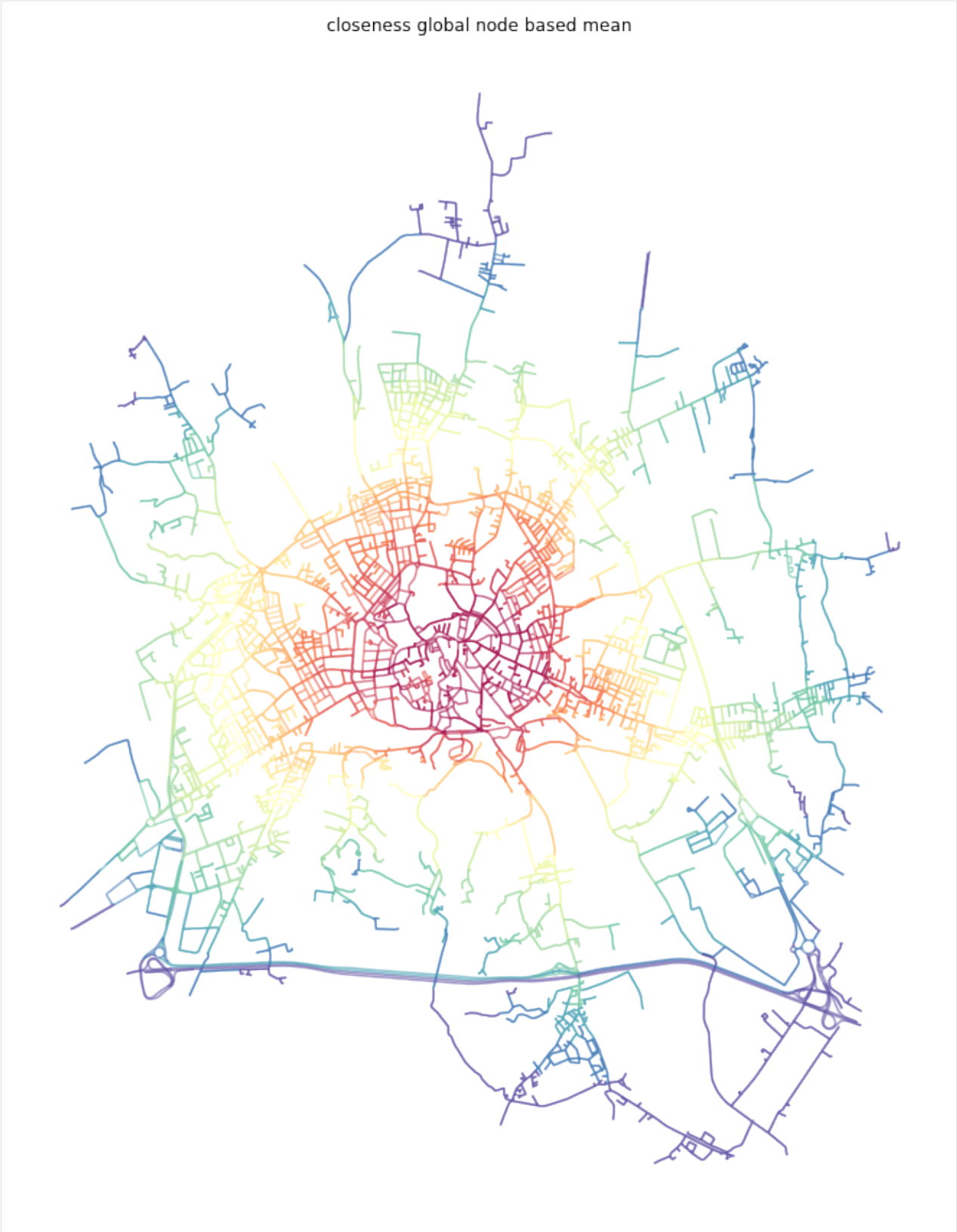
[17]:
```
primal_gdf = momepy.nx_to_gdf(primal, points=False)

f, ax = plt.subplots(figsize=(15, 15))
primal_gdf.plot(ax=ax, column='straightness', cmap='Spectral_r', scheme='quantiles',␣
↪k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('straightness')
plt.show()
```


straightness

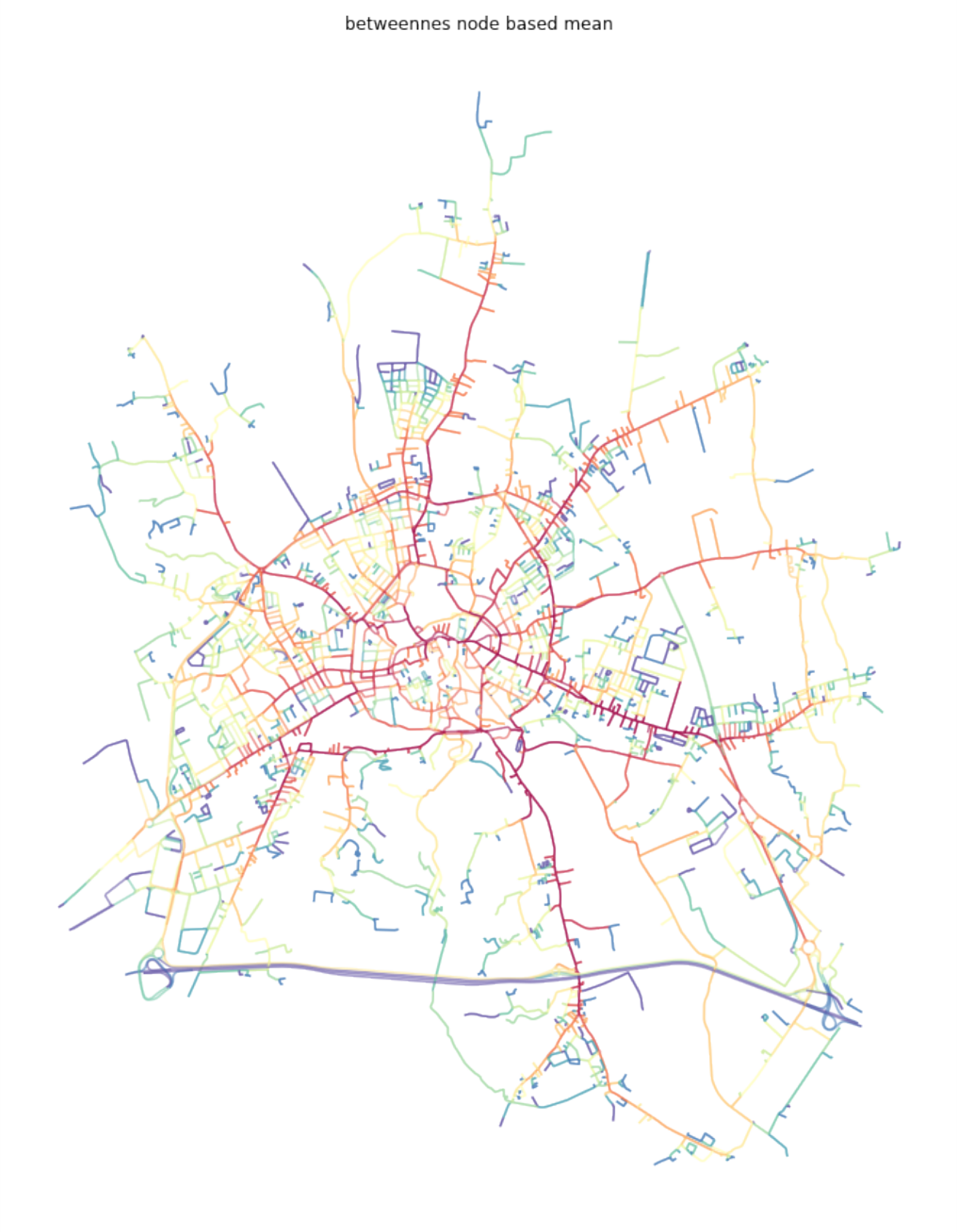
[18]:
```
f, ax = plt.subplots(figsize=(15, 15))
primal_gdf.plot(ax=ax, column='closeness400', cmap='Spectral_r', scheme='quantiles',␣
↪k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness400 node based mean')
plt.show()
```


closeness400 node based mean

[19]:
```
f, ax = plt.subplots(figsize=(15, 15))
primal_gdf.plot(ax=ax, column='closeness_global', cmap='Spectral_r', scheme='quantiles
↪', k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness global node based mean')
plt.show()
```


closeness global node based mean

[20]:
```
f, ax = plt.subplots(figsize=(15, 15))
primal_gdf.plot(ax=ax, column='betweenness_metric_n', cmap='Spectral_r', scheme=
↪'quantiles', k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('betweennes node based mean')
plt.show()
```



Once we have finished our network analysis on primal graph, we can save both nodes and edges back to GeoDataFrames.

[21]:
```
nodes, edges_p = momepy.nx_to_gdf(primal)
```

Topological vs metric distances

Centrality can be measure topologically (ignoring physical lenght of street segments) or metrically. Moreover, in the case of local centrality, local subgraph can be also defined topologically and metrically. In the end, you have four options how to measure centralilty, illustrated on `betweenness_centrality`:

```
# topologically defined subgraph (5 steps) and topologically measured shortest path
betweenness_centrality(graph, radius=5, distance=None, weight=None)

# topologically defined subgraph (5 steps) and metrically measured shortest path
betweenness_centrality(graph, radius=5, distance=None, weight='edge_length')

# metrically defined subgraph (800 meters) and topologically measured shortest path
betweenness_centrality(graph, radius=800, distance='edge_length', weight=None)

# metrically defined subgraph (800 meters) and metrically measured shortest path
betweenness_centrality(graph, radius=800, distance='edge_length', weight='edge_length
↪')
```

Dual graph

Dual graph is a bit more complicated concept as it represents street segments as nodes while intersections as edges connecting nodes. The geographical distance is lost as edges are of virtually no length, but we can capture the angle between each connected streets. Momepy does that by default, using angles between lines connecting start and end points of each segment. Hence, we can measure angular centrality.

Note: Dual graphs have naturally much more connections than primal ones, so computation of centrality on dual graph takes longer.

[22]:
```
dual = momepy.gdf_to_nx(edges, approach='dual')
```

Angular closeness

The situation with closeness is similar to the one done on primal graph. We can again do global and local closeness.

Local

Difference is that we do not have geographic distance, so we can limit closeness topologically or by another attribute (line angle). Example below uses topological distance of 5 steps.

[23]:
```
dual = momepy.closeness_centrality(dual, radius=5, name='angcloseness400', weight=
↪'angle')
```

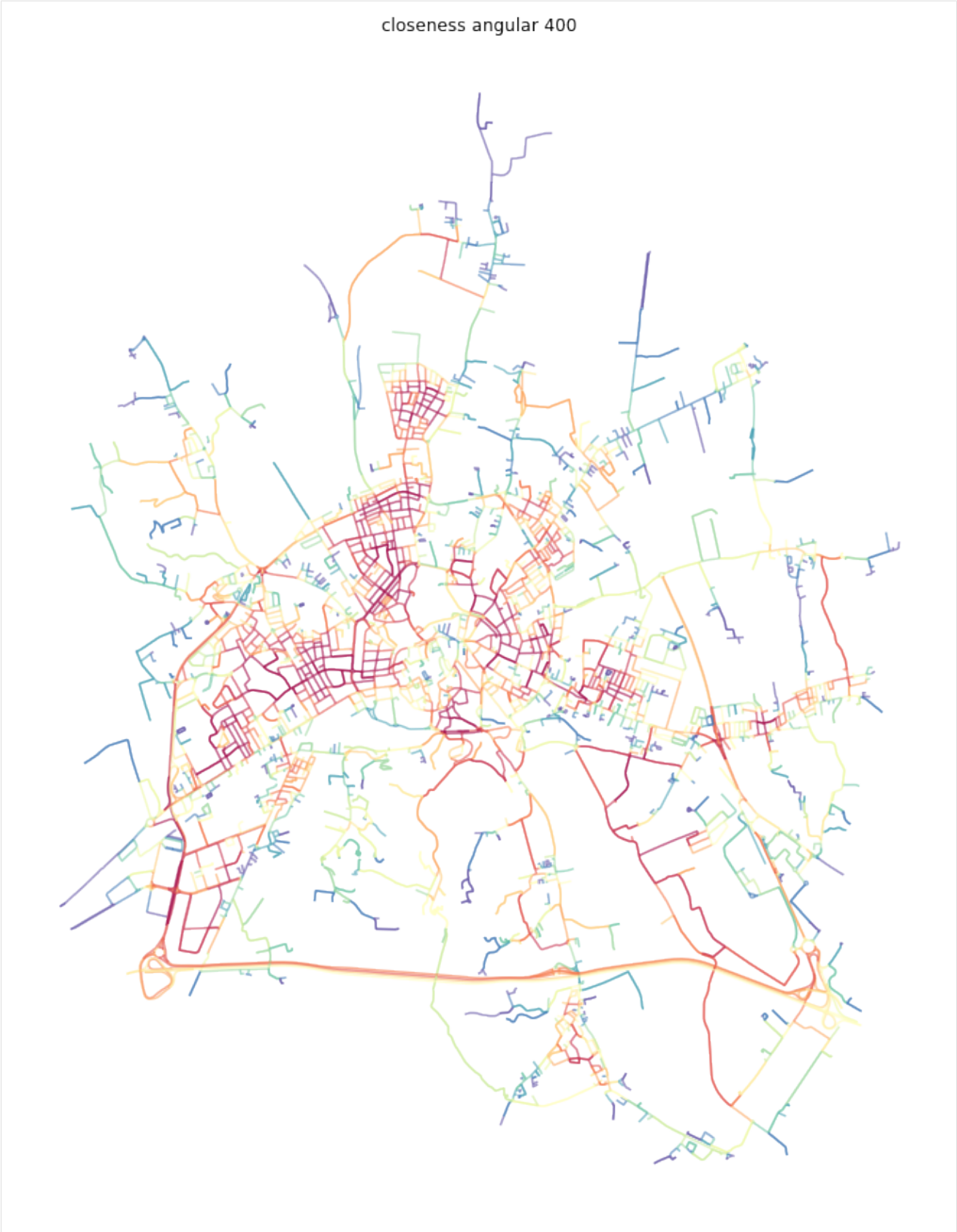
```
100%|| 6071/6071 [00:22<00:00, 275.92it/s]
```

[24]:
```
dual_gdf = momepy.nx_to_gdf(dual, points=False)

f, ax = plt.subplots(figsize=(15, 15))
```

(continues on next page)

(continued from previous page)

```
dual_gdf.plot(ax=ax, column='angcloseness400', cmap='Spectral_r', scheme='quantiles',␣
↪k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness angular 400')
plt.show()
```

