Mesa-Geo Documentation

Release 0.7.0

Project Mesa-Geo Team

Jan 17, 2024

CONTENTS

Mesa-Geo implements a `GeoSpace` that can host GIS-based `GeoAgents`, which are like normal Agents, except they have a `geometry` attribute that is a Shapely object and a `crs` attribute for its Coordinate Reference System. You can use `Shapely` directly to create arbitrary geometries, but in most cases you will want to import your geometries from a file. Mesa-Geo allows you to create GeoAgents from any vector data file (e.g. shapefiles), valid GeoJSON objects or a GeoPandas GeoDataFrame.

CHAPTER

ONE

USING MESA-GEO

To install Mesa-Geo on linux or macOS run

```
pip install mesa-geo
```

On windows you should first use Anaconda to install some of the requirements with

```
conda install fiona pyproj rtree shapely
pip install mesa-geo
```

Since Mesa-Geo is in early development you could also install the latest version directly from Github via

```
pip install -e git+https://github.com/projectmesa/mesa-geo.git#egg=mesa-geo
```

Take a look at the examples folder for sample models demonstrating Mesa-Geo features.

For more help on using Mesa-Geo, check out the following resources:

- Introductory Tutorial
- Docs
- Mesa-Geo Discussions
- PyPI

CHAPTER

TWO

CONTRIBUTING TO MESA-GEO

Want to join the team or just curious about what is happening with Mesa & Mesa-Geo? You can…

- Join our Matrix chat room in which questions, issues, and ideas can be (informally) discussed.
- Come to a monthly dev session (you can find dev session times, agendas and notes at Mesa discussions.
- Just check out the code at GitHub.

If you run into an issue, please file a ticket for us to discuss. If possible, follow up with a pull request.

If you would like to add a feature, please reach out via ticket or join a dev session (see Mesa discussions). A feature is most likely to be added if you build it!

Don't forget to check out the Contributors guide.

CHAPTER

THREE

CITING MESA-GEO

To cite Mesa-Geo in your publication, you can use the CITATION.bib.

3.1 Introductory Tutorial

3.1.1 Getting started

You should be familiar with how Mesa works.

So let's get started with some geometries! We will work with records of US states. We use the `requests` library to retrieve the data, but of course you can work with local data.

```
import warnings

warnings.filterwarnings("ignore")

import mesa
import mesa_geo as mg
import requests


url = "http://eric.clst.org/assets/wiki/uploads/Stuff/gz_2010_us_040_00_20m.json"
r = requests.get(url)
geojson_states = r.json()
```

First we create a `State` Agent and a `GeoModel`. Both should look familiar if you have worked with Mesa before.

```
class State(mg.GeoAgent):
    def __init__(self, unique_id, model, geometry, crs):
        super().__init__(unique_id, model, geometry, crs)


class GeoModel(mesa.Model):
    def __init__(self):
        self.space = mg.GeoSpace()

        ac = mg.AgentCreator(agent_class=State, model=self)
        agents = ac.from_GeoJSON(GeoJSON=geojson_states, unique_id="NAME")
        self.space.add_agents(agents)
```

In the `GeoModel` we first create an instance of AgentCreator, where we provide the Agent class (State) and its required arguments, except geometry and unique_id. We then use the `.from_GeoJSON` function to create our agents from the geometries in the GeoJSON file. We provide the feature "name" as the key from which the agents get their unique_ids. Finally, we add the agents to the GeoSpace

Let's instantiate our model and look at one of the agents:

```
m = GeoModel()

agent = m.space.agents[0]
print(agent.unique_id)
agent.geometry
```

```
Arizona
```

```
<POLYGON ((-12527738.867 4439200.735, -12527288.246 4439202.128, -12508853.8...>
```

If you work in the Jupyter Notebook your output should give you the name of the state and a visual representation of the geometry.

