Mesa-Geo Documentation

Release 0.9.3

Project Mesa Team

Jun 10, 2026

CONTENTS

1 Using Mesa-Geo **3**

2 Contributing to Mesa-Geo **5**

3 Citing Mesa-Geo **7**

3.1 Mesa-Geo Introductory Model 7

3.2 Examples 22

3.3 APIs 25

4 Indices and tables **45**

Python Module Index **47**

Index **49**

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/mesa/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 click the "Cite this repository" button in the right sidebar of the repository landing page, and choose either the APA or BibTeX citation format.

3.1 Mesa-Geo Introductory Model

To overview the critical parts of Mesa-Geo this tutorial uses the pandemic modelling approach known as a S(usceptible), I(infected) and R(ecovered) or SIR model.

Components of the model are:

Agents: Each agent in the model represents an individual in the population. Agents have states of susceptible, infected, recovered, or dead. The Agents are point agents, randomly placed into the environment.

Environment: The environment is a set of polygons of a few Toronto neighborhoods.

Interaction Rules: Susceptible agents can become infected with a certain probability, if they come into contact with infected agents. Infected agents then recover after a certain period or perish based on a probability.

Parameters:

- Population Size (number of human agents in the model)
- Initial Infection (percent of the population initial infected)
- Exposure Distance (proximity suscpetible agents must be to infected agents to possibly get infected)
- Infection Risk (probability of becoming infected)
- Recovery Rate (time infection lasts)
- Mobility (distance agent moves)

The tutorial then proceeds in three parts:

- Part 1 Create the Basic Model
- Part 2 Add Agent Behaviors and Model Complexity
- Part 3 Add Visualizations and Interface

(You can use the table of contents button on the left side of the interface to skip to any specific part)

Users can use Google Colab* (Please ensure you run the Colab dependency import cell- below)

*Based on a recent Google Colab update currently, Solara visuals are not rendering. However, the link still provides a easy way to download the jupyter file. You can see the issue on the Solara GitHub site

You can also download the file directly from GitHub

```
# Run this if in colab or if you need to install mesa and mesa-geo in your local␣
↪environment.
pip install -U --pre mesa-geo --quiet
mkdir -p data
wget -P data https://raw.githubusercontent.com/mesa/mesa-geo/main/docs/tutorials/data/
↪TorontoNeighbourhoods.geojson
```

3.1.1 Part 1 Create the Basic Model

This portion initializes the human agents, the neighborhood agents, and the model class that manages the model dynamics.

First we import our dependencies

This cell imports the specific libraries we need to create our model.

- Shapley a library GIS library for object in the cartesian plane. From Shapely we specifically need the Point class to create our human agents
- Mesa the parent ABM library to Mesa-Geo

Then of course mesa-geo which although not strictly necessary we also specifically import the visualization part of the library so we do not have to write out mesa-geo.visualization modules when we call them.

```
from shapely.geometry import Point

import mesa
from mesa.visualization import SolaraViz, make_plot_component
import mesa_geo as mg
from mesa_geo.visualization import make_geospace_component
```

Create the person agent class

The person in this model represents one human being and we initialize each person agent with two key parts:

1. The agent attributes, such as recovery rate and death risk
2. The step function, actions the agent will take each model step

The first thing we are going to do is create the person agent class. This class has several attributes necessary to make a more holistic model.

First, there are the required attributes for any GeoAgent in Mesa-Geo

- **model**: Model object class that we will build later, this is a pointer to the model instance so the agent can get information from the model as it behaves
- **geometry**: GIS geometric object in this case a GIS Point
- **crs**: A string describing the coordinate reference system the agent is using

As you can see these are inherited from the mesa-geo librarary through the "mg.GeoAgent" in the class instantiation.

Second, the variable attributes these are unique to our SIR model:

- **agent_type**: A string which describes the agent state (susceptible, infected, recovered, or dead)
- **mobility_range**: Distance the agent can move in meters
- **infection risk**: A float from 0.0 to 1.0 that determines the risk of the agent being infected if exposed.
- **recovery_rate**: A float from 0.0 to 1.0 that determine how long the agents takes to recover

- **death_risk**: A float from 0.0 to 1.0 that determines the probability the agent will die

The **__repr__** function is a Python primitive that will print out information as directed by the code. In this case we will print out the agent ID

The **step** function is a Mesa primitive that describes what action the agent takes each step

```
class PersonAgent(mg.GeoAgent):
    """Person Agent."""

    def __init__(
        self,
        model,
        geometry,
        crs,
        agent_type,
        mobility_range,
        infection_risk,
        recovery_rate,
        death_risk,
    ):
        super().__init__(model, geometry, crs)
        # Agent attributes
        self.atype = agent_type
        self.mobility_range = mobility_range
        self.infection_risk = infection_risk
        self.recovery_rate = recovery_rate
        self.death_risk = death_risk

    def __repr__(self):
        return f"Person {self.unique_id}"

    def step(self):
        print(repr(self))
        print(self.atype, self.death_risk, self.recovery_rate)
```

Create the neighborhood agent

The neighborhood in this model represents one geographical area as defined by the geojson file we uploaded.