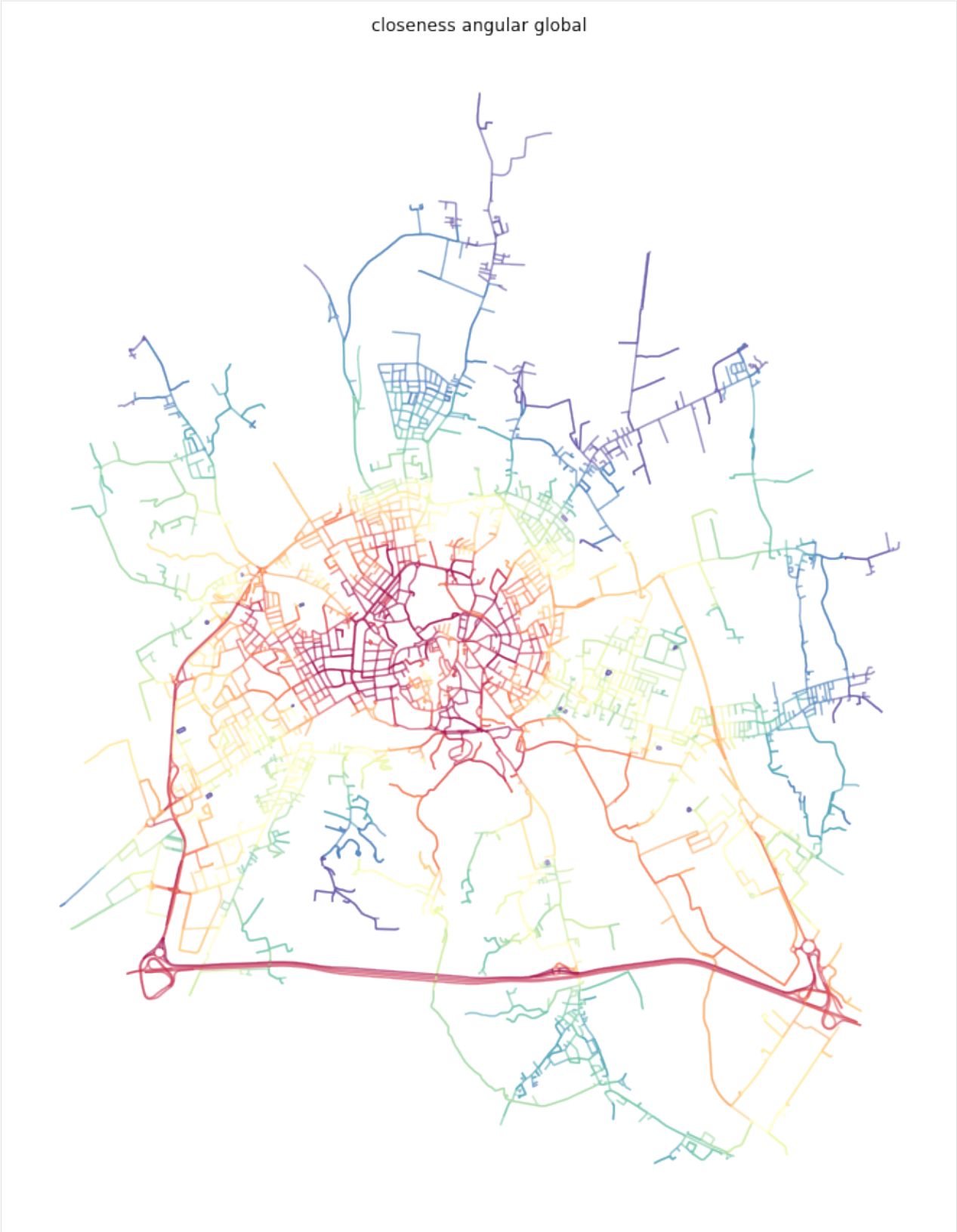

closeness angular 400

Global

Global angular closeness is then straightforward.

[25]:
```
dual = momepy.closeness_centrality(dual, name='closeness_global_ang', weight='angle')
```

[26]:
```
dual_gdf = momepy.nx_to_gdf(dual, points=False)
f, ax = plt.subplots(figsize=(15, 15))
dual_gdf.plot(ax=ax, column='closeness_global_ang', cmap='Spectral_r', scheme=
↪'quantiles', k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('closeness angular global')
plt.show()
```


closeness angular global

Angular betweenness

The last option momepy offers is angular betweenness. Just keep in mind, that nodes and edges are not representing the same concepts and it does not make much sense to measure angular betweennes on edges. Moreover, `nx_to_gdf` converts dual graph to LineString gdf only.

[27]:
```
dual = momepy.betweenness_centrality(dual, name='angbetweenness', mode='nodes',␣
↪weight='angle')
```

[28]:
```
dual_gdf = momepy.nx_to_gdf(dual, points=False)
f, ax = plt.subplots(figsize=(15, 15))
dual_gdf.plot(ax=ax, column='angbetweenness', cmap='Spectral_r', scheme='quantiles',␣
↪k=15, alpha=0.6)
ax.set_axis_off()
ax.set_title('betweenness angular')
plt.show()
```


betweenness angular

[29]:
```
edges_d = momepy.nx_to_gdf(dual)
```

8.3 momepy API reference

8.3.1 elements

Blocks(tessellation, edges, buildings, ...)	Generate blocks based on buildings, tesselation and street network.
buffered_limit(gdf[, buffer])	Define limit for *momepy.Tessellation* as a buffer around buildings.
enclosures(primary_barriers[, limit, ...])	Generate enclosures based on passed barriers.
get_network_id(left, right, network_id[, ...])	Snap each element (preferably building) to the closest street network segment, saves its id.
get_network_ratio(df, edges[, initial_buffer])	Link polygons to network edges based on the proportion of overlap (if a cell intersects more than one edge)
get_node_id(objects, nodes, edges, node_id)	Snap each building to closest street network node on the closest network edge.
Tessellation(gdf, unique_id[, limit, ...])	Generates tessellation.

momepy.Blocks

class momepy.**Blocks**(*tessellation*, *edges*, *buildings*, *id_name*, *unique_id*, *verbose=True*)

Generate blocks based on buildings, tesselation and street network.

Dissolves tessellation cells based on street-network based polygons. Links resulting id to `buildings` and `tesselation` as attributes.

Parameters

tessellation [GeoDataFrame] GeoDataFrame containing morphological tessellation

edges [GeoDataFrame] GeoDataFrame containing street network

buildings [GeoDataFrame] GeoDataFrame containing buildings

id_name [str] name of the unique blocks id column to be generated

unique_id [str] name of the column with unique id. If there is none, it could be generated by *momepy.unique_id()*. This should be the same for cells and buildings, id's should match.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> blocks_generate = mm.Blocks(tessellation_df, streets_df, buildings_df, 'bID',
↪'uID')
Buffering streets...
Generating spatial index...
Difference...
Defining adjacency...
Defining street-based blocks...
Defining block ID...
Generating centroids...
Spatial join...
Attribute join (tesselation)...
Generating blocks...
Multipart to singlepart...
Attribute join (buildings)...
Attribute join (tesselation)...
>>> blocks_generate.blocks.head()
    bID     geometry
0   1.0     POLYGON ((1603560.078648818 6464202.366899694,...
1   2.0     POLYGON ((1603457.225976106 6464299.454696888,...
2   3.0     POLYGON ((1603056.595487018 6464093.903488506,...
3   4.0     POLYGON ((1603260.943782872 6464141.327631323,...
4   5.0     POLYGON ((1603183.399594798 6463966.109982309,...
```

Attributes

blocks [GeoDataFrame] GeoDataFrame containing generated blocks

buildings_id [Series] Series derived from buildings with block ID

tessellation_id [Series] Series derived from morphological tessellation with block ID

tessellation [GeoDataFrame] GeoDataFrame containing original tessellation

edges [GeoDataFrame] GeoDataFrame containing original edges

buildings [GeoDataFrame] GeoDataFrame containing original buildings

id_name [str] name of the unique blocks id column

unique_id [str] name of the column with unique id

__init__(*tessellation*, *edges*, *buildings*, *id_name*, *unique_id*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(tessellation, edges, buildings, ...)	Initialize self.

momepy.buffered_limit

momepy.**buffered_limit**(*gdf*, *buffer=100*)

Define limit for *momepy.Tessellation* as a buffer around buildings.

See [FFRP20] for details.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing building footprints

buffer [float] buffer around buildings limiting the extend of tessellation

Returns

MultiPolygon MultiPolygon or Polygon defining the study area

Examples

```
>>> limit = mm.buffered_limit(buildings_df)
>>> type(limit)
shapely.geometry.polygon.Polygon
```

momepy.enclosures

momepy.**enclosures**(*primary_barriers*, *limit=None*, *additional_barriers=None*)

Generate enclosures based on passed barriers.

Enclosures are areas enclosed from all sides by at least one type of a barrier. Barriers are typically roads, railways, natural features like rivers and other water bodies or coastline. Enclosures are a result of polygonization of the `primary_barrier` and `limit` and its subdivision based on additional_barriers.

Parameters

primary_barriers [GeoDataFrame, GeoSeries] GeoDataFrame or GeoSeries containing primary barriers. (Multi)LineString geometry is expected.

limit [GeoDataFrame, GeoSeries (default None)] GeoDataFrame or GeoSeries containing external limit of enclosures, i.e. the area which gets partitioned. If None is passed, the internal area of `primary_barriers` will be used.

additional_barriers [GeoDataFrame] GeoDataFrame or GeoSeries containing additional barriers. (Multi)LineString geometry is expected.

Returns

enclosures [GeoSeries] GeoSeries containing enclosure geometries

Examples

```
>>> enclosures = mm.enclosures(streets, admin_boundary, [railway, rivers])
```

momepy.get_network_id

momepy.**get_network_id**(*left*, *right*, *network_id*, *min_size=100*, *verbose=True*)

Snap each element (preferably building) to the closest street network segment, saves its id.

Adds network ID to elements.

Parameters

left [GeoDataFrame] GeoDataFrame containing objects to snap

right [GeoDataFrame] GeoDataFrame containing street network with unique network ID. If there is none, it could be generated by *momepy.unique_id()*.

network_id [str, list, np.array, pd.Series (default None)] the name of the streets dataframe column, `np.array`, or `pd.Series` with network unique id.

min_size [int (default 100)] min_size should be a vaule such that if you build a box centered in each building centroid with edges of size `2*min_size`, you know a priori that at least one segment is intersected with the box.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

elements_nID [Series] Series containing network ID for elements

See also:

momepy.get_network_ratio

momepy.get_node_id

Examples

```
>>> buildings_df['nID'] = momepy.get_network_id(buildings_df, streets_df, 'nID')
Generating centroids...
Generating rtree...
Snapping: 100%|| 144/144 [00:00<00:00, 2718.98it/s]
>>> buildings_df['nID'][0]
1
```

momepy.get_network_ratio

momepy.**get_network_ratio**(*df*, *edges*, *initial_buffer=500*)

Link polygons to network edges based on the proportion of overlap (if a cell intersects more than one edge)

Useful if you need to link enclosed tessellation to street network. Ratios can be used as weights when linking network-based values to cells. For a purely distance-based link use *momepy.get_network_id()*.

Links are based on the integer position of edge (`iloc`).

Parameters

df [GeoDataFrame] GeoDataFrame containing objects to snap (typically enclosed tessellation)

edges [GeoDataFrame] GeoDataFrame containing street network

initial_buffer [float] Initial buffer used to link non-intersecting cells.

Returns

DataFrame

See also:

momepy.get_network_id

momepy.get_node_id

Examples

```
>>> links = mm.get_network_ratio(enclosed_tessellation, streets)
>>> links.head()
  edgeID_keys                              edgeID_values
0        [34]                                      [1.0]
1     [0, 34]  [0.38508998545027145, 0.6149100145497285]
2        [32]                                        [1]
3         [0]                                    [1.0]
4        [26]                                        [1]
```

momepy.get_node_id

momepy.**get_node_id**(*objects*, *nodes*, *edges*, *node_id*, *edge_id=None*, *edge_keys=None*, *edge_values=None*, *verbose=True*)

Snap each building to closest street network node on the closest network edge.

Adds node ID to objects (preferably buildings). Gets ID of edge (*momepy.get_network_id()* or *get_network_ratio()*) , and determines which of its end points is closer to building centroid.

Pass either `edge_id` with a single value or `edge_keys` and `edge_values` with ratios.

Parameters

objects [GeoDataFrame] GeoDataFrame containing objects to snap

nodes [GeoDataFrame] GeoDataFrame containing street nodes with unique node ID. If there is none, it could be generated by *momepy.unique_id()*.

edges [GeoDataFrame] GeoDataFrame containing street edges with unique edge ID and IDs of start and end points of each segment. Start and endpoints are default outcome of *momepy.nx_to_gdf()*.

node_id [str, list, np.array, pd.Series] the name of the nodes dataframe column, `np.array`, or `pd.Series` with unique id

edge_id [str (default None)] the name of the objects dataframe column with unique edge id (like an outcome of *momepy.get_network_id()*)

edge_keys [str (default None)] name the name of the objects dataframe column with edgeID_keys (like an outcome of *momepy.get_network_ratio()*)

edge_values [str (default None)] name the name of the objects dataframe column with edgeID_values (like an outcome of *momepy.get_network_ratio()*)

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

node_ids [Series] Series containing node ID for objects

momepy.Tessellation

class momepy.**Tessellation**(*gdf*, *unique_id*, *limit=None*, *shrink=0.4*, *segment=0.5*, *verbose=True*, *enclosures=None*, *enclosure_id='eID'*, *threshold=0.05*, *use_dask=True*, *n_chunks=8*, ***kwargs*)

Generates tessellation.

Three versions of tessellation can be created:

1. Morphological tessellation around given buildings `gdf` within set `limit`.
2. Proximity bands around given street network `gdf` within set `limit`.
3. Enclosed tessellation based on given buildings `gdf` within `enclosures`.

Pass either `limit` to create morphological tessellation or proximity bands or `enclosures` to create enclosed tessellation.

See [FFRP20] for details of implementation of morphological tessellation and [AF19] for proximity bands.

Tessellation requires data of relatively high level of precision and there are three particular patterns causing issues.

1. Features will collapse into empty polygon - these do not have tessellation cell in the end.

2. Features will split into MultiPolygon - at some cases, features with narrow links between parts split into two during 'shrinking'. In most cases that is not an issue and resulting tessellation is correct anyway, but sometimes this result in a cell being MultiPolygon, which is not correct.

3. Overlapping features - features which overlap even after 'shrinking' cause invalid tessellation geometry.

All three types can be tested prior *momepy.Tessellation* using *momepy.CheckTessellationInput*.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing building footprints or street network

unique_id [str] name of the column with unique id

limit [MultiPolygon or Polygon (default None)] MultiPolygon or Polygon defining the study area limiting morphological tessellation or proximity bands (otherwise it could go to infinity).

shrink [float (default 0.4)] distance for negative buffer to generate space between adjacent polygons (if geometry type of gdf is (Multi)Polygon).

segment [float (default 0.5)] maximum distance between points after discretization

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

enclosures [GeoDataFrame, GeoSeries (default None)] Enclosures geometry. Can be generated using *momepy.enclosures()*.

enclosure_id [str (default 'eID')] name of the enclosure_id (to be created). Applies only if `enclosures` are passed.

threshold [float (default 0.05)] The minimum threshold for a building to be considered within an enclosure. Threshold is a ratio of building area which needs to be within an enclosure to inlude it in the tessellation of that enclosure. Resolves sliver geometry issues. Applies only if `enclosures` are passed.

use_dask [bool (default True)] Use parallelised algorithm based on `dask.bag`. Requires dask. Applies only if `enclosures` are passed.

n_chunks [int (default 8)] Number of chunks to be used in parallelization. Ideal is one chunk per thread. Applies only if `enclosures` are passed.

Examples

```
>>> tess = mm.Tessellation(
... buildings_df, 'uID', limit=mm.buffered_limit(buildings_df)
... )
Inward offset...
Generating input point array...
Generating Voronoi diagram...
Generating GeoDataFrame...
Dissolving Voronoi polygons...
>>> tess.tessellation.head()
    uID     geometry
0   1       POLYGON ((1603586.677274485 6464344.667944215,...
1   2       POLYGON ((1603048.399497852 6464176.180701573,...
2   3       POLYGON ((1603071.342637536 6464158.863329805,...
3   4       POLYGON ((1603055.834005827 6464093.614718676,...
4   5       POLYGON ((1603106.417554705 6464130.215958447,...
```