By default the AgentCreator also sets further agent attributes from the Feature properties.

```
agent.CENSUSAREA
```

```
113594.084
```

Let's start to do some spatial analysis. We can use usual Mesa function names to get neighboring states.

```
neighbors = m.space.get_neighbors(agent)
print([a.unique_id for a in neighbors])
```

```
['California', 'Colorado', 'Utah', 'Nevada', 'New Mexico']
```

To get a list of all states within a certain distance you can use the following:

```
[a.unique_id for a in m.space.get_neighbors_within_distance(agent, 600000)]
```

```
['California',
 'Nevada',
 'Arizona',
 'Utah',
 'Wyoming',
 'New Mexico',
 'Colorado',
 'Texas']
```

The unit for the distance depends on the coordinate reference system (CRS) of the GeoSpace. Since we did not specify the CRS, Mesa-Geo defaults to the 'Web Mercator' projection (in meters). If you want to do some serious measurements you should always set an appropriate CRS, since the accuracy of Web Mercator declines with distance from the equator. We can achieve this by initializing the AgentCreator and the GeoSpace with the `crs` keyword `crs="epsg:2163"`. Mesa-Geo then transforms all coordinates from the GeoJSON geographic coordinates into the set crs.

3.1.2 Going further

To get a deeper understanding of Mesa-Geo you should check out the GeoSchelling example. It implements a Leaflet visualization which is similar to use as the CanvasGridVisualization of Mesa.

To add further functionality, I need feedback on which functionality is desired by users. Please post a message at Mesa-Geo discussions or open an issue if you have any ideas or recommendations.

3.2 Examples

Vector Data

- *GeoSchelling Model (Polygons)*
- *GeoSchelling Model (Points & Polygons)*
- *GeoSIR Epidemics Model*

Raster Data

- *Rainfall Model*
- *Urban Growth Model*

Raster and Vector Data Overlay

- *Population Model*

3.2.1 GeoSchelling Model (Polygons)

Summary

This is a geoversion of a simplified Schelling example. For the original implementation details please see the Mesa Schelling examples.

GeoSpace

Instead of an abstract grid space, we represent the space using NUTS-2 regions to create the GeoSpace in the model.

GeoAgent

NUTS-2 regions are the GeoAgents. The neighbors of a polygon are considered those polygons that touch its border (i.e., edge neighbours). During the running of the model, a polygon queries the colors of the surrounding polygon and if the ratio falls below a certain threshold (e.g., 40% of the same color), the agent moves to an uncolored polygon.

How to Run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.2.2 GeoSchelling Model (Points & Polygons)

Summary

This is a geoversion of a simplified Schelling example.

GeoSpace

The NUTS-2 regions are considered as a shared definition of neighborhood among all people agents, instead of a locally defined neighborhood such as Moore or von Neumann.

GeoAgent

There are two types of GeoAgents: people and regions. Each person resides in a randomly assigned region, and checks the color ratio of its region against a pre-defined "happiness" threshold at every time step. If the ratio falls below a certain threshold (e.g., 40%), the agent is found to be "unhappy", and randomly moves to another region. People are represented as points, with locations randomly chosen within their regions. The color of a region depends on the color of the majority population it contains (i.e., point in polygon calculations).

How to run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.2.3 GeoSIR Epidemics Model

Summary

This is a geoversion of a simple agent-based pandemic SIR model, as an example to show the capabilities of mesa-geo.

It uses geographical data of Toronto's regions on top of a an Leaflet map to show the location of agents (in a continuous space).

Person agents are initially located in random positions in the city, then start moving around unless they die. A fraction of agents start with an infection and may recover or die in each step. Susceptible agents (those who have never been infected) who come in proximity with an infected agent may become infected.

Neighbourhood agents represent neighbourhoods in the Toronto, and become hot-spots (colored red) if there are infected agents inside them. Data obtained from this link.

How to run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.2.4 Rainfall Model

Summary

This is an implementation of the Rainfall Model in Python, using Mesa and Mesa-Geo. Inspired by the NetLogo Grand Canyon model, this is an example of how a digital elevation model (DEM) can be used to create an artificial world.