```
>>> enclosures = mm.enclosures(streets, admin_boundary, [railway, rivers])
>>> encl_tess = mm.Tessellation(
... buildings_df, 'uID', enclosures=enclosures
... )
>>> encl_tess.tessellation.head()
     uID                                              geometry  eID
0  109.0  POLYGON ((1603369.789 6464340.661, 1603368.754...    0
1  110.0  POLYGON ((1603368.754 6464340.097, 1603369.789...    0
2  111.0  POLYGON ((1603458.666 6464332.614, 1603458.332...    0
3  112.0  POLYGON ((1603462.235 6464285.609, 1603454.795...    0
4  113.0  POLYGON ((1603524.561 6464388.609, 1603532.241...    0
```

Attributes

tessellation [GeoDataFrame] GeoDataFrame containing resulting tessellation

For enclosed tessellation, gdf contains three columns:

- `geometry`,
- `unique_id` matching with parental building,
- `enclosure_id` matching with enclosure integer index

gdf [GeoDataFrame] original GeoDataFrame

id [Series] Series containing used unique ID

limit [MultiPolygon or Polygon] limit

shrink [float] used shrink value

segment [float] used segment value

collapsed [list] list of unique_id's of collapsed features (if there are some) Applies only if `limit` is passed.

multipolygons [list] list of unique_id's of features causing MultiPolygons (if there are some) Applies only if `limit` is passed.

__init__(*gdf*, *unique_id*, *limit=None*, *shrink=0.4*, *segment=0.5*, *verbose=True*, *enclosures=None*, *enclosure_id='eID'*, *threshold=0.05*, *use_dask=True*, *n_chunks=8*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, unique_id[, limit, shrink, ...])	Initialize self.

8.3.2 dimension

Area(gdf)	Calculates area of each object in given GeoDataFrame.
AverageCharacter(gdf, values, ... [, rng, ...])	Calculates the average of a character within a set neighbourhood defined in `spatial_weights`
CourtyardArea(gdf[, areas])	Calculates area of holes within geometry - area of courtyards.
CoveredArea(gdf, spatial_weights, unique_id)	Calculates the area covered by neighbours
FloorArea(gdf, heights[, areas])	Calculates floor area of each object based on height and area.
LongestAxisLength(gdf)	Calculates the length of the longest axis of object.
Perimeter(gdf)	Calculates perimeter of each object in given GeoDataFrame.
PerimeterWall(gdf[, spatial_weights, verbose])	Calculate the perimeter wall length the joined structure.
SegmentsLength(gdf[, spatial_weights, mean, ...])	Calculate the cummulative and/or mean length of segments.
StreetProfile(left, right[, heights, ...])	Calculates the street profile characters.
Volume(gdf, heights[, areas])	Calculates volume of each object in given GeoDataFrame based on its height and area.
WeightedCharacter(gdf, values, ... [, areas, ...])	Calculates the weighted character

momepy.Area

class momepy.**Area**(*gdf*)
Calculates area of each object in given GeoDataFrame. It can be used for any suitable element (building footprint, plot, tessellation, block).

It is a simple wrapper for GeoPandas `.area` for the consistency of momepy.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

Examples

```
>>> buildings = gpd.read_file(momepy.datasets.get_path('bubenec'), layer=
↪'buildings')
>>> buildings['area'] = momepy.Area(buildings).series
>>> buildings.area[0]
728.5574947044363
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf)	Initialize self.

momepy.AverageCharacter

class momepy.**AverageCharacter**(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *mode='all'*, *verbose=True*)
Calculates the average of a character within a set neighbourhood defined in `spatial_weights`

Average value of the character within a set neighbourhood defined in `spatial_weights`. Can be set to `mean`, `median` or `mode`. `mean` is defined as:

$$\frac{1}{n}\left(\sum_{i=1}^{n} value_i\right)$$

Adapted from [HBP17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

unique_id [str] name of the column with unique id used as `spatial_weights` index.

spatial_weights [libpysal.weights] spatial weights matrix

rng [Two-element sequence containing floats in range of [0,100], optional] Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The order of the elements is not important.

mode [str (default 'all')] mode of average calculation. Can be set to *all*, *mean*, *median* or *mode* or list of any of the options.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = libpysal.weights.DistanceBand.from_dataframe(tessellation, threshold=100,
↪ silence_warnings=True, ids='uID')
>>> tessellation['mean_area'] = momepy.AverageCharacter(tessellation, values='area
↪', spatial_weights=sw, unique_id='uID').mean
100%|| 144/144 [00:00<00:00, 1433.32it/s]
>>> tessellation.mean_area[0]
4823.1334436678835
```

Attributes

series [Series] Series containing resulting mean values

mean [Series] Series containing resulting mean values

median [Series] Series containing resulting median values

mode [Series] Series containing resulting mode values

gdf [GeoDataFrame] original GeoDataFrame

values [GeoDataFrame] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

rng [tuple] range

modes [str] mode

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *mode='all'*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.CourtyardArea

class momepy.**CourtyardArea**(*gdf*, *areas=None*)
Calculates area of holes within geometry - area of courtyards.

Expects pygeos backend of geopandas.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to None, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings['courtyard_area'] = momepy.CourtyardArea(buildings).series
>>> buildings.courtyard_area[80]
353.33274206543274
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [GeoDataFrame] Series containing used areas values

__init__(*gdf*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas])	Initialize self.

momepy.CoveredArea

class momepy.**CoveredArea**(*gdf*, *spatial_weights*, *unique_id*, *verbose=True*)
Calculates the area covered by neighbours

Total area covered by neighbours defined in `spatial_weights` and element itself.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing Polygon geometry

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['covered3steps'] = mm.CoveredArea(tessellation_df, sw, 'uID').
↪series
100%|| 144/144 [00:00<00:00, 549.15it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__ (*gdf*, *spatial_weights*, *unique_id*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights, unique_id[, ...])	Initialize self.

momepy.FloorArea

class momepy.**FloorArea** (*gdf*, *heights*, *areas=None*)
Calculates floor area of each object based on height and area.

Number of floors is simplified into formula height / 3 (it is assumed that on average one floor is approximately 3 metres)

$$area * \frac{height}{3}$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

heights [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored height value

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to None, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings['floor_area'] = momepy.FloorArea(buildings, heights='height_col').
↪series
Calculating floor areas...
Floor areas calculated.
>>> buildings.floor_area[0]
2185.672484113309
```

```
>>> buildings['floor_area'] = momepy.FloorArea(buildings, heights='height_col',␣
↪areas='area_col').series
>>> buildings.floor_area[0]
2185.672484113309
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

heights [Series] Series containing used heights values

areas [GeoDataFrame] Series containing used areas values

__init__ (*gdf*, *heights*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, heights[, areas])	Initialize self.

momepy.LongestAxisLength

class momepy.**LongestAxisLength**(*gdf*)

Calculates the length of the longest axis of object.

Axis is defined as a diameter of minimal circumscribed circle around the convex hull. It does not have to be fully inside an object.

$$\max\{d_1, d_2, \ldots, d_n\}$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

Examples

```
>>> buildings['lal'] = momepy.LongestAxisLength(buildings).series
>>> buildings.lal[0]
40.2655616057102
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf)	Initialize self.

momepy.Perimeter

class momepy.**Perimeter**(*gdf*)

Calculates perimeter of each object in given GeoDataFrame. It can be used for any suitable element (building footprint, plot, tessellation, block).

It is a simple wrapper for GeoPandas `.length` for the consistency of momepy.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

Examples

```
>>> buildings = gpd.read_file(momepy.datasets.get_path('bubenec'), layer=
↪'buildings')
>>> buildings['perimeter'] = momepy.Perimeter(buildings).series
>>> buildings.perimeter[0]
137.18630991119903
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf)	Initialize self.

momepy.PerimeterWall

class momepy.**PerimeterWall**(*gdf*, *spatial_weights=None*, *verbose=True*)
Calculate the perimeter wall length the joined structure.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix will be calculated based on gdf. It is to denote adjacent buildings (note: based on index, not ID).

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Notes

It might take a while to compute this character.

Examples

```
>>> buildings_df['wall_length'] = mm.PerimeterWall(buildings_df).series
Calculating spatial weights...
Spatial weights ready...
100%|| 144/144 [00:00<00:00, 4171.39it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

__init__(*gdf*, *spatial_weights=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, spatial_weights, verbose])	Initialize self.

momepy.SegmentsLength

class momepy.**SegmentsLength**(*gdf*, *spatial_weights=None*, *mean=False*, *verbose=True*)
Calculate the cummulative and/or mean length of segments.

Length of segments within set topological distance from each of them. Reached topological distance should be captured by `spatial_weights`. If `mean=False` it will compute sum of length, if `mean=True` it will compute sum and mean.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing streets (edges) to analyse

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix will be calculated based on streets (note: spatial_weights should be based on index, not unique ID).

mean [boolean, optional] If mean=False it will compute sum of length, if mean=True it will compute sum and mean

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> streets_df['length_neighbours'] = mm.SegmentsLength(streets_df, mean=True).
↪mean
Calculating spatial weights...
Spatial weights ready...
```

Attributes

series [Series] Series containing resulting total lengths

mean [Series] Series containing resulting total lengths

sum [Series] Series containing resulting total lengths

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

__init__(*gdf*, *spatial_weights=None*, *mean=False*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, spatial_weights, mean, verbose])	Initialize self.

momepy.StreetProfile

class momepy.**StreetProfile**(*left*, *right*, *heights=None*, *distance=10*, *tick_length=50*, *verbose=True*)

Calculates the street profile characters.

Returns a dictionary with widths, standard deviation of width, openness, heights, standard deviation of height and ratio height/width. Algorithm generates perpendicular ines to `right` dataframe features every `distance` and measures values on intersectionwith features of `left`. If no feature is reached within `tick_length` its value is set as width (being a theoretical maximum).

Derived from [AF19].

Parameters

left [GeoDataFrame] GeoDataFrame containing streets to analyse

right [GeoDataFrame] GeoDataFrame containing buildings along the streets (only Polygon geometry type is supported)

heights: str, list, np.array, pd.Series (default None) the name of the buildings dataframe column, `np.array`, or `pd.Series` where is stored building height. If set to None, height and ratio height/width will not be calculated.

distance [int (default 10)] distance between perpendicular ticks

tick_length [int (default 50)] length of ticks

Examples

```
>>> street_profile = momepy.StreetProfile(streets_df, buildings_df, heights=
↪'height')
100%|| 33/33 [00:02<00:00, 15.66it/s]
>>> streets_df['width'] = street_profile.w
>>> streets_df['deviations'] = street_profile.wd
```

Attributes

w [Series] Series containing street profile width values

wd [Series] Series containing street profile standard deviation values

o [Series] Series containing street profile openness values

h [Series] Series containing street profile heights values. Returned only when heights is set.

hd [Series] Series containing street profile heights standard deviation values. Returned only when heights is set.

p [Series] Series containing street profile height/width ratio values. Returned only when heights is set.

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

distance [int] distance between perpendicular ticks

tick_length [int] length of ticks

heights [GeoDataFrame] Series containing used height values

__init__(*left*, *right*, *heights=None*, *distance=10*, *tick_length=50*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right[, heights, distance, ...])	Initialize self.

momepy.Volume

class momepy.**Volume**(*gdf*, *heights*, *areas=None*)
Calculates volume of each object in given GeoDataFrame based on its height and area.

$$area * height$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

heights [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored height value

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to None, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings['volume'] = momepy.Volume(buildings, heights='height_col').series
>>> buildings.volume[0]
7285.5749470443625
```

```
>>> buildings['volume'] = momepy.Volume(buildings, heights='height_col', areas=
↪'area_col').series
>>> buildings.volume[0]
7285.5749470443625
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

heights [Series] Series containing used heights values

areas [GeoDataFrame] Series containing used areas values

__init__(*gdf*, *heights*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, heights[, areas])	Initialize self.

momepy.WeightedCharacter

class momepy.**WeightedCharacter**(*gdf*, *values*, *spatial_weights*, *unique_id*, *areas=None*, *verbose=True*)

Calculates the weighted character

Character weighted by the area of the objects within `k` topological steps defined in `spatial_weights`.

$$\frac{\sum_{i=1}^{n} character_i * area_i}{\sum_{i=1}^{n} area_i}$$

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

values [str, list, np.array, pd.Series] the name of the gdf dataframe column, `np.array`, or `pd.Series` where is stored character to be weighted

spatial_weights [libpysal.weights] spatial weights matrix - If None, Queen contiguity matrix of set order will be calculated based on left.

unique_id [str] name of the column with unique id used as `spatial_weights` index.

areas [str, list, np.array, pd.Series (default None)] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored area value

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = libpysal.weights.DistanceBand.from_dataframe(tessellation_df,␣
↪threshold=100, silence_warnings=True)
>>> buildings_df['w_height_100'] = momepy.WeightedCharacter(buildings_df, values=
↪'height', spatial_weights=sw,
                                                            unique_id='uID').
↪series
100%|| 144/144 [00:00<00:00, 361.60it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [GeoDataFrame] Series containing used values

areas [GeoDataFrame] Series containing used areas

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *areas=None*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(gdf, values, spatial_weights, unique_id)	Initialize self.

8.3.3 shape

`CentroidCorners`(gdf[, verbose])	Calculates mean distance centroid - corners and st.
`CircularCompactness`(gdf[, areas])	Calculates compactness index of each object in given GeoDataFrame.
`CompactnessWeightedAxis`(gdf[, areas, ...])	Calculates compactness-weighted axis of each object in given GeoDataFrame.
`Convexity`(gdf[, areas])	Calculates Convexity index of each object in given GeoDataFrame.
`Corners`(gdf[, verbose])	Calculates number of corners of each object in given GeoDataFrame.
`CourtyardIndex`(gdf, courtyard_areas[, areas])	Calculates courtyard index of each object in given GeoDataFrame.
`Elongation`(gdf)	Calculates elongation of object seen as elongation of its minimum bounding rectangle.
`EquivalentRectangularIndex`(gdf[, areas, ...])	Calculates equivalent rectangular index of each object in given GeoDataFrame.
`FormFactor`(gdf, volumes[, areas])	Calculates form factor of each object in given GeoDataFrame.
`FractalDimension`(gdf[, areas, perimeters])	Calculates fractal dimension of each object in given GeoDataFrame.
`Linearity`(gdf[, verbose])	Calculates linearity of each LineString object in given GeoDataFrame.
`Rectangularity`(gdf[, areas])	Calculates rectangularity of each object in given GeoDataFrame.
`ShapeIndex`(gdf, longest_axis[, areas])	Calculates shape index of each object in given GeoDataFrame.
`SquareCompactness`(gdf[, areas, perimeters])	Calculates compactness index of each object in given GeoDataFrame.
`Squareness`(gdf[, verbose])	Calculates squareness of each object in given GeoDataFrame.
`VolumeFacadeRatio`(gdf, heights[, volumes, ...])	Calculates volume/facade ratio of each object in given GeoDataFrame.

momepy.CentroidCorners

class momepy.**CentroidCorners**(*gdf*, *verbose=True*)

Calculates mean distance centroid - corners and st. deviation.

$$\overline{x} = \frac{1}{n}\left(\sum_{i=1}^{n} dist_i\right); \text{SD} = \sqrt{\frac{\sum |x - \overline{x}|^2}{n}}$$

Adapted from [SA15] and [Cim17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> ccd = momepy.CentroidCorners(buildings_df)
100%|| 144/144 [00:00<00:00, 846.58it/s]
>>> buildings_df['ccd_means'] = ccd.means
>>> buildings_df['ccd_stdev'] = ccd.std
>>> buildings_df['ccd_means'][0]
15.961531913184833
>>> buildings_df['ccd_stdev'][0]
3.0810634305400177
```

Attributes

mean [Series] Series containing mean distance values.

std [Series] Series containing standard deviation values.