GeoSpace

The GeoSpace contains a raster layer representing elevations. It is this elevation value that impacts how the raindrops move over the terrain. Apart from `elevation`, each cell of the raster layer also has a `water_level` attribute that is used to track the amount of water it contains.

GeoAgent

In this example, the raindrops are the GeoAgents. At each time step, raindrops are randomly created across the landscape to simulate rainfall. The raindrops flow from cells of higher elevation to lower elevation based on their eight surrounding cells (i.e., Moore neighbourhood). The raindrop also has its own height, which allows them to accumulate, gain height and flow if they are trapped at places such as potholes, pools, or depressions. When they reach the boundary of the GeoSpace, they are removed from the model as outflow.

How to run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.2.5 Urban Growth Model

Summary

This is an implementation of the UrbanGrowth Model in Python, using Mesa and Mesa-Geo.

How to run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.2.6 Population Model

Summary

This is an implementation of the Uganda Example in Python, using Mesa and Mesa-Geo.

GeoSpace

The GeoSpace consists of both a raster and a vector layer. The raster layer contains population data for each cell, and it is this data that is used for model initialisation, in the sense creating the agents. The vector layer shown in blue color represents a lake in Uganda. It overlays with the raster layer to mask out the cells that agents cannot move into.

GeoAgent

The GeoAgents are people, created based on the population data. As this is a simple example model, the agents only move randomly to neighboring cells at each time step. To make the simulation more realistic and visually appealing, the agents in the same cell have a randomized position within the cell, so that they don't stand on top of each other at exactly the same coordinate.

How to run

To run the model interactively, run `mesa runserver` in this directory. e.g.

```
mesa runserver
```

Then open your browser to http://127.0.0.1:8521/ and press `Start`.

3.3 APIs

3.3.1 GeoBase

class GeoBase(*crs=None*)

Base class for all geo-related classes.

Create a new GeoBase object.

Parameters

crs – The coordinate reference system of the object.

abstract property total_bounds: np.ndarray | None

Return the bounds of the object in [min_x, min_y, max_x, max_y] format.

Returns
The bounds of the object in [min_x, min_y, max_x, max_y] format.

Return type
np.ndarray | None

property crs: pyproj.CRS | None

Return the coordinate reference system of the object.

abstract to_crs(*crs*, *inplace=False*) → *GeoBase* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

3.3.2 GeoAgent and AgentCreator classes

class GeoAgent(*unique_id*, *model*, *geometry*, *crs*)

Base class for a geo model agent.

Create a new agent.

Parameters

- **unique_id** – Unique ID for the agent.
- **model** – The model the agent is in.
- **geometry** – A Shapely object representing the geometry of the agent.
- **crs** – The coordinate reference system of the geometry.

property total_bounds: np.ndarray | None

Return the bounds of the object in [min_x, min_y, max_x, max_y] format.

Returns
The bounds of the object in [min_x, min_y, max_x, max_y] format.

Return type
np.ndarray | None

to_crs(*crs*, *inplace=False*) → *GeoAgent* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

`get_transformed_geometry`(*transformer*)

Return the transformed geometry given a transformer.

`step`()

Advance one step.

`advance`() → None

`property crs: pyproj.CRS | None`

Return the coordinate reference system of the object.

`property random: Random`

`class AgentCreator`(*agent_class*, *model=None*, *crs=None*, *agent_kwargs=None*)

Create GeoAgents from files, GeoDataFrames, GeoJSON or Shapely objects.

Define the agent_class and required agent_kwargs.

Parameters

- **agent_class** – The class of the agent to create.
- **model** – The model to create the agent in.
- **crs** – The coordinate reference system of the agent. Default to None, and the crs from the file/GeoDataFrame/GeoJSON will be used. Otherwise, geometries are converted into this crs automatically.
- **agent_kwargs** – Keyword arguments to pass to the agent_class. Must NOT include unique_id.

`property crs`

Return the coordinate reference system of the GeoAgents.

`create_agent`(*geometry*, *unique_id*)

Create a single agent from a geometry and a unique_id. Shape must be a valid Shapely object.

Parameters

- **geometry** – The geometry of the agent.
- **unique_id** – The unique_id of the agent.

Returns
The created agent.

Return type
self.agent_class

`from_GeoDataFrame`(*gdf*, *unique_id='index'*, *set_attributes=True*)

Create a list of agents from a GeoDataFrame.

Parameters

- **gdf** – The GeoDataFrame to create agents from.
- **unique_id** – The column name of the data to use as the agents unique_id. If "index", the index of the GeoDataFrame is used. Default to "index".

- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

from_file(*filename*, *unique_id='index'*, *set_attributes=True*)

Create agents from vector data files (e.g. Shapefiles).

Parameters

- **filename** – The vector data file to create agents from.
- **unique_id** – The column name of the data to use as the agents unique_id. If "index", the index of the GeoDataFrame is used. Default to "index".
- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

from_GeoJSON(*GeoJSON*, *unique_id='index'*, *set_attributes=True*)

Create agents from a GeoJSON object or string. CRS is set to epsg:4326.

Parameters

- **GeoJSON** – The GeoJSON object or string to create agents from.
- **unique_id** – The column name of the data to use as the agents unique_id. If "index", the index of the GeoDataFrame is used. Default to "index".
- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

3.3.3 GeoSpace

class GeoSpace(*crs='epsg:3857'*, ***, *warn_crs_conversion=True*)

Space used to add a geospatial component to a model.

Create a GeoSpace for GIS enabled mesa modeling.

Parameters

- **crs** – The coordinate reference system of the GeoSpace. If *crs* is not set, epsg:3857 (Web Mercator) is used as default. However, this system is only accurate at the equator and errors increase with latitude.
- **warn_crs_conversion** – Whether to warn when converting layers and GeoAgents of different crs into the crs of GeoSpace. Default to True.

to_crs(*crs*, *inplace=False*) → *GeoSpace* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

property transformer

Return the pyproj.Transformer that transforms the GeoSpace into epsg:4326. Mainly used for GeoJSON serialization.

property agents

Return a list of all agents in the Geospace.

property layers: list[*ImageLayer* | *RasterLayer* | gpd.GeoDataFrame]

Return a list of all layers in the Geospace.

property total_bounds: np.ndarray | None

Return the bounds of the GeoSpace in [min_x, min_y, max_x, max_y] format.

add_layer(*layer:* ImageLayer | RasterLayer | *gpd.GeoDataFrame*) → None

Add a layer to the Geospace.

Parameters
layer (ImageLayer | RasterLayer | *gpd.GeoDataFrame*) – The layer to add.

add_agents(*agents*)

Add a list of GeoAgents to the Geospace.

GeoAgents must have a geometry attribute. This function may also be called with a single GeoAgent.

Parameters
agents – A list of GeoAgents or a single GeoAgent to be added into GeoSpace.

Raises
AttributeError – If the GeoAgents do not have a geometry attribute.

remove_agent(*agent*)

Remove an agent from the GeoSpace.

get_relation(*agent*, *relation*)

Return a list of related agents.

Parameters

- **agent** (GeoAgent) – The agent to find related agents for.
- **relation** (str) – The relation to find. Must be one of 'intersects', 'within', 'contains', 'touches'.

get_intersecting_agents(*agent*)

get_neighbors_within_distance(*agent*, *distance*, *center=False*, *relation='intersects'*)

Return a list of agents within *distance* of *agent*.

Distance is measured as a buffer around the agent's geometry, set center=True to calculate distance from center.

agents_at(*pos*)

Return a list of agents at given pos.

distance(*agent_a*, *agent_b*)

Return distance of two agents.

get_neighbors(*agent*)

Get (touching) neighbors of an agent.

get_agents_as_GeoDataFrame(*agent_cls=<class 'mesa_geo.geoagent.GeoAgent'>*) → GeoDataFrame

Extract GeoAgents as a GeoDataFrame.

Parameters
agent_cls – The class of the GeoAgents to extract. Default is *GeoAgent*.