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, verbose])	Initialize self.

momepy.CircularCompactness

class momepy.**CircularCompactness**(*gdf*, *areas=None*)
Calculates compactness index of each object in given GeoDataFrame.

$$\frac{area}{\textit{area of enclosing circle}}$$

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['comp'] = momepy.CircularCompactness(buildings_df, 'area').series
>>> buildings_df['comp'][0]
0.572145421828038
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

__init__(*gdf*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas])	Initialize self.

momepy.CompactnessWeightedAxis

class momepy.**CompactnessWeightedAxis**(*gdf*, *areas=None*, *perimeters=None*, *longest_axis=None*)
Calculates compactness-weighted axis of each object in given GeoDataFrame.

Initially designed for blocks.

$$d_i \times \left(\frac{4}{\pi} - \frac{16(area_i)}{perimeter_i^2}\right)$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

perimeters [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value. If set to `None`, function will calculate perimeters during the process without saving them separately.

longest_axis [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored longest axis length value. If set to `None`, function will calculate it during the process without saving them separately.

Examples

```
>>> blocks_df['cwa'] = mm.CompactnessWeightedAxis(blocks_df).series
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

longest_axis [Series] Series containing used area values

perimeters [Series] Series containing used area values

__init__(*gdf*, *areas=None*, *perimeters=None*, *longest_axis=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas, perimeters, longest_axis])	Initialize self.

momepy.Convexity

class momepy.**Convexity**(*gdf*, *areas=None*)
Calculates Convexity index of each object in given GeoDataFrame.

$$\frac{area}{convex\ hull\ area}$$

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['convexity'] = momepy.Convexity(buildings_df).series
>>> buildings_df['convexity'][0]
0.8151964258521672
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

__init__(*gdf*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas])	Initialize self.

momepy.Corners

class momepy.**Corners**(*gdf*, *verbose=True*)

Calculates number of corners of each object in given GeoDataFrame.

Uses only external shape (`shapely.geometry.exterior`), courtyards are not included.

$$\sum corner$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['corners'] = momepy.Corners(buildings_df).series
100%|| 144/144 [00:00<00:00, 1042.15it/s]
>>> buildings_df.corners[0]
24
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, verbose])	Initialize self.

momepy.CourtyardIndex

class momepy.**CourtyardIndex**(*gdf*, *courtyard_areas*, *areas=None*)

Calculates courtyard index of each object in given GeoDataFrame.

$$\frac{area\ of\ courtyards}{total\ area}$$

Adapted from [SA15].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

courtyard_areas [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value (To calculate volume you can use *momepy.CourtyardArea*)

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['courtyard_index'] = momepy.CourtyardIndex(buildings, 'courtyard_
↪area', 'area').series
>>> buildings_df.courtyard_index[80]
0.16605915738643523
```

```
>>> buildings_df['courtyard_index2'] = momepy.CourtyardIndex(buildings_df, momepy.
↪CourtyardArea(buildings_df).series).series
>>> buildings_df.courtyard_index2[80]
0.16605915738643523
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

courtyard_areas [Series] Series containing used courtyard areas values

areas [Series] Series containing used area values

__init__(*gdf*, *courtyard_areas*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, courtyard_areas[, areas])	Initialize self.

momepy.Elongation

class momepy.**Elongation**(*gdf*)
Calculates elongation of object seen as elongation of its minimum bounding rectangle.

$$\frac{\frac{p-\sqrt{p^2-16a}}{4}}{\frac{p}{2}-\frac{p-\sqrt{p^2-16a}}{4}}$$

where a is the area of the object and p its perimeter.

Based on [GMBeiraoD12].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

Examples

```
>>> buildings_df['elongation'] = momepy.Elongation(buildings_df).series
>>> buildings_df['elongation'][0]
0.9082437463675544
```

Attributes

e [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf)	Initialize self.

momepy.EquivalentRectangularIndex

class momepy.**EquivalentRectangularIndex**(*gdf*, *areas=None*, *perimeters=None*)
Calculates equivalent rectangular index of each object in given GeoDataFrame.

$$\sqrt{\frac{area}{area\ of\ bounding\ rectangle}} * \frac{perimeter\ of\ bounding\ rectangle}{perimeter}$$

Based on [BC17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

perimeters [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value. If set to `None`, function will calculate perimeters during the process without saving them separately.

Examples

```
>>> buildings_df['eri'] = momepy.EquivalentRectangularIndex(buildings_df, 'area',
↪'peri').series
>>> buildings_df['eri'][0]
0.7879229963118455
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

perimeters [Series] Series containing used perimeter values

__init__(*gdf*, *areas=None*, *perimeters=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas, perimeters])	Initialize self.

momepy.FormFactor

class momepy.**FormFactor**(*gdf*, *volumes*, *areas=None*)
Calculates form factor of each object in given GeoDataFrame.

$$\frac{area}{volume^{\frac{2}{3}}}$$

Adapted from [BSN12].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

volumes [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored volume value. (To calculate volume you can use `momepy.volume()`)

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['formfactor'] = momepy.FormFactor(buildings_df, 'volume').series
>>> buildings_df.formfactor[0]
1.9385988170288635
```

```
>>> buildings_df['formfactor'] = momepy.FormFactor(buildings_df, momepy.
↪volume(buildings_df, 'height').volume).series
>>> buildings_df.formfactor[0]
1.9385988170288635
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

volumes [Series] Series containing used volume values

areas [Series] Series containing used area values

__init__(*gdf*, *volumes*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(gdf, volumes[, areas])	Initialize self.

momepy.FractalDimension

class momepy.**FractalDimension**(*gdf*, *areas=None*, *perimeters=None*)

Calculates fractal dimension of each object in given GeoDataFrame.

$$\frac{2log(\frac{perimeter}{4})}{log(area)}$$

Based on [McG95].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

perimeters [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value. If set to `None`, function will calculate perimeters during the process without saving them separately.

Examples

```
>>> buildings_df['fractal'] = momepy.FractalDimension(buildings_df, 'area', 'peri
↪').series
>>> buildings_df.fractal[0]
1.0726778567038908
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

perimeters [Series] Series containing used perimeter values

areas [Series] Series containing used area values

__init__(*gdf*, *areas=None*, *perimeters=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas, perimeters])	Initialize self.

momepy.Linearity

class momepy.**Linearity**(*gdf*, *verbose=True*)

Calculates linearity of each LineString object in given GeoDataFrame.

$$\frac{l_{euclidean}}{l_{segment}}$$

where *l* is the length of the LineString.

Adapted from [AF19].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> streets_df['linearity'] = momepy.Linearity(streets_df).series
>>> streets_df['linearity'][0]
1.0
```

Attributes

series [Series] Series containing mean distance values.

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, verbose])	Initialize self.

momepy.Rectangularity

class momepy.**Rectangularity**(*gdf*, *areas=None*)

Calculates rectangularity of each object in given GeoDataFrame.

$$\frac{area}{\textit{minimum bounding rotated rectangle area}}$$

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['rectangularity'] = momepy.Rectangularity(buildings_df, 'area').
↪series
100%|| 144/144 [00:00<00:00, 866.62it/s]
>>> buildings_df.rectangularity[0]
0.6942676157646379
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

__init__(*gdf*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas])	Initialize self.

momepy.ShapeIndex

class momepy.**ShapeIndex**(*gdf*, *longest_axis*, *areas=None*)
Calculates shape index of each object in given GeoDataFrame.

$$\frac{\sqrt{\frac{area}{\pi}}}{0.5 * longest\ axis}$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

longest_axis [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored longest axis value

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

Examples

```
>>> buildings_df['shape_index'] = momepy.ShapeIndex(buildings_df, longest_axis=
↪'long_ax', areas='area').series
100%|| 144/144 [00:00<00:00, 5558.33it/s]
>>> buildings_df['shape_index'][0]
0.7564029493781987
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

longest_axis [Series] Series containing used longest axis values

areas [Series] Series containing used area values

__init__(*gdf*, *longest_axis*, *areas=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, longest_axis[, areas])	Initialize self.

momepy.SquareCompactness

class momepy.**SquareCompactness**(*gdf*, *areas=None*, *perimeters=None*)
Calculates compactness index of each object in given GeoDataFrame.

$$\left(\frac{4\sqrt{area}}{perimeter}\right)^2$$

Adapted from [Fel18].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

areas [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If set to `None`, function will calculate areas during the process without saving them separately.

perimeters [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value. If set to `None`, function will calculate perimeters during the process without saving them separately.

Examples

```
>>> buildings_df['squ_comp'] = momepy.SquareCompactness(buildings_df).series
>>> buildings_df['squ_comp'][0]
0.6193872538650996
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

areas [Series] Series containing used area values

perimeters [Series] Series containing used perimeter values

__init__(*gdf*, *areas=None*, *perimeters=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, areas, perimeters])	Initialize self.

momepy.Squareness

class momepy.**Squareness**(*gdf*, *verbose=True*)
Calculates squareness of each object in given GeoDataFrame.

Uses only external shape (`shapely.geometry.exterior`), courtyards are not included.

$$\mu = \frac{\sum_{i=1}^{N} d_i}{N}$$

where d is the deviation of angle of corner i from 90 degrees.

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['squareness'] = momepy.Squareness(buildings_df).series
100%|| 144/144 [00:01<00:00, 129.49it/s]
>>> buildings_df.squareness[0]
3.7075816043359864
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, verbose])	Initialize self.

momepy.VolumeFacadeRatio

class momepy.**VolumeFacadeRatio**(*gdf*, *heights*, *volumes=None*, *perimeters=None*)
Calculates volume/facade ratio of each object in given GeoDataFrame.

$$\frac{volume}{perimeter * height}$$

Adapted from [SA15].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects

heights [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored height value

volumes [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored volume value

perimeters [, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value

Examples

```
>>> buildings_df['vfr'] = momepy.VolumeFacadeRatio(buildings_df, 'height').series
>>> buildings_df.vfr[0]
5.310715735236504
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

perimeters [Series] Series containing used perimeter values

volumes [Series] Series containing used volume values

__init__(*gdf*, *heights*, *volumes=None*, *perimeters=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, heights[, volumes, perimeters])	Initialize self.

8.3.4 spatial distribution

Alignment(gdf, spatial_weights, unique_id, ...)	Calculate the mean deviation of solar orientation of objects on adjacent cells from an object
BuildingAdjacency(gdf, ...[, ...])	Calculate the level of building adjacency
CellAlignment(left, right, ...)	Calculate the difference between cell orientation and orientation of object
MeanInterbuildingDistance(gdf, ...[, ...])	Calculate the mean interbuilding distance
NeighborDistance(gdf, spatial_weights, unique_id)	Calculate the mean distance to adjacent buildings (based on `spatial_weights`)
NeighboringStreetOrientationDeviation(gdf)	Calculate the mean deviation of solar orientation of adjacent streets
Neighbors(gdf, spatial_weights, unique_id[, ...])	Calculate the number of neighbours captured by `spatial_weights`
Orientation(gdf[, verbose])	Calculate the orientation of object
SharedWallsRatio(gdf[, unique_id, perimeters])	Calculate shared walls ratio of adjacent elements (typically buildings)
StreetAlignment(left, right, orientations[, ...])	Calculate the difference between street orientation and orientation of object in degrees

momepy.Alignment

class momepy.**Alignment**(*gdf*, *spatial_weights*, *unique_id*, *orientations*, *verbose=True*)

Calculate the mean deviation of solar orientation of objects on adjacent cells from an object

$$\frac{1}{n}\sum_{i=1}^{n} dev_i = \frac{dev_1 + dev_2 + \cdots + dev_n}{n}$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

spatial_weights [libpysal.weights, optional] spatial weights matrix

orientations [str, list, np.array, pd.Series] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored object orientation value (can be calculated using *momepy.Orientation*)

unique_id [str] name of the column with unique id used as `spatial_weights` index.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['alignment'] = momepy.Alignment(buildings_df, sw, 'uID', bl_
↪orient).series
100%|| 144/144 [00:01<00:00, 140.84it/s]
>>> buildings_df['alignment'][0]
18.299481296455237
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

orientations [Series] Series containing used orientation values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__(*gdf*, *spatial_weights*, *unique_id*, *orientations*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights, unique_id, ...)	Initialize self.

momepy.BuildingAdjacency

class momepy.**BuildingAdjacency**(*gdf*, *spatial_weights_higher*, *unique_id*, *spatial_weights=None*, *verbose=True*)

Calculate the level of building adjacency

Building adjacency reflects how much buildings tend to join together into larger structures. It is calculated as a ratio of joined built-up structures and buildings within the extent defined in `spatial_weights_higher`.

Adapted from [VC17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

spatial_weights_higher [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix will be calculated based on gdf. It is to denote adjacent buildings (note: based on unique ID).

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['adjacency'] = momepy.BuildingAdjacency(buildings_df, swh,␣
↪unique_id='uID').series
Calculating spatial weights...
Spatial weights ready...
Calculating adjacency: 100%|| 144/144 [00:00<00:00, 335.55it/s]
>>> buildings_df['adjacency'][10]
0.23809523809523808
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw_higher [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

sw [libpysal.weights] spatial weights matrix

__init__(*gdf*, *spatial_weights_higher*, *unique_id*, *spatial_weights=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights_higher, unique_id)	Initialize self.

momepy.CellAlignment

class momepy.**CellAlignment**(*left*, *right*, *left_orientations*, *right_orientations*, *left_unique_id*, *right_unique_id*)
Calculate the difference between cell orientation and orientation of object

$$|building\ orientation - cell\ orientation|$$

Parameters

left [GeoDataFrame] GeoDataFrame containing objects to analyse

right [GeoDataFrame] GeoDataFrame containing tessellation cells (or relevant spatial units)

left_orientations [str, list, np.array, pd.Series] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored object orientation value (can be calculated using *momepy.Orientation*)

right_orientations [str, list, np.array, pd.Series] the name of the right dataframe column, `np.array`, or `pd.Series` where is stored object orientation value (can be calculated using *momepy.Orientation*)

left_unique_id [str] the name of the `left` dataframe column with unique id shared between `left` and `right` gdf

right_unique_id [str] the name of the `right` dataframe column with unique id shared between `left` and `right` gdf

Examples

```
>>> buildings_df['cell_alignment'] = momepy.CellAlignment(buildings_df,␣
↪tessellation_df, 'bl_orient', 'tes_orient', 'uID', 'uID').series
>>> buildings_df['cell_alignment'][0]
0.8795123936951939
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

left_orientations [Series] Series containing used left orientations

right_orientations [Series] Series containing used right orientations

left_unique_id [Series] Series containing used left ID

right_unique_id [Series] Series containing used right ID

__init__(*left*, *right*, *left_orientations*, *right_orientations*, *left_unique_id*, *right_unique_id*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, left_orientations, ...)	Initialize self.

momepy.MeanInterbuildingDistance

class momepy.**MeanInterbuildingDistance**(*gdf*, *spatial_weights*, *unique_id*, *spatial_weights_higher=None*, *order=3*, *verbose=True*)

Calculate the mean interbuilding distance

Interbuilding distances are calculated between buildings on adjacent cells based on `spatial_weights`, while the extent is defined as order of contiguity.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

unique_id [str] name of the column with unique id used as `spatial_weights` index

spatial_weights [libpysal.weights] spatial weights matrix

order [int] Order of contiguity defining the extent

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Notes

Fix UserWarning.