Returns
A GeoDataFrame of the GeoAgents.

Return type
gpd.GeoDataFrame

property crs: pyproj.CRS | None
Return the coordinate reference system of the object.

3.3.4 Raster Layers

class RasterBase(*width*, *height*, *crs*, *total_bounds*)
Base class for raster layers.

Initialize a raster base layer.

Parameters

- **width** – Width of the raster base layer.
- **height** – Height of the raster base layer.
- **crs** – Coordinate reference system of the raster base layer.
- **total_bounds** – Bounds of the raster base layer in [min_x, min_y, max_x, max_y] format.

property width: int
Return the width of the raster base layer.

Returns
Width of the raster base layer.

Return type
int

property height: int
Return the height of the raster base layer.

Returns
Height of the raster base layer.

Return type
int

property total_bounds: np.ndarray | None
Return the bounds of the object in [min_x, min_y, max_x, max_y] format.

Returns
The bounds of the object in [min_x, min_y, max_x, max_y] format.

Return type
np.ndarray | None

property transform: Affine
Return the affine transformation of the raster base layer.

Returns
Affine transformation of the raster base layer.

Return type
Affine

property resolution: tuple[float, float]

Returns the (width, height) of a cell in the units of CRS.

Returns
Width and height of a cell in the units of CRS.

Return type
Tuple[float, float]

to_crs(*crs*, *inplace=False*) → *RasterBase* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

out_of_bounds(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid.

Parameters
pos (*Coordinate*) – Position to check.

Returns
True if position is off the grid, False otherwise.

Return type
bool

property crs: pyproj.CRS | None

Return the coordinate reference system of the object.

class Cell(*pos=None*, *indices=None*)

Cells are containers of raster attributes, and are building blocks of *RasterLayer*.

Initialize a cell.

Parameters

- **pos** – Position of the cell in (x, y) format. Origin is at lower left corner of the grid
- **indices** – Indices of the cell in (row, col) format. Origin is at upper left corner of the grid

pos: Coordinate | None

indices: Coordinate | None

step()

A single step of the agent.

advance() → None

property random: Random

class RasterLayer(*width*, *height*, *crs*, *total_bounds*, *cell_cls: type[~mesa_geo.raster_layers.Cell] = <class 'mesa_geo.raster_layers.Cell'>*)

Some methods in *RasterLayer* are copied from *mesa.space.Grid*, including:

__getitem__ __iter__ coord_iter iter_neighborhood get_neighborhood iter_neighbors get_neighbors # copied and renamed to *get_neighboring_cells* out_of_bounds # copied into *RasterBase* iter_cell_list_contents get_cell_list_contents

Methods from *mesa.space.Grid* that are not copied over:

torus_adj neighbor_iter move_agent place_agent _place_agent remove_agent is_cell_empty move_to_empty find_empty exists_empty_cells

Another difference is that *mesa.space.Grid* has *self.grid: List[List[Agent | None]]*, whereas it is *self.cells: List[List[Cell]]* here in *RasterLayer*.

Initialize a raster base layer.

Parameters

- **width** – Width of the raster base layer.
- **height** – Height of the raster base layer.
- **crs** – Coordinate reference system of the raster base layer.
- **total_bounds** – Bounds of the raster base layer in [min_x, min_y, max_x, max_y] format.

cells: list[list[*Cell*]]

property attributes: set[str]

Return the attributes of the cells in the raster layer.

Returns
Attributes of the cells in the raster layer.

Return type
Set[str]

coord_iter() → Iterator[tuple[*Cell*, int, int]]

An iterator that returns coordinates as well as cell contents.

apply_raster(*data: np.ndarray*, *attr_name: str | None = None*) → None

Apply raster data to the cells.

Parameters

- **data** (*np.ndarray*) – 2D numpy array with shape (1, height, width).
- **attr_name** (*str | None*) – Name of the attribute to be added to the cells. If None, a random name will be generated. Default is None.

Raises
ValueError – If the shape of the data is not (1, height, width).

get_raster(*attr_name: str | None = None*) → np.ndarray

Return the values of given attribute.