Examples

```
>>> buildings_df['mean_interbuilding_distance'] = momepy.
↪MeanInterbuildingDistance(buildings_df, sw, 'uID').series
Computing mean interbuilding distances...
100%|| 144/144 [00:00<00:00, 317.42it/s]
>>> buildings_df['mean_interbuilding_distance'][0]
29.305457092042744
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

sw_higher [libpysal.weights] Spatial weights matrix of higher order

order [int] Order of contiguity.

__init__(*gdf*, *spatial_weights*, *unique_id*, *spatial_weights_higher=None*, *order=3*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights, unique_id[, ...])	Initialize self.

momepy.NeighborDistance

class momepy.**NeighborDistance**(*gdf*, *spatial_weights*, *unique_id*, *verbose=True*)
Calculate the mean distance to adjacent buildings (based on `spatial_weights`)

If no neighbours are found, return `np.nan`.

$$\frac{1}{n}\sum_{i=1}^{n} dist_i = \frac{dist_1 + dist_2 + \cdots + dist_n}{n}$$

Adapted from [SA15].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

spatial_weights [libpysal.weights] spatial weights matrix based on unique_id

unique_id [str] name of the column with unique id used as `spatial_weights` index.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['neighbour_distance'] = momepy.NeighborDistance(buildings_df, sw,
↪ 'uID').series
100%|| 144/144 [00:00<00:00, 345.78it/s]
>>> buildings_df['neighbour_distance'][0]
29.18589019096464
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__(*gdf*, *spatial_weights*, *unique_id*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights, unique_id[, ...])	Initialize self.

momepy.NeighboringStreetOrientationDeviation

class momepy.**NeighboringStreetOrientationDeviation**(*gdf*)
Calculate the mean deviation of solar orientation of adjacent streets

Orientation of street segment is represented by the orientation of line connecting first and last point of the segment.

$$\frac{1}{n}\sum_{i=1}^{n} dev_i = \frac{dev_1 + dev_2 + \cdots + dev_n}{n}$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing street network to analyse

Examples

```
>>> streets_df['orient_dev'] = momepy.
↪NeighboringStreetOrientationDeviation(streets_df).series
>>> streets_df['orient_dev'][6]
7.043096518688273
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

orientation [Series] Series containing used street orientation values

__init__(*gdf*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf)	Initialize self.

momepy.Neighbors

class momepy.**Neighbors**(*gdf*, *spatial_weights*, *unique_id*, *weighted=False*, *verbose=True*)
Calculate the number of neighbours captured by `spatial_weights`

If `weighted=True`, number of neighbours will be divided by the perimeter of object to return relative value.

Adapted from [HRRCambraLopez12].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

weighted [bool (default False)] if `True`, number of neighbours will be divided by the perimeter of object, to return relative value

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = libpysal.weights.contiguity.Queen.from_dataframe(tessellation_df, ids=
↪'uID')
>>> tessellation_df['neighbours'] = momepy.Neighbors(tessellation_df, sw, 'uID').
↪series
100%|| 144/144 [00:00<00:00, 6909.50it/s]
>>> tessellation_df['neighbours'][0]
4
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

weighted [bool] used weighted value

__init__(*gdf*, *spatial_weights*, *unique_id*, *weighted=False*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, spatial_weights, unique_id[, ...])	Initialize self.

momepy.Orientation

class momepy.**Orientation**(*gdf*, *verbose=True*)

Calculate the orientation of object

Captures the deviation of orientation from cardinal directions. Defined as an orientation of the longext axis of bounding rectangle in range 0 - 45. Orientation of LineStrings is represented by the orientation of line connecting first and last point of the segment.

Adapted from [SA15].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['orientation'] = momepy.Orientation(buildings_df).series
100%|| 144/144 [00:00<00:00, 630.54it/s]
>>> buildings_df['orientation'][0]
41.05146788287027
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

__init__(*gdf*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, verbose])	Initialize self.

momepy.SharedWallsRatio

class momepy.**SharedWallsRatio**(*gdf*, *unique_id=None*, *perimeters=None*)

Calculate shared walls ratio of adjacent elements (typically buildings)

$$\frac{length\ of\ shared\ walls}{perimeter}$$

Note that data needs to be topologically correct. Overlapping polygons will lead to incorrect results.

Adapted from [HLM12].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing gdf to analyse

unique_id [(deprecated)]

perimeters [str, list, np.array, pd.Series (default None)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored perimeter value

Examples

```
>>> buildings_df['swr'] = momepy.SharedWallsRatio(buildings_df).series
>>> buildings_df['swr'][10]
0.3424804411228673
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

perimeters [GeoDataFrame] Series containing used perimeters values

__init__(*gdf*, *unique_id=None*, *perimeters=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, unique_id, perimeters])	Initialize self.

momepy.StreetAlignment

class momepy.**StreetAlignment**(*left*, *right*, *orientations*, *network_id=None*, *left_network_id=None*, *right_network_id=None*)
Calculate the difference between street orientation and orientation of object in degrees

Orientation of street segment is represented by the orientation of line connecting first and last point of the segment. Network ID linking each object to specific street segment is needed. Can be generated by *momepy.get_network_id()*. Either `network_id` or both `left_network_id` and `right_network_id` are required.

$$|\textit{building orientation} - \textit{street orientation}|$$

Parameters

left [GeoDataFrame] GeoDataFrame containing objects to analyse

right [GeoDataFrame] GeoDataFrame containing street network

orientations [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored object orientation value (can be calculated using *momepy.Orientation*)

network_id [str (default None)] the name of the column storing network ID in both left and right

left_network_id [str, list, np.array, pd.Series (default None)] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored object network ID

right_network_id [str, list, np.array, pd.Series (default None)] the name of the right dataframe column, `np.array`, or `pd.Series` of streets with unique network id (has to be defined beforehand) (can be defined using *momepy.unique_id()*)

Examples

```
>>> buildings_df['street_alignment'] = momepy.StreetAlignment(buildings_df,␣
↪streets_df, 'orientation', 'nID', 'nID').series
>>> buildings_df['street_alignment'][0]
0.29073888476702336
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

network_id [str] the name of the column storing network ID in both left and right

left_network_id [Series] Series containing used left ID

right_network_id [Series] Series containing used right ID

__init__(*left*, *right*, *orientations*, *network_id=None*, *left_network_id=None*, *right_network_id=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, orientations[, ...])	Initialize self.

8.3.5 intensity

AreaRatio(left, right, left_areas, right_areas)	Calculate covered area ratio or floor area ratio of objects.
BlocksCount(gdf, block_id, spatial_weights, ...)	Calculates the weighted number of blocks
Count(left, right, left_id, right_id[, weighted])	Calculate the number of elements within an aggregated structure.
Courtyards(gdf[, block_id, spatial_weights, ...])	Calculate the number of courtyards within the joined structure.
Density(gdf, values, spatial_weights, unique_id)	Calculate the gross density
NodeDensity(left, right, spatial_weights[, ...])	Calculate the density of nodes neighbours on street network defined in `spatial_weights`.
Reached(left, right, left_id, right_id[, ...])	Calculates the number of objects reached within neighbours on street network

momepy.AreaRatio

class momepy.**AreaRatio**(*left*, *right*, *left_areas*, *right_areas*, *unique_id=None*, *left_unique_id=None*, *right_unique_id=None*)

Calculate covered area ratio or floor area ratio of objects.

Either `unique_id` or both `left_unique_id` and `right_unique_id` are required.

$$\frac{\textit{covering object area}}{\textit{covered object area}}$$

Adapted from [SA15].

Parameters

left [GeoDataFrame] GeoDataFrame containing objects being covered (e.g. land unit)

right [GeoDataFrame] GeoDataFrame with covering objects (e.g. building)

left_areas [str, list, np.array, pd.Series] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored area value.

right_areas [str, list, np.array, pd.Series] the name of the right dataframe column, `np.array`, or `pd.Series` where is stored area value representing either projected or floor area.

unique_id [str (default None)] name of the column with unique id shared amongst left and right gdfs. If there is none, it could be generated by :py:func:'momepy.unique_id()'.

left_unique_id [str, list, np.array, pd.Series (default None)] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored shared unique ID

right_unique_id [str, list, np.array, pd.Series (default None)] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored shared unique ID

Examples

```
>>> tessellation_df['CAR'] = mm.AreaRatio(tessellation_df, buildings_df, 'area',
↪'area', 'uID').series
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

left_areas [Series] Series containing used left areas

right_areas [Series] Series containing used right areas

left_unique_id [Series] Series containing used left ID

right_unique_id [Series] Series containing used right ID

__init__(*left*, *right*, *left_areas*, *right_areas*, *unique_id=None*, *left_unique_id=None*, *right_unique_id=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, left_areas, right_areas)	Initialize self.

momepy.BlocksCount

class momepy.**BlocksCount**(*gdf*, *block_id*, *spatial_weights*, *unique_id*, *weighted=True*, *verbose=True*)

Calculates the weighted number of blocks

Number of blocks within neighbours defined in `spatial_weights`.

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

block_id [str, list, np.array, pd.Series] the name of the objects dataframe column, `np.array`, or `pd.Series` where is stored block ID.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

weigted [bool, default True] return value weighted by the analysed area (`True`) or pure count (`False`)

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw4 = mm.sw_high(k=4, gdf='tessellation_df', ids='uID')
>>> tessellation_df['blocks_within_4'] = mm.BlocksCount(tessellation_df, 'bID',␣
↪sw4, 'uID').series
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

block_id [Series] Series containing used block ID

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

weighted [bool] used weighted value

__init__(*gdf*, *block_id*, *spatial_weights*, *unique_id*, *weighted=True*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, block_id, spatial_weights, ...)	Initialize self.

momepy.Count

class momepy.**Count**(*left*, *right*, *left_id*, *right_id*, *weighted=False*)

Calculate the number of elements within an aggregated structure.

Aggregated structure can be typically block, street segment or street node (their snapepd objects). Right gdf has to have unique id of aggregated structure assigned before hand (e.g. using *momepy.get_network_id()*). If `weighted=True`, number of elements will be divided by the area of length (based on geometry type) of aggregated element, to return relative value.

$$\sum_{i \in aggr} (n_i); \frac{\sum_{i \in aggr}(n_i)}{area_{aggr}}$$

Adapted from [HRRCambraLopez12] and [Fel18].

Parameters

left [GeoDataFrame] GeoDataFrame containing aggregation to analyse

right [GeoDataFrame] GeoDataFrame containing objects to analyse

left_id [str] name of the column where is stored unique ID in left gdf

right_id [str] name of the column where is stored unique ID of aggregation in right gdf

weighted [bool (default False)] if `True`, count will be divided by the area or length

Examples

```
>>> blocks_df['buildings_count'] = mm.Count(blocks_df, buildings_df, 'bID', 'bID',
↪ weighted=True).series
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

left_id [Series] Series containing used left ID

right_id [Series] Series containing used right ID

weighted [bool] used weighted value

__init__(*left*, *right*, *left_id*, *right_id*, *weighted=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, left_id, right_id[, ...])	Initialize self.

momepy.Courtyards

class momepy.**Courtyards**(*gdf*, *block_id=None*, *spatial_weights=None*, *verbose=True*)

Calculate the number of courtyards within the joined structure.

Adapted from [SA15].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

block_id [(deprecated)]

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix will be calculated based on objects. It is to denote adjacent buildings (note: based on integer index).

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> buildings_df['courtyards'] = mm.Courtyards(buildings_df).series
Calculating spatial weights...
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

sw [libpysal.weights] spatial weights matrix

__init__(*gdf*, *block_id=None*, *spatial_weights=None*, *verbose=True*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, block_id, spatial_weights, ...])	Initialize self.

momepy.Density

class momepy.**Density**(*gdf*, *values*, *spatial_weights*, *unique_id*, *areas=None*, *verbose=True*)

Calculate the gross density

$$\frac{\sum \text{values}}{\sum \text{areas}}$$

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weight] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

areas [str, list, np.array, pd.Series (optional)] the name of the dataframe column, `np.array`, or `pd.Series` where is stored area value. If None, gdf.geometry.area will be used.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> tessellation_df['floor_area_dens'] = mm.Density(tessellation_df, 'floor_area',
↪ sw, 'uID').series
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

areas [Series] Series containing used area values

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *areas=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.NodeDensity

class momepy.**NodeDensity**(*left*, *right*, *spatial_weights*, *weighted=False*, *node_degree=None*, *node_start='node_start'*, *node_end='node_end'*, *verbose=True*)

Calculate the density of nodes neighbours on street network defined in `spatial_weights`.

Calculated as number of neighbouring nodes / cummulative length of street network within neighbours. node_start and node_end is standard output of *momepy.nx_to_gdf()* and is compulsory.

Adapted from [DPR+17].

Parameters

left [GeoDataFrame] GeoDataFrame containing nodes of street network

right [GeoDataFrame] GeoDataFrame containing edges of street network

spatial_weights [libpysal.weights] spatial weights matrix capturing relationship between nodes

weighted [bool (default False)] if True density will take into account node degree as `k-1`

node_degree [str (optional)] name of the column of left gdf containing node degree. Used if `weighted=True`

node_start [str (default 'node_start')] name of the column of right gdf containing id of starting node

node_end [str (default 'node_end')] name of the column of right gdf containing id of ending node

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> nodes['density'] = mm.NodeDensity(nodes, edges, sw).series
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

node_start [Series] Series containing used ids of starting node

node_end [Series] Series containing used ids of ending node

sw [libpysal.weights] spatial weights matrix

weighted [bool] used weighted value

node_degree [Series] Series containing used node degree values

__init__(*left*, *right*, *spatial_weights*, *weighted=False*, *node_degree=None*, *node_start='node_start'*, *node_end='node_end'*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, spatial_weights[, ...])	Initialize self.

momepy.Reached

class momepy.**Reached**(*left*, *right*, *left_id*, *right_id*, *spatial_weights=None*, *mode='count'*, *values=None*, *verbose=True*)
Calculates the number of objects reached within neighbours on street network

Number of elements within neighbourhood defined in `spatial_weights`. If `spatial_weights` are `None`, it will assume topological distance `0` (element itself). If `mode='area'`, returns sum of areas of reached elements. Requires unique_id of network assigned beforehand (e.g. using *momepy.get_network_id()*).

Parameters

left [GeoDataFrame] GeoDataFrame containing streets (either segments or nodes)

right [GeoDataFrame] GeoDataFrame containing elements to be counted

left_id [str, list, np.array, pd.Series (default None)] the name of the left dataframe column, `np.array`, or `pd.Series` where is stored ID of streets (segments or nodes).

right_id [str, list, np.array, pd.Series (default None)] the name of the right dataframe column, `np.array`, or `pd.Series` where is stored ID of streets (segments or nodes).

spatial_weights [libpysal.weights (default None)] spatial weights matrix

mode [str (default 'count')] mode of calculation. If `'count'` function will return the count of reached elements. If `'sum'`, it will return sum of `'values'`. If `'mean'` it will return mean value of `'values'`. If `'std'` it will return standard deviation of `'values'`. If `'values'` not set it will use of areas of reached elements.

values [str (default None)] the name of the objects dataframe column with values used for calculations

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> streets_df['reached_buildings'] = mm.Reached(streets_df, buildings_df, 'uID').
↪series
```

Attributes

series [Series] Series containing resulting values

left [GeoDataFrame] original left GeoDataFrame

right [GeoDataFrame] original right GeoDataFrame

left_id [Series] Series containing used left ID

right_id [Series] Series containing used right ID

mode [str] mode of calculation

sw [libpysal.weights] spatial weights matrix (if set)

__init__(*left*, *right*, *left_id*, *right_id*, *spatial_weights=None*, *mode='count'*, *values=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(left, right, left_id, right_id[, ...])	Initialize self.

8.3.6 graph

betweenness_centrality(graph[, name, mode, ...])	Calculates the shortest-path betweenness centrality for nodes.
cds_length(graph[, radius, mode, name, ...])	Calculates length of cul-de-sacs for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
closeness_centrality(graph[, name, weight, ...])	Calculates the closeness centrality for nodes.
clustering(graph[, name])	Calculates the squares clustering coefficient for nodes.
cyclomatic(graph[, radius, name, distance, ...])	Calculates cyclomatic complexity for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
edge_node_ratio(graph[, radius, name, ...])	Calculates edge / node ratio for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
gamma(graph[, radius, name, distance, verbose])	Calculates connectivity gamma index for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
mean_node_degree(graph[, radius, name, ...])	Calculates mean node degree for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
mean_node_dist(graph[, name, length, verbose])	Calculates mean distance to neighbouring nodes.
mean_nodes(G, attr)	Calculates mean value of nodes attr for each edge.
meshedness(graph[, radius, name, distance, ...])	Calculates meshedness for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
node_degree(graph[, name])	Calculates node degree for each node.
proportion(graph[, radius, three, four, ...])	Calculates the proportion of intersection types for subgraph around each node if radius is set, or for whole graph, if `radius=None`.
straightness_centrality(graph[, weight, ...])	Calculates the straightness centrality for nodes.
subgraph(graph[, radius, distance, ...])	Calculates all subgraph-based characters.

momepy.betweenness_centrality

momepy.**betweenness_centrality**(*graph*, *name='betweenness'*, *mode='nodes'*, *weight='mm_len'*, *endpoints=True*, *radius=None*, *distance=None*, *normalized=False*, *verbose=True*, ***kwargs*)

Calculates the shortest-path betweenness centrality for nodes.

Wrapper around `networkx.betweenness_centrality` or `networkx.edge_betweenness_centrality`.

Betweenness centrality of a node *v* is the sum of the fraction of all-pairs shortest paths that pass through *v*

$$c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

where *V* is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|v)$ is the number of those paths passing through some node *v* other than *s, t*. If *s = t*, $\sigma(s,t) = 1$, and if *v* in *{s, t}`*, $\sigma(s,t|v) = 0$.

Betweenness centrality of an edge e is the sum of the fraction of all-pairs shortest paths that pass through e

$$c_B(e) = \sum_{s,t \in V} \frac{\sigma(s,t|e)}{\sigma(s,t)}$$

where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|e)$ is the number of those paths passing through edge e.

Adapted from [PCL06].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

name [str, optional] calculated attribute name

mode [str, default 'nodes'] mode of betweenness calculation. 'node' for node-based, 'edges' for edge-based

weight [str (default 'mm_len')] attribute holding the weight of edge (e.g. length, angle)

radius: int Include all neighbors of distance <= radius from n

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n during ego_graph generation.

normalized [bool, optional] If True the betweenness values are normalized by *2/((n-1)(n-2))*, where n is the number of nodes in subgraph.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

****kwargs** kwargs for `networkx.betweenness_centrality` or `networkx.edge_betweenness_centrality`

Returns

Graph networkx.Graph

Notes

In case of angular betweenness, implementation is based on "Tasos Implementation".

Examples

```
>>> network_graph = mm.betweenness_centrality(network_graph)
```

momepy.cds_length

momepy.**cds_length**(*graph*, *radius=5*, *mode='sum'*, *name='cds_len'*, *degree='degree'*, *length='mm_len'*, *distance=None*, *verbose=True*)

Calculates length of cul-de-sacs for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in distance attribute.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius [int] Include all neighbors of distance <= radius from n

mode [str (default 'sum')] if `'sum'`, calculate total length, if `'mean'` calculate mean length

name [str, optional] calculated attribute name

degree [str] name of attribute of node degree (*momepy.node_degree()*)

length [str, optional] name of attribute of segment length (geographical)

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float length of cul-de-sacs for graph if `radius=None`

Examples

```
>>> network_graph = mm.cds_length(network_graph, radius=9, mode='mean')
```

momepy.closeness_centrality

momepy.**closeness_centrality**(*graph*, *name='closeness'*, *weight='mm_len'*, *radius=None*, *distance=None*, *verbose=True*, ***kwargs*)

Calculates the closeness centrality for nodes.

Wrapper around `networkx.closeness_centrality`.

Closeness centrality of a node *u* is the reciprocal of the average shortest path distance to *u* over all *n-1* nodes within reachable nodes.

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

where $d(v,u)$ is the shortest-path distance between v and u, and n is the number of nodes that can reach u.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

name [str, optional] calculated attribute name

weight [str (default 'mm_len')] attribute holding the weight of edge (e.g. length, angle)

radius: int Include all neighbors of distance <= radius from n

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n during ego_graph generation.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

****kwargs** kwargs for `networkx.closeness_centrality`

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.closeness_centrality(network_graph)
```

momepy.clustering

momepy.**clustering**(*graph*, *name='cluster'*)

Calculates the squares clustering coefficient for nodes.

Wrapper around `networkx.square_clustering`.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

name [str, optional] calculated attribute name

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.clustering(network_graph)
```

momepy.cyclomatic

momepy.**cyclomatic**(*graph*, *radius=5*, *name='cyclomatic'*, *distance=None*, *verbose=True*)

Calculates cyclomatic complexity for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in `distance` attribute.

$$\alpha = e - v + 1$$

where e is the number of edges in subgraph and v is the number of nodes in subgraph.

Adapted from [BSN12].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int Include all neighbors of distance <= radius from n

name [str, optional] calculated attribute name

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float cyclomatic complexity for graph if `radius=None`

Examples

```
>>> network_graph = mm.cyclomatic(network_graph, radius=3)
```

momepy.edge_node_ratio

momepy.**edge_node_ratio**(*graph*, *radius=5*, *name='edge_node_ratio'*, *distance=None*, *verbose=True*)

Calculates edge / node ratio for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in `distance` attribute.

$$\alpha = e/v$$

where e is the number of edges in subgraph and v is the number of nodes in subgraph.

Adapted from [DPR+17].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int Include all neighbors of distance <= radius from n

name [str, optional] calculated attribute name

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float edge / node ratio for graph if `radius=None`

Examples

```
>>> network_graph = mm.edge_node_ratio(network_graph, radius=3)
```

momepy.gamma

momepy.**gamma**(*graph*, *radius=5*, *name='gamma'*, *distance=None*, *verbose=True*)

Calculates connectivity gamma index for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in `distance` attribute.

$$\alpha = \frac{e}{3(v-2)}$$

where e is the number of edges in subgraph and v is the number of nodes in subgraph.

Adapted from [DPR+17].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int Include all neighbors of distance <= radius from n

name [str, optional] calculated attribute name

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float gamma index for graph if `radius=None`

Examples

```
>>> network_graph = mm.gamma(network_graph, radius=3)
```

momepy.mean_node_degree

momepy.**mean_node_degree**(*graph*, *radius=5*, *name='mean_nd'*, *degree='degree'*, *distance=None*, *verbose=True*)

Calculates mean node degree for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in `distance` attribute.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int radius defining the extent of subgraph

name [str, optional] calculated attribute name

degree [str] name of attribute of node degree (*momepy.node_degree()*)

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float mean node degree for graph if `radius=None`

Examples

```
>>> network_graph = mm.mean_node_degree(network_graph, radius=3)
```

momepy.mean_node_dist

momepy.**mean_node_dist**(*graph*, *name='meanlen'*, *length='mm_len'*, *verbose=True*)

Calculates mean distance to neighbouring nodes.

Mean of values in `length` attribute.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

name [str, optional] calculated attribute name

length [str, optional] name of attribute of segment length (geographical)

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.mean_node_dist(network_graph)
```

momepy.mean_nodes

momepy.**mean_nodes**(*G*, *attr*)

Calculates mean value of nodes attr for each edge.

momepy.meshedness

momepy.**meshedness**(*graph*, *radius=5*, *name='meshedness'*, *distance=None*, *verbose=True*)

Calculates meshedness for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in distance attribute.

$$\alpha = \frac{e - v + 1}{2v - 5}$$

where e is the number of edges in subgraph and v is the number of nodes in subgraph.

Adapted from [Fel18].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int, optional Include all neighbors of distance <= radius from n

name [str, optional] calculated attribute name

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

float meshedness for graph if `radius=None`

Examples

```
>>> network_graph = mm.meshedness(network_graph, radius=800, distance='edge_length
↪')
```

momepy.node_degree

momepy.**node_degree**(*graph*, *name='degree'*)

Calculates node degree for each node.

Wrapper around `networkx.degree()`.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

name [str (default 'degree')] calculated attribute name

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.node_degree(network_graph)
```

momepy.proportion

momepy.**proportion**(*graph*, *radius=5*, *three=None*, *four=None*, *dead=None*, *degree='degree'*, *distance=None*, *verbose=True*)

Calculates the proportion of intersection types for subgraph around each node if radius is set, or for whole graph, if `radius=None`.

Subgraph is generated around each node within set radius. If `distance=None`, radius will define topological distance, otherwise it uses values in `distance` attribute.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int Include all neighbors of distance <= radius from n

three [str, optional] attribute name for 3-way intersections proportion

four [str, optional] attribute name for 4-way intersections proportion

dead [str, optional] attribute name for deadends proportion

degree [str] name of attribute of node degree (*momepy.node_degree()*)

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph if radius is set

dict dict with proportions for graph if `radius=None`

Examples

```
>>> network_graph = mm.proportion(network_graph, three='threeway', four='fourway',
↪ dead='deadends')
```

momepy.straightness_centrality

momepy.**straightness_centrality**(*graph*, *weight='mm_len'*, *normalized=True*, *name='straightness'*, *radius=None*, *distance=None*, *verbose=True*)

Calculates the straightness centrality for nodes.

$$C_S(i) = \frac{1}{n-1} \sum_{j \in V, j \neq i} \frac{d_{ij}^{Eu}}{d_{ij}}$$

where $\mathrm{d}_{\mathrm{ij}}^{\mathrm{Eu}}$ is the Euclidean distance between nodes *i* and *j* along a straight line.

Adapted from [PCL06].

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

weight [str (default 'mm_len')] attribute holding length of edge

normalized [bool] normalize to number of nodes-1 in connected part (for local straightness is recommended to set to normalized False)

name [str, optional] calculated attribute name

radius: int Include all neighbors of distance <= radius from n

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n during ego_graph generation.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.straightness_centrality(network_graph)
```

momepy.subgraph

momepy.**subgraph**(*graph*, *radius=5*, *distance=None*, *meshedness=True*, *cds_length=True*, *mode='sum'*, *degree='degree'*, *length='mm_len'*, *mean_node_degree=True*, *proportion={0: True, 3: True, 4: True}*, *cyclomatic=True*, *edge_node_ratio=True*, *gamma=True*, *local_closeness=True*, *closeness_weight=None*, *verbose=True*)

Calculates all subgraph-based characters.

Generating subgraph might be a time consuming activity. If we want to use the same subgraph for more characters, `subgraph` allows this by generating subgraph and then analysing it using selected options.

Parameters

graph [networkx.Graph] Graph representing street network. Ideally generated from GeoDataFrame using *momepy.gdf_to_nx()*

radius: int radius defining the extent of subgraph

distance [str, optional] Use specified edge data key as distance. For example, setting `distance='weight'` will use the edge `weight` to measure the distance from the node n.

meshedness [bool, default True] Calculate meshedness (True/False)

cds_length [bool, default True] Calculate cul-de-sac length (True/False)

mode [str (defualt 'sum')] if `'sum'`, calculate total cds_length, if `'mean'` calculate mean cds_length

degree [str] name of attribute of node degree (*momepy.node_degree()*)

length [str, default *mm_len*] name of attribute of segment length (geographical)

mean_node_degree [bool, default True] Calculate mean node degree (True/False)

proportion [dict, default {3: True, 4: True, 0: True}] Calculate proportion {3: True/False, 4: True/False, 0: True/False}

cyclomatic [bool, default True] Calculate cyclomatic complexity (True/False)

edge_node_ratio [bool, default True] Calculate edge node ratio (True/False)

gamma [bool, default True] Calculate gamma index (True/False)

local_closeness [bool, default True] Calculate local closeness centrality (True/False)

closeness_weight [str, optional] Use the specified edge attribute as the edge distance in shortest path calculations in closeness centrality algorithm

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

Graph networkx.Graph

Examples

```
>>> network_graph = mm.subgraph(network_graph)
```

8.3.7 diversity

Gini(gdf, values, spatial_weights, unique_id)	Calculates the Gini index of values within neighbours defined in `spatial_weights`.
Percentiles(gdf, values, spatial_weights, ...)	Calculates the percentiles of values within neighbours defined in `spatial_weights`.
Range(gdf, values, spatial_weights, unique_id)	Calculates the range of values within neighbours defined in `spatial_weights`.
Shannon(gdf, values, spatial_weights, unique_id)	Calculates the Shannon index of values within neighbours defined in `spatial_weights`.
Simpson(gdf, values, spatial_weights, unique_id)	Calculates the Simpson's diversity index of values within neighbours defined in `spatial_weights`.
Theil(gdf, values, spatial_weights, unique_id)	Calculates the Theil measure of inequality of values within neighbours defined in `spatial_weights`.
Unique(gdf, values, spatial_weights, unique_id)	Calculates the number of unique values within neighbours defined in `spatial_weights`.
shannon_diversity(data[, bins, categorical, ...])	Calculates the Shannon's diversity index of data.
simpson_diversity(data[, bins, categorical, ...])	Calculates the Simpson's diversity index of data.

momepy.Gini

class momepy.**Gini**(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *verbose=True*)

Calculates the Gini index of values within neighbours defined in `spatial_weights`.

Uses `inequality.gini.Gini` under the hood. Requires '*inequality*' package.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

rng [Two-element sequence containing floats in range of [0,100], optional] Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The order of the elements is not important.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['area_Gini'] = mm.Gini(tessellation_df, 'area', sw, 'uID').
↪series
100%|| 144/144 [00:00<00:00, 597.37it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

rng [tuple] range

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Percentiles

class momepy.**Percentiles**(*gdf*, *values*, *spatial_weights*, *unique_id*, *percentiles=[25, 50, 75]*, *interpolation='midpoint'*, *verbose=True*)
Calculates the percentiles of values within neighbours defined in `spatial_weights`.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

percentiles [array-like (default [25, 50, 75])] percentiles to return

interpolation [{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use when the desired percentile lies between two data points `i < j`: * 'linear': `i + (j - i) * fraction`, where `fraction`

is the fractional part of the index surrounded by `i` and `j`.