Parameters
attr_name (*str | None*) – Name of the attribute to be returned. If None, returns all attributes. Default is None.

Returns
The values of given attribute as a 2D numpy array with shape (1, height, width).

Return type
np.ndarray

iter_neighborhood(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[Tuple[int, int]]

Return an iterator over cell coordinates that are in the neighborhood of a certain point.

Parameters

- **pos** (*Coordinate*) – Coordinate tuple for the neighborhood to get.
- **moore** (*bool*) – Whether to use Moore neighborhood or not. If True, return Moore neighborhood (including diagonals). If False, return Von Neumann neighborhood (exclude diagonals).
- **include_center** (*bool*) – If True, return the (x, y) cell as well. Otherwise, return surrounding cells only. Default is False.
- **radius** (*int*) – Radius, in cells, of the neighborhood. Default is 1.

Returns
An iterator over cell coordinates that are in the neighborhood. For example with radius 1, it will return list with number of elements equals at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

Return type
Iterator[Coordinate]

iter_neighbors(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → Iterator[*Cell*]

Return an iterator over neighbors to a certain point.

Parameters

- **pos** (*Coordinate*) – Coordinate tuple for the neighborhood to get.
- **moore** (*bool*) – Whether to use Moore neighborhood or not. If True, return Moore neighborhood (including diagonals). If False, return Von Neumann neighborhood (exclude diagonals).
- **include_center** (*bool*) – If True, return the (x, y) cell as well. Otherwise, return surrounding cells only. Default is False.
- **radius** (*int*) – Radius, in cells, of the neighborhood. Default is 1.

Returns
An iterator of cells that are in the neighborhood; at most 9 (8) if Moore, 5 (4) if Von Neumann (if not including the center).

Return type
Iterator[*Cell*]

iter_cell_list_contents(*positions*) → Any

property crs: pyproj.CRS | None

Return the coordinate reference system of the object.

get_cell_list_contents(*positions*) → Any

property height: int

Return the height of the raster base layer.

Returns
Height of the raster base layer.

Return type
int

`out_of_bounds`(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid.

Parameters
pos (*Coordinate*) – Position to check.

Returns
True if position is off the grid, False otherwise.

Return type
bool

`property resolution: tuple[float, float]`

Returns the (width, height) of a cell in the units of CRS.

Returns
Width and height of a cell in the units of CRS.

Return type
Tuple[float, float]

`property total_bounds: np.ndarray | None`

Return the bounds of the object in [min_x, min_y, max_x, max_y] format.

Returns
The bounds of the object in [min_x, min_y, max_x, max_y] format.

Return type
np.ndarray | None

`property transform: Affine`

Return the affine transformation of the raster base layer.

Returns
Affine transformation of the raster base layer.

Return type
Affine

`property width: int`

Return the width of the raster base layer.

Returns
Width of the raster base layer.

Return type
int

`get_neighborhood`(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[Tuple[int, int]]

`get_neighboring_cells`(*pos: Tuple[int, int]*, *moore: bool*, *include_center: bool = False*, *radius: int = 1*) → list[*Cell*]

`to_crs`(*crs*, *inplace=False*) → *RasterLayer* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

`to_image`(*colormap*) → *ImageLayer*

Returns an ImageLayer colored by the provided colormap.

`classmethod from_file`(*raster_file: str*, *cell_cls: type[Cell] = <class 'mesa_geo.raster_layers.Cell'>*, *attr_name: str | None = None*) → *RasterLayer*

Creates a RasterLayer from a raster file.

Parameters

- **raster_file** (*str*) – Path to the raster file.
- **cell_cls** (*Type[Cell]*) – The class of the cells in the layer.
- **attr_name** (*str | None*) – The name of the attribute to use for the cell values. If None, a random name will be generated. Default is None.

`to_file`(*raster_file: str*, *attr_name: str | None = None*, *driver: str = 'GTiff'*) → None

Writes a raster layer to a file.