- 'lower': `i`.
- 'higher': `j`.
- 'nearest': `i` or `j`, whichever is nearest.
- 'midpoint': `(i + j) / 2`.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['cluster_unique'] = mm.Percentiles(tessellation_df, 'cluster',
↪ sw, 'uID').series
100%|| 144/144 [00:00<00:00, 722.50it/s]
```

Attributes

frame [DataFrame] DataFrame containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *percentiles=[25, 50, 75]*, *interpolation='midpoint'*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Range

class momepy.**Range**(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=0, 100*, *verbose=True*, ***kwargs*)
Calculates the range of values within neighbours defined in `spatial_weights`.

Uses `scipy.stats.iqr` under the hood.

Adapted from [DPR+17].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

rng [Two-element sequence containing floats in range of [0,100], optional] Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The order of the elements is not important.

****kwargs** [keyword arguments] optional arguments for `scipy.stats.iqr`

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['area_IQR_3steps'] = mm.Range(tessellation_df, 'area', sw,
↪'uID', rng=(25, 75)).series
100%|| 144/144 [00:00<00:00, 722.50it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

rng [tuple] range

kwargs [dict] kwargs

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=0, 100*, *verbose=True*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Shannon

class momepy.**Shannon**(*gdf*, *values*, *spatial_weights*, *unique_id*, *binning='HeadTailBreaks'*, *categorical=False*, *categories=None*, *verbose=True*, ***classification_kwds*)
Calculates the Shannon index of values within neighbours defined in `spatial_weights`.

Uses `mapclassify.classifiers` under the hood for binning. Requires `mapclassify>=.2.1.0` dependency.

$$H' = -\sum_{i=1}^{R} p_i \ln p_i$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix of set order will be calculated based on objects.

unique_id [str] name of the column with unique id used as `spatial_weights` index

binning [str] One of mapclassify classification schemes. For details see mapclassify API documentation.

categorical [bool (default False)] treat values as categories (will not use binning)

categories [list-like (default None)] list of categories. If None values.unique() is used.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

****classification_kwds** [dict] Keyword arguments for classification scheme For details see mapclassify documentation.

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['area_Shannon'] = mm.Shannon(tessellation_df, 'area', sw, 'uID
↪').series
100%|| 144/144 [00:00<00:00, 455.83it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

binning [str] binning method

bins [mapclassify.classifiers.Classifier] generated bins

classification_kwds [dict] classification_kwds

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *binning='HeadTailBreaks'*, *categorical=False*, *categories=None*, *verbose=True*, ***classification_kwds*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Simpson

class momepy.**Simpson**(*gdf*, *values*, *spatial_weights*, *unique_id*, *binning='HeadTailBreaks'*, *gini_simpson=False*, *inverse=False*, *categorical=False*, *categories=None*, *verbose=True*, ***classification_kwds*)

Calculates the Simpson's diversity index of values within neighbours defined in `spatial_weights`.

Uses `mapclassify.classifiers` under the hood for binning. Requires `mapclassify>=.2.1.0` dependency.

$$\lambda = \sum_{i=1}^{R} p_i^2$$

Adapted from [Fel18].

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights, optional] spatial weights matrix - If None, Queen contiguity matrix of set order will be calculated based on objects.

unique_id [str] name of the column with unique id used as `spatial_weights` index

binning [str (default 'HeadTailBreaks')] One of mapclassify classification schemes. For details see mapclassify API documentation.

gini_simpson [bool (default False)] return Gini-Simpson index instead of Simpson index (`1 -`)

inverse [bool (default False)] return Inverse Simpson index instead of Simpson index (`1 /`)

categorical [bool (default False)] treat values as categories (will not use `binning`)

categories [list-like (default None)] list of categories. If None `values.unique()` is used.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

****classification_kwds** [dict] Keyword arguments for classification scheme For details see mapclassify documentation.

See also:

momepy.simpson_diversity Calculates the Simpson's diversity index of data

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['area_Simpson'] = mm.Simpson(tessellation_df, 'area', sw, 'uID
↪').series
100%|| 144/144 [00:00<00:00, 455.83it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

binning [str] binning method

bins [mapclassify.classifiers.Classifier] generated bins

classification_kwds [dict] classification_kwds

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *binning='HeadTailBreaks'*, *gini_simpson=False*, *inverse=False*, *categorical=False*, *categories=None*, *verbose=True*, ***classification_kwds*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Theil

class momepy.**Theil**(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *verbose=True*)
Calculates the Theil measure of inequality of values within neighbours defined in `spatial_weights`.

Uses `inequality.theil.Theil` under the hood. Requires '*inequality*' package.

$$T = \sum_{i=1}^{n} \left(\frac{y_i}{\sum_{i=1}^{n} y_i} \ln \left[N \frac{y_i}{\sum_{i=1}^{n} y_i} \right] \right)$$

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

rng [Two-element sequence containing floats in range of [0,100], optional] Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The order of the elements is not important.

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['area_Theil'] = mm.Theil(tessellation_df, 'area', sw, 'uID').
↪series
100%|| 144/144 [00:00<00:00, 597.37it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

rng [tuple, optional] range

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *rng=None*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.Unique

class momepy.**Unique**(*gdf*, *values*, *spatial_weights*, *unique_id*, *verbose=True*)
Calculates the number of unique values within neighbours defined in `spatial_weights`.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing morphological tessellation

values [str, list, np.array, pd.Series] the name of the dataframe column, `np.array`, or `pd.Series` where is stored character value.

spatial_weights [libpysal.weights] spatial weights matrix

unique_id [str] name of the column with unique id used as `spatial_weights` index

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Examples

```
>>> sw = momepy.sw_high(k=3, gdf=tessellation_df, ids='uID')
>>> tessellation_df['cluster_unique'] = mm.Unique(tessellation_df, 'cluster', sw,
↪'uID').series
100%|| 144/144 [00:00<00:00, 722.50it/s]
```

Attributes

series [Series] Series containing resulting values

gdf [GeoDataFrame] original GeoDataFrame

values [Series] Series containing used values

sw [libpysal.weights] spatial weights matrix

id [Series] Series containing used unique ID

__init__(*gdf*, *values*, *spatial_weights*, *unique_id*, *verbose=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(gdf, values, spatial_weights, unique_id)	Initialize self.

momepy.shannon_diversity

momepy.**shannon_diversity**(*data*, *bins=None*, *categorical=False*, *categories=None*)

Calculates the Shannon's diversity index of data. Helper function for *momepy.Shannon*.

$$\lambda = \sum_{i=1}^{R} p_i^2$$

Formula adapted from https://gist.github.com/audy/783125

Parameters

data [GeoDataFrame] GeoDataFrame containing morphological tessellation

bins [array, optional] array of top edges of classification bins. Result of binnng.bins.

categorical [bool (default False)] treat values as categories (will not use `bins`)

categories [list-like (default None)] list of categories

Returns

float Shannon's diversity index

See also:

momepy.Shannon Calculates the Shannon's diversity index of values within neighbours

momepy.Simpson Calculates the Simpson's diversity index of values within neighbours

momepy.simpson_diversity Calculates the Simpson's diversity index of data

momepy.simpson_diversity

momepy.**simpson_diversity**(*data*, *bins=None*, *categorical=False*, *categories=None*)

Calculates the Simpson's diversity index of data. Helper function for *momepy.Simpson*.

$$\lambda = \sum_{i=1}^{R} p_i^2$$

Formula adapted from https://gist.github.com/martinjc/f227b447791df8c90568.

Parameters

data [GeoDataFrame] GeoDataFrame containing morphological tessellation

bins [array, optional] array of top edges of classification bins. Result of binnng.bins.

categorical [bool (default False)] treat values as categories (will not use `bins`)

categories [list-like (default None)] list of categories

Returns

float Simpson's diversity index

See also:

momepy.Simpson Calculates the Simpson's diversity index of values within neighbours

8.3.8 spatial weights

DistanceBand(gdf, threshold[, centroid, ids])	On demand distance-based spatial weights-like class.
sw_high(k[, gdf, weights, ids, contiguity, ...])	Generate spatial weights based on Queen or Rook contiguity of order k.

momepy.DistanceBand

class momepy.**DistanceBand**(*gdf*, *threshold*, *centroid=True*, *ids=None*)

On demand distance-based spatial weights-like class.

Mimic the behavior of `libpysal.weights.DistanceBand` but do not compute all neighbors at once but only on demand. Only `DistanceBand.neighbors[key]` is implemented. Once user asks for `DistanceBand.neighbors[key]`, neighbors for specified key will be computed using rtree. The algorithm is significantly slower than `libpysal.weights.DistanceBand` but allows for large number of neighbors which may cause memory issues in libpysal.

Use `libpysal.weights.DistanceBand` if possible. `momepy.weights.DistanceBand` only when necessary. `DistanceBand.neighbors[key]` should yield same results as *momepy.DistanceBand*.

Parameters

gdf [GeoDataFrame or GeoSeries] GeoDataFrame containing objects to be used

threshold [float] distance band to be used as buffer

centroid [bool (default True)] use centroid of geometry (as in `libpysal.weights.DistanceBand`). If `False`, works with the geometry as it is.

ids [str] column to be used as geometry ids. If not set, integer position is used.

Attributes

neighbors[key] [list] list of ids of neighboring features

Methods

fetch_items	

__init__(*gdf*, *threshold*, *centroid=True*, *ids=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(gdf, threshold[, centroid, ids])	Initialize self.
`fetch_items`(key)	

momepy.sw_high

momepy.**sw_high**(*k*, *gdf=None*, *weights=None*, *ids=None*, *contiguity='queen'*, *silent=True*)

Generate spatial weights based on Queen or Rook contiguity of order k.

Adjacent are all features within <= k steps. Pass either `gdf` or `weights`. If both are passed, `weights` is used. If `weights` are passed, `contiguity` is ignored and high order spatial weights based on `weights` are computed.

Parameters

k [int] order of contiguity

gdf [GeoDataFrame] GeoDataFrame containing objects to analyse. Index has to be consecutive range `0:x`. Otherwise, spatial weights will not match objects.

weights [libpysal.weights] libpysal.weights of order 1

contiguity [str (default 'queen')] type of contiguity weights. Can be `'queen'` or `'rook'`.

silent [bool (default True)] silence libpysal islands warnings

Returns

libpysal.weights libpysal.weights object

Examples

```
>>> first_order = libpysal.weights.Queen.from_dataframe(geodataframe)
>>> first_order.mean_neighbors
5.848032564450475
>>> fourth_order = sw_high(k=4, gdf=geodataframe)
>>> fourth.mean_neighbors
85.73188602442333
```

8.3.9 preprocessing

`close_gaps`(gdf, tolerance)	Close gaps in LineString geometry where it should be contiguous.
`CheckTessellationInput`(gdf[, shrink, ...])	Check input data for `Tessellation` for potential errors.
`extend_lines`(gdf, tolerance[, target, ...])	Extends lines from gdf to istelf or target within a set tolerance
`remove_false_nodes`(gdf)	Clean topology of existing LineString geometry by removal of nodes of degree 2.
`preprocess`(buildings[, size, compactness, ...])	Preprocesses building geometry to eliminate additional structures being single features.

continues on next page

Table 64 – continued from previous page

snap_street_network_edge(edges, buildings, ...)	Fix street network before performing *momepy.Blocks*.

momepy.close_gaps

momepy.**close_gaps**(*gdf*, *tolerance*)

Close gaps in LineString geometry where it should be contiguous.

Snaps both lines to a centroid of a gap in between.

Parameters

gdf [GeoDataFrame, GeoSeries] GeoDataFrame or GeoSeries containing LineString representation of a network.

tolerance [float] nodes within a tolerance will be snapped together

Returns

GeoSeries

See also:

momepy.extend_lines

momepy.remove_false_nodes

momepy.CheckTessellationInput

class momepy.**CheckTessellationInput**(*gdf*, *shrink=0.4*, *collapse=True*, *split=True*, *overlap=True*)

Check input data for *Tessellation* for potential errors.

Tessellation requires data of relatively high level of precision and there are three particular patterns causing issues.

1. Features will collapse into empty polygon - these do not have tessellation cell in the end.

2. Features will split into MultiPolygon - at some cases, features with narrow links between parts split into two during 'shrinking'. In most cases that is not an issue and resulting tessellation is correct anyway, but sometimes this result in a cell being MultiPolygon, which is not correct.

3. Overlapping features - features which overlap even after 'shrinking' cause invalid tessellation geometry.

CheckTessellationInput will check for all of these. Overlapping features have to be fixed prior Tessellation. Features which will split will cause issues only sometimes, so should be checked and fixed if necessary. Features which will collapse could be ignored, but they will have to excluded from next steps of tessellation-based analysis.

Parameters

gdf [GeoDataFrame or GeoSeries] GeoDataFrame containing objects to be used as `gdf` in *Tessellation*

shrink [float (default 0.4)] distance for negative buffer

collapse [bool (default True)] check for features which would collapse to empty polygon

split [bool (default True)] check for features which would split into Multi-type

overlap [bool (default True)] check for overlapping features (after negative buffer)

Examples