Parameters

- **raster_file** (*str*) – The path to the raster file to write to.
- **attr_name** (*str | None*) – The name of the attribute to write to the raster. If None, all attributes are written. Default is None.
- **driver** (*str*) – The GDAL driver to use for writing the raster file. Default is 'GTiff'. See GDAL docs at https://gdal.org/drivers/raster/index.html.

`class ImageLayer`(*values*, *crs*, *total_bounds*)

Initializes an ImageLayer.

Parameters

- **values** – The values of the image layer.
- **crs** – The coordinate reference system of the image layer.
- **total_bounds** – The bounds of the image layer in [min_x, min_y, max_x, max_y] format.

`property crs: pyproj.CRS | None`

Return the coordinate reference system of the object.

`property height: int`

Return the height of the raster base layer.

Returns
Height of the raster base layer.

Return type
int

`out_of_bounds`(*pos: Tuple[int, int]*) → bool

Determines whether position is off the grid.

Parameters
pos (*Coordinate*) – Position to check.

Returns
True if position is off the grid, False otherwise.

Return type
bool

`property resolution: tuple[float, float]`

Returns the (width, height) of a cell in the units of CRS.

Returns
Width and height of a cell in the units of CRS.

Return type
Tuple[float, float]

`property total_bounds: np.ndarray | None`

Return the bounds of the object in [min_x, min_y, max_x, max_y] format.

Returns
The bounds of the object in [min_x, min_y, max_x, max_y] format.

Return type
np.ndarray | None

`property transform: Affine`

Return the affine transformation of the raster base layer.

Returns
Affine transformation of the raster base layer.

Return type
Affine

`property width: int`

Return the width of the raster base layer.

Returns
Width of the raster base layer.

Return type
int

`property values: ndarray`

Returns the values of the image layer.

Returns
The values of the image layer.

Return type
np.ndarray

`to_crs`(*crs*, *inplace=False*) → *ImageLayer* | None

Transform the object to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.
- **inplace** – Whether to transform the object in place or return a new object. Defaults to False.

Returns
The transformed object if not inplace.

Return type
GeoBase | None

`classmethod from_file`(*image_file*) → *ImageLayer*

Creates an ImageLayer from an image file.

Parameters
image_file – The path to the image file.

Returns
The ImageLayer.

Return type
ImageLayer

3.3.5 Visualization

Modules

Container for all built-in visualization modules.

`class LeafletPortrayal`(*style: dict[str, LeafletOption] | None = None*, *pointToLayer: dict[str, LeafletOption] | None = None*, *popupProperties: dict[str, LeafletOption] | None = None*)

A dataclass defining the portrayal of a GeoAgent in Leaflet map.

The fields are defined to be consistent with GeoJSON options in Leaflet.js: https://leafletjs.com/reference.html#geojson

`style: dict[str, LeafletOption] | None = None`

`pointToLayer: dict[str, LeafletOption] | None = None`

`popupProperties: dict[str, LeafletOption] | None = None`

`class MapModule`(*portrayal_method=None*, *view=None*, *zoom=None*, *map_width=500*, *map_height=500*, *tiles={'attribution': '(C) OpenStreetMap contributors', 'html_attribution': '© OpenStreetMap contributors', 'max_zoom': 19, 'name': 'OpenStreetMap.Mapnik', 'url': 'https://tile.openstreetmap.org/{z}/{x}/{y}.png'}*, *scale_options=None*)

A MapModule for Leaflet maps that uses a user-defined portrayal method to generate a portrayal of a raster Cell or a GeoAgent.

For a raster Cell, the portrayal method should return a (r, g, b, a) tuple.

For a GeoAgent, the portrayal method should return a dictionary.

- For a Line or a Polygon, the available options can be found at: https://leafletjs.com/reference.html#path-option
- For a Point, the available options can be found at: https://leafletjs.com/reference.html#circlemarker-option
- In addition, the portrayal dictionary can contain a "description" key, which will be used as the popup text.

Create a new MapModule.