```
>>> check = CheckTessellationData(df)
Collapsed features  : 3157
Split features      : 519
Overlapping features: 22
```

Attributes

collapse [GeoDataFrame or GeoSeries] features which would collapse to empty polygon

split [GeoDataFrame or GeoSeries] features which would split into Multi-type

overlap [GeoDataFrame or GeoSeries] overlapping features (after negative buffer)

__init__(*gdf*, *shrink=0.4*, *collapse=True*, *split=True*, *overlap=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(gdf[, shrink, collapse, split, overlap])	Initialize self.

momepy.extend_lines

momepy.**extend_lines**(*gdf*, *tolerance*, *target=None*, *barrier=None*, *extension=0*)
Extends lines from gdf to istelf or target within a set tolerance

Extends unjoined ends of LineString segments to join with other segments or target. If `target` is passed, extend lines to target. Otherwise extend lines to itself.

If `barrier` is passed, each extended line is checked for intersection with `barrier`. If they intersect, extended line is not returned. This can be useful if you don't want to extend street network segments through buildings.

Parameters

gdf [GeoDataFrame] GeoDataFrame containing LineString geometry

tolerance [float] tolerance in snapping (by how much could be each segment extended).

target [GeoDataFrame, GeoSeries] target geometry to which `gdf` gets extended. Has to be (Multi)LineString geometry.

barrier [GeoDataFrame, GeoSeries] extended line is not used if it intersects barrier

extension [float] by how much to extend line beyond the snapped geometry. Useful when creating enclosures to avoid floating point imprecision.

Returns

GeoDataFrame GeoDataFrame of with extended geometry

See also:

momepy.close_gaps

momepy.remove_false_nodes

momepy.remove_false_nodes

momepy.**remove_false_nodes**(*gdf*)

Clean topology of existing LineString geometry by removal of nodes of degree 2.

Parameters

gdf [GeoDataFrame, GeoSeries, array of pygeos geometries] (Multi)LineString data of street network

Returns

gdf [GeoDataFrame, GeoSeries]

See also:

momepy.extend_lines

momepy.close_gaps

momepy.preprocess

momepy.**preprocess**(*buildings*, *size=30*, *compactness=0.2*, *islands=True*, *loops=2*, *verbose=True*)

Preprocesses building geometry to eliminate additional structures being single features.

Certain data providers (e.g. Ordnance Survey in GB) do not provide building geometry as one feature, but divided into different features depending their level (if they are on ground floor or not - passages, overhangs). Ideally, these features should share one building ID on which they could be dissolved. If this is not the case, series of steps needs to be done to minimize errors in morphological analysis.

This script attempts to preprocess such geometry based on several condidions: If feature area is smaller than set size it will be a) deleted if it does not touch any other feature; b) will be joined to feature with which it shares the longest boundary. If feature is fully within other feature, these will be joined. If feature's circular compactness (*momepy.CircularCompactness*) is < 0.2, it will be joined to feature with which it shares the longest boundary. Function does multiple loops through.

Parameters

buildings [geopandas.GeoDataFrame] geopandas.GeoDataFrame containing building layer

size [float (default 30)] maximum area of feature to be considered as additional structure. Set to None if not wanted.

compactness [float (default .2)] if set, function will resolve additional structures identified based on their circular compactness.

islands [bool (default True)] if True, function will resolve additional structures which are fully within other structures (share 100% of exterior boundary).

loops [int (default 2)] number of loops

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

GeoDataFrame GeoDataFrame containing preprocessed geometry

momepy.snap_street_network_edge

momepy.**snap_street_network_edge**(*edges*, *buildings*, *tolerance_street*, *tessellation=None*, *tolerance_edge=None*, *edge=None*, *verbose=True*)

Fix street network before performing *momepy.Blocks*.

Extends unjoined ends of street segments to join with other segments or tessellation boundary.

Parameters

edges [GeoDataFrame] GeoDataFrame containing street network

buildings [GeoDataFrame] GeoDataFrame containing building footprints

tolerance_street [float] tolerance in snapping to street network (by how much could be street segment extended).

tessellation [GeoDataFrame (default None)] GeoDataFrame containing morphological tessellation. If edge is not passed it will be used as edge.

tolerance_edge [float (default None)] tolerance in snapping to edge of tessellated area (by how much could be street segment extended).

edge [Polygon] edge of area covered by morphological tessellation (same as `limit` in *momepy.Tessellation*)

verbose [bool (default True)] if True, shows progress bars in loops and indication of steps

Returns

GeoDataFrame GeoDataFrame of extended street network.

8.3.10 utilities

gdf_to_nx(gdf_network[, approach, length])	Convert LineString GeoDataFrame to networkx.MultiGraph
limit_range(vals, rng)	Extract values within selected range
nx_to_gdf(net[, points, lines, ...])	Convert `networkx.Graph` to LineString GeoDataFrame and Point GeoDataFrame
unique_id(objects)	Add an attribute with unique ID to each row of GeoDataFrame.

momepy.gdf_to_nx

momepy.**gdf_to_nx**(*gdf_network*, *approach='primal'*, *length='mm_len'*)

Convert LineString GeoDataFrame to networkx.MultiGraph

Parameters

gdf_network [GeoDataFrame] GeoDataFrame containing objects to convert

approach [str, default 'primal'] Decide wheter genereate `'primal'` or `'dual'` graph.

length [str, default mm_len] name of attribute of segment length (geographical) which will be saved to graph

Returns

networkx.Graph Graph

momepy.limit_range

momepy.**limit_range**(*vals*, *rng*)
Extract values within selected range

Parameters

vals [array]

rng [Two-element sequence containing floats in range of [0,100], optional] Percentiles over which to compute the range. Each must be between 0 and 100, inclusive. The order of the elements is not important.

Returns

———-

array limited array

momepy.nx_to_gdf

momepy.**nx_to_gdf**(*net*, *points=True*, *lines=True*, *spatial_weights=False*, *nodeID='nodeID'*)
Convert `networkx.Graph` to LineString GeoDataFrame and Point GeoDataFrame

Parameters

net [networkx.Graph] `networkx.Graph`

points [bool] export point-based gdf representing intersections

lines [bool] export line-based gdf representing streets

spatial_weights [bool] export libpysal spatial weights for nodes

nodeID [str] name of node ID column to be generated

Returns

GeoDataFrame Selected gdf or tuple of both gdf or tuple of gdfs and weights

momepy.unique_id

momepy.**unique_id**(*objects*)
Add an attribute with unique ID to each row of GeoDataFrame.

Parameters

objects [GeoDataFrame] GeoDataFrame containing objects to analyse

Returns

Series Series containing resulting values.

8.4 Contributing to momepy

(Contribution guidelines are largely based on geopandas.)

Contributions of any kind to momepy are more than welcome. That does not mean new code only, but also improvements of documentation and user guide, additional tests (ideally filling the gaps in existing suite) or bug report or idea what could be added or done better.

All contributions should go through our GitHub repository. Bug reports, ideas or even questions should be raised by opening an issue on the GitHub tracker. Suggestions for changes in code or documentation should be submitted as a pull request. However, if you are not sure what to do, feel free to open an issue. All discussion will then take place on GitHub to keep the development of momepy transparent.

If you decide to contribute to the codebase, ensure that you are using an up-to-date *master* branch. The latest development version will always be there, including the documentation (powered by sphinx).

8.4.1 Eight Steps for Contributing

There are seven basic steps to contributing to momepy:

1. Fork the momepy git repository
2. Create a development environment
3. Install momepy dependencies
4. Make a development build of momepy
5. Make changes to code and add tests
6. Update the documentation
7. Format code
8. Submit a Pull Request

Each of the steps is detailed below.

1. Fork the momepy git repository

Git can be complicated for new users, but you no longer need to use command line to work with git. If you are not familiar with git, we recommend using tools on GitHub.org, GitHub Desktop or tools with included git like Atom. However, if you want to use command line, you can fork momepy repository using following:

```
git clone git@github.com:your-user-name/momepy.git momepy-yourname
cd momepy-yourname
git remote add upstream git://github.com/martinfleis/momepy.git
```

This creates the directory momepy-yourname and connects your repository to the upstream (main project) momepy repository.

Then simply create a new branch of master branch.

2. Create a development environment

A development environment is a virtual space where you can keep an independent installation of momepy. This makes it easy to keep both a stable version of python in one place you use for work, and a development version (which you may break while playing with code) in another.

An easy way to create a momepy development environment is as follows:

- Install either Anaconda or miniconda
- Make sure that you have cloned the repository
- `cd` to the *momepy* source directory

Tell conda to create a new environment, named `momepy_dev`, or any other name you would like for this environment, by running:

```
conda create -n momepy_dev
```

This will create the new environment, and not touch any of your existing environments, nor any existing python installation.

To work in this environment, Windows users should `activate` it as follows:

```
activate momepy_dev
```

macOS and Linux users should use:

```
conda activate momepy_dev
```

You will then see a confirmation message to indicate you are in the new development environment.

To view your environments:

```
conda info -e
```

To return to you home root environment:

```
deactivate
```

See the full conda docs here.

At this point you can easily do a *development* install, as detailed in the next sections.

3. Installing Dependencies

To run *momepy* in an development environment, you must first install *momepy*'s dependencies. We suggest doing so using the following commands (executed after your development environment has been activated) to ensure compatibility of all dependencies:

```
conda config --env --add channels conda-forge
conda config --env --set channel_priority strict
conda install geopandas networkx libpysal tqdm pysal mapclassify pytest
```

This should install all necessary dependencies including optional.

4. Making a development build

Once dependencies are in place, make an in-place build by navigating to the git clone of the *momepy* repository and running:

```
python setup.py develop
```

This will install momepy into your environment but allows any further changes without the need of reinstalling new version.

5. Making changes and writing tests

momepy is serious about testing and strongly encourages contributors to embrace test-driven development (TDD). This development process "relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test." So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

momepy uses the pytest testing system.

Writing tests

All tests should go into the `tests` directory. This folder contains many current examples of tests, and we suggest looking to these for inspiration.

Running the test suite

The tests can then be run directly inside your Git clone (without having to install *momepy*) by typing:

```
pytest
```

6. Updating the Documentation and User Guide

momepy documentation resides in the *docs* folder. Changes to the docs are make by modifying the appropriate file within *doc*. *momepy* docs us reStructuredText syntax, which is explained here and the docstrings follow the Numpy Docstring standard.

Once you have made your changes, you may try if they render correctly by building the docs using sphinx. To do so, you can navigate to the doc folder and type:

```
make html
```

The resulting html pages will be located in doc/build/html. In case of any errors, you can try to use make html within a new environment based on environment.yml specification in the doc folder. Using conda:

```
conda env create -f environment.yml
conda activate geopandas_docs
make html
```

For minor updates, you can skip whole make html part as reStructuredText syntax is usually quite straightforward.

Updating User Guide

Updating user guide might be slightly more complicated as it consists of collection of reStructuredText files and Jupyter notebooks. Changes in reStructuredText are straightforward, changes in notebooks should be done using Jupyter. Make sure that all cells have their correct outputs as notebooks are not executed by readthedocs.

7. Formatting the code

Python (PEP8 / black)

momepy follows the PEP8 standard and uses Black to ensure a consistent code format throughout the project.

CI will run `black --check` and fails if there are files which would be auto-formatted by `black`. Therefore, it is helpful before submitting code to auto-format your code:

```
black momepy
```

Additionally, many editors have plugins that will apply `black` as you edit files. If you don't have black, you can install it using pip:

```
pip install black
```

8. Submitting a Pull Request

Once you've made changes and pushed them to your forked repository, you then submit a pull request to have them integrated into the *momepy* code base.

You can find a pull request (or PR) tutorial in the GitHub's Help Docs.

8.5 Changelog

8.5.1 Version 0.4.0 (TBD)

Requirements:

- momepy now requires GeoPandas 0.8 or newer
- momepy now requires pygeos

API changes:

- `network_false_nodes` is now deprecated. Use new `remove_false_nodes` instead.

Enhancements:

- New performant algorithm `remove_false_nodes` to remove nodes of a degree 2 of a LineString network.

8.5.2 Version 0.3.0 (July 29, 2020)

API changes:

- `Convexeity` is now `Convexity` (#171)
- `local_` centrality (betweenness, closeness, straightness) has been included in respective global versions (#178)

New features:

- `CheckTessellationInput` to check building footprint data for potential issues during `Tessellation` (#163)
- On demand `DistanceBand` spatial weights (neighbors) for larger weight which would not fit in memory (#165)

Enhancements:

- New documentation (#167)
- Support network analysis for full network (#176)
- Options for `preprocess` (#180)
- Expose underlying `simpson` and `shannon` functions (#183)
- `MeanInterbuildingDistance` performance (#187)
- `StreetProfile` performance refactor (#186)
- Retain attributes in `network_false_nodes` (#189)
- Performance improvements in `elements` module (#190)
- Performance refactor of `SharedWallsRatio` (#191)
- Performance refactor of `NeighboringStreetOrientationDeviation` (#192)
- Minor performance improvements (#193, #196)
- Allow specification of verbosity (#195)
- Perfomance enhancements in `sw_high` (#198)

Bug fixes:

- `Density` TypeError for islands (#164)
- Preserve CRS in `network_false_nodes` (#181)
- Fixed `Squareness` for non-Polygon geom types (#182)
- CRS lost with older geopandas (#188)

8.5.3 Version 0.2.1 (April 15, 2020)

- fixed regression causing `MeanInterbuildingDistance` failure (#161)

8.5.4 Version 0.2.0 (April 13, 2020)

API changes:

- `AverageCharacter` allows calculation of multiple modes (mean, median, mode) at the same time. Each can be accessed via its own attribute. Apart from `mean`, none is accessible using `.series` anymore. (#147)

Enhancements:

- `Shannon` index (#158)
- `Simpson` allows Gini-Simpson and Inverse Simpson index modes (#157)
- Diversity classes support categorical values (#159)
- `SegmentsLength` allows sum and mean at the same time (#146)
- `AverageCharacter` allows calculation of multiple modes (mean, median, mode) at the same time (#147)
- Better compatibility with OSMnx Graphs (#149)
- `Orientation` support LineString geometry (#156)
- `AreaRatio` for uneven number of features (#135)
- Performance improvements (#144, #145, #152, #155)

Bug fixes:

- float precision errors in `network_false_nodes` (#133)
- `network_false_nodes` for multiindex (#136)
- `BlocksCount` no neighbors error (#139, #140)
- LineString Z support in `nx_to_gdf` (#148)
- accidental 'rtd' print (#150)
- `CentroidCorner` may fail for Polygon Z (#151)

8.5.5 Version 0.1.1 (December 15, 2019)

Small bug-fix release:

- fix for `AreaRatio` resetting index of resulting Series (#127)
- fix for `network_false_nodes` to work with GeoSeries (#128)
- fix for incomplete spatial_weights and missing neighbors. Instead of raising KeyError momepy now returns np.nan for affected row. np.nan is also returned if there are no neighbors (instead of 0). (#131)

8.6 References

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[AF19] Alessandro Araldi and Giovanni Fusco. From the street to the metropolitan region: Pedestrian perspective in urban fabric analysis:. *Environment and Planning B: Urban Analytics and City Science*, 46(7):1243–1263, August 2019. doi:10.1177/2399808319832612.

[BC17] Melih Basaraner and Sinan Cetinkaya. Performance of shape indices and classification schemes for characterising perceptual shape complexity of building footprints in GIS. *International Journal of Geographical Information Science*, 31(10):1952–1977, July 2017. doi:10.1080/13658816.2017.1346257.

[BSN12] Loeiz Bourdic, Serge Salat, and Caroline Nowacki. Assessing cities: a new system of cross-scale spatial indicators. *Building Research & Information*, 40(5):592–605, January 2012. doi:10.1080/09613218.2012.703488.

[Cim17] Zofie Cimburova. *Urban Morphology in Prague: Automatic Classification in GIS*. PhD thesis, Czech Technical University, Prague, January 2017.

[DPR+17] Jacob Dibble, Alexios Prelorendjos, Ombretta Romice, Mattia Zanella, Emanuele Strano, Mark Pagel, and Sergio Porta. On the origin of spaces: Morphometric foundations of urban form evolution. *Environment and Planning B: Urban Analytics and City Science*, 46(4):707–730, August 2017. doi:10.1177/2399808317725075.

[Fel18] Alessandra Feliciotti. *RESILIENCE AND URBAN DESIGN: A SYSTEMS APPROACH TO THE STUDY OF RESILIENCE IN URBAN FORM*. PhD thesis, University of Strathclyde, Glasgow, January 2018.

[FFRP20] Martin Fleischmann, Alessandra Feliciotti, Ombretta Romice, and Sergio Porta. Morphological tessellation as a way of partitioning space: Improving consistency in urban morphology at the plot scale. *Computers, Environment and Urban Systems*, 80:101441, January 2020. doi:10.1016/j.compenvurbsys.2019.101441.

[GMBeiraoD12] Jorge Gil, Nuno Montenegro, J N Beirão, and J P Duarte. On the Discovery of Urban Typologies: Data Mining the Multi-dimensional Character of Neighbourhoods. *Urban Morphology*, 16(1):27–40, January 2012.

[HLM12] Rachid Hamaina, Thomas Leduc, and Guillaume Moreau. Towards Urban Fabrics Characterization Based on Buildings Footprints. In *Bridging the Geographic Information Sciences*, volume 2, pages 327–346. Springer, Berlin, Heidelberg, Berlin, Heidelberg, January 2012. doi:10.1007/978-3-642-29063-3_18.

[HBP17] Birgit Hausleitner and Meta Berghauser Pont. Development of a configurational typology for micro-businesses integrating geometric and configurational variables. In *11th Space Syntax Symposium*. January 2017.

[HRRCambraLopez12] T Hermosilla, L A Ruiz, J A Recio, and M Cambra-López. Assessing contextual descriptive features for plot-based classification of urban areas. *Landscape and Urban Planning*, 106(1):124–137, May 2012. doi:10.1016/j.landurbplan.2012.02.008.

[McG95] Kevin McGarigal. *FRAGSTATS: Spatial Pattern Analysis Program for Quantifying Landscape Structure*. Volume 351. US Department of Agriculture, Forest Service, Pacific Northwest Research Station, 1995.

[PCL06] Sergio Porta, Paolo Crucitti, and Vito Latora. The network analysis of urban streets: A primal approach. *Environment and Planning B: Planning and Design*, 33(5):705–725, January 2006. doi:10.1068/b32045.

[SA15] Patrick Michael Schirmer and Kay W Axhausen. A multiscale classification of urban morphology. *Journal of Transport and Land Use*, 9(1):101–130, May 2015. doi:10.5198/jtlu.2015.667.

[VC17] Sven Vanderhaegen and Frank Canters. Mapping urban form and function at city block level using spatial metrics. *Landscape and Urban Planning*, 167:399–409, November 2017. doi:10.1016/j.landurbplan.2017.05.023.

PYTHON MODULE INDEX

m

momepy, 169

INDEX

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

R

S

T

U

V

W