Parameters

- **portrayal_method** – A method that takes a GeoAgent (or a Cell) and returns a dictionary of options (or a (r, g, b, a) tuple) for Leaflet.js.
- **view** – The initial view of the map. Must be set together with zoom. If both view and zoom are None, the map will be centered on the total bounds of the space. Default is None.
- **zoom** – The initial zoom level of the map. Must be set together with view. If both view and zoom are None, the map will be centered on the total bounds of the space. Default is None.
- **map_width** – The width of the map in pixels. Default is 500.
- **map_height** – The height of the map in pixels. Default is 500.
- **tiles** – An optional tile layer to use. Can be a `RasterWebTile` or a `xyzservices.TileProvider`. Default is *xyzservices.providers.OpenStreetMap.Mapnik*.

  If the tile provider requires registration, you can pass the API key inside the *options* parameter of the `RasterWebTile` constructor.

  For example, to use the *Mapbox* raster tile provider, you can use:

```
import mesa_geo as mg

mg.RasterWebTile(
    url="https://api.mapbox.com/v4/mapbox.satellite/{z}/{x}/{y}.png?
↪access_token={access_token}",
    options={
        "access_token": "my-private-ACCESS_TOKEN",
        "attribution": '&copy; <a href="https://www.mapbox.com/about/
↪maps/" target="_blank">Mapbox</a> &copy; <a href="https://www.
↪openstreetmap.org/copyright">OpenStreetMap</a> contributors <a␣
↪href="https://www.mapbox.com/map-feedback/" target="_blank">
↪Improve this map</a>',
    },
)
```

  Note that *access_token* can have different names depending on the provider, e.g., *api_key* or *key*. You can check the documentation of the provider for more details.

  xyzservices provides a list of providers requiring registration as well: https://xyzservices.readthedocs.io/en/stable/registration.html

  For example, you may use the following code to use the *Mapbox* provider:

```
import xyzservices.providers as xyz

xyz.MapBox(id="<insert map_ID here>", accessToken="my-private-ACCESS_
↪TOKEN")
```

- **scale_options** – A dictionary of options for the map scale. Default is None (no map scale). The available options can be found at: https://leafletjs.com/reference.html#control-scale-option

local_includes: ClassVar = ['js/MapModule.js', 'css/external/leaflet.css', 'js/external/leaflet.js']

local_dir = PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/mesa-geo/checkouts/stable/mesa_geo/visualization/templates')

js_code = ''

render(*model*)

Build visualization data from a model object.

Args:
model: A model object

Returns:
A JSON-ready object.

package_includes: ClassVar = []

render_args: ClassVar = {}

3.3.6 Tile Layers

class RasterWebTile(*url: str*, *options: dict[str, LeafletOption] | None = None*, *kind: str = 'raster_web_tile'*)

A class for the background tile layer of Leaflet map that uses a raster tile server as the source of the tiles.

The available options can be found at: https://leafletjs.com/reference.html#tilelayer

url: str

options: dict[str, LeafletOption] | None = None

kind: str = 'raster_web_tile'

classmethod from_xyzservices(*provider: TileProvider*) → *RasterWebTile*

Create a RasterWebTile from an xyzservices TileProvider.

Parameters
provider – The xyzservices TileProvider to use.

Returns
A RasterWebTile instance.

to_dict() → dict

class WMSWebTile(*url: str*, *options: dict[str, LeafletOption] | None = None*, *kind: str = 'wms_web_tile'*)

A class for the background tile layer of Leaflet map that uses a WMS service as the source of the tiles.

The available options can be found at: https://leafletjs.com/reference.html#tilelayer-wms

kind: str = 'wms_web_tile'

classmethod from_xyzservices(*provider: TileProvider*) → *RasterWebTile*

Create a RasterWebTile from an xyzservices TileProvider.

Parameters
provider – The xyzservices TileProvider to use.

Returns
A RasterWebTile instance.

options: dict[str, LeafletOption] | None = None

to_dict() → dict

url: str

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

v

INDEX

O

P

R

S

T

U

V

W