Similar to the person agent, we initialize each neighborhood agent with the same two key parts.

1. The agent attributes, such as geometry and state of neighborhood
2. The step function, behaviors the agent will take during each model step.

Similar to the person agent for the neighborhood agent there are two types of attributes.

The required attributes for any GeoAgent in Mesa-Geo:

- **model**: Model object class that we will build later, this is a pointer to the model instance so the agent can get information from the model as it behaves
- **geometry**: GIS geometric object in this case a polygon form the geojson defining the perimeter of the neighborhood
- **crs**: A string describing the coordinate reference system the agent is using

Similar to the person agent, "mg.GeoAgent" is inherited from mesa-geo.

Next are the variable attributes:

- **agent_type**: A string which describes the state of the neighborhood which will be either safe or hot spot
- **hotspot_threshold**: An integer that is the number of infected people in a neighborhood to call it a hotspot

We will also use the **__repr__** function to print out the agent ID

Then the **step** function, which is a primitive that describes what action the agent takes each step

```
class NeighbourhoodAgent(mg.GeoAgent):
    """Neighbourhood agent. Changes color according to number of infected inside it."""

    def __init__(self, model, geometry, crs, agent_type="safe", hotspot_threshold=1):
        super().__init__(model, geometry, crs)
        self.atype = agent_type
        self.hotspot_threshold = (
            hotspot_threshold  # When a neighborhood is considered a hot-spot
        )

    def __repr__(self):
        return f"Neighbourhood {self.HOODNUM}"

    def step(self):
        """Advance agent one step."""
        print(repr(self))
```

Create the Model Class

The model class is the manager that instantiates the agents, then manages what is happening in the model through the step function, and collects data.

We will create the model with parameters that will set the attributes of the agents as it instantiates them and a step function to call the agent step function.

First, we name our class in this case GeoSIR and we inherit the model class from Mesa. We store the path to our GeoJSON file in the object geojson regions. As JSONs mirror Pythons dictionary structure, we store the key for the neighbourhood id ("HOODNUM") as attribute, and update the **repr** function to print out the neighbourhood id.

Second, we set up the python initializer to initiate our model class. To do this we will, set up key word arguments or kwargs of the parameters we want for our model. In this case we will use:

- population size (pop_size): An integer that determines the number of person agents
- initial infection (init_infection): A float between 0.0 and 1.0 which determines what percentage of the population is infected as the model initiates
- exposure_distance (exposure_dist): An integer for the distance in meters a susceptible person agent must within to be infected by a person agent who is infected
- maximum infection risk (max_infection_risk): A float between 0.0 and 1.0 of which determines the highest suscpetibility rate in the population

Third, we initialize our agents. Mesa-Geo has an AgentCreator class inside is geoagent.py file that can create GeoAgents from files, GeoDataFrames, GeoJSON or Shapely objects.

Creating the NeighbourhoodAgents

In this case we will use the `torontoneighbourhoods.geojson` file located in the data folder to to create the NeighbourhoodAgents. Next, we will add them to the environment with the space.add_agents function.

Creating the PersonAgents

We will use Mesa-Geo AgentCreator to create the person agents. To create a heterogeneous (diverse) population we will use the random object created as part of Mesa's base class to help initialize the population's parameters.

- death_risk: A float from 0 to 1
- agent_type: Compares the model parameter of initial infection of a random float between 0 and 1 and the initial infection parameter. If it is less than the initial infection parameter the agent is initialized as infected.
- recover: Is an integer between 1 and the recovery rate. This determines the number of steps it takes for the agent to recover.
- infection_risk: is a float between 0 and the parameter of max_infection_risk, which will then determine how likely a person is to get infected.
- death_risk: Is a random float between 0 and 1 that will determine how likely a person is to die when infected.

By using Python's random library to create these attributes for each agent, we can now create a diverse agent population.

Passing these parameters through the `AgentCreator` class we initialize our agent object.

As Mesa-Geo is an GIS based ABM, we need assign each PersonAgent a Geometry and location. To do this we will use a helper function `find_home`. This helper function first identifies a NeighbourhoodAgent where the PersonAgent will start. Next it identifies the center of the neighborhood and its boundary and then randomly moving from the center point, put staying within the bounds, it a lat and long to aissgn the PersonAgent is starting location.

Step Function

The final piece is to initialize a step function. This function is a Mesa primitive that iterates through each agent calling their step function.

The Model

We know have the pieces of our Model. A GIS layer of polygons that creates NeighbourhoodAgents from our GeoJSON file. A diverse population of GIS Point objects, with different infection, recovery and death risks. A model class that initializes these agents, a GIS space and step function to execute the simulation

```
class GeoSIR(mesa.Model):
    """Model class for a simplistic infection model."""

    # Geographical parameters for desired map
    geojson_regions = "data/TorontoNeighbourhoods.geojson"

    def __init__(
        self,
        pop_size=30,
        mobility_range=500,
        init_infection=0.2,
        exposure_dist=500,
        max_infection_risk=0.2,
        max_recovery_time=5,
    ):
        super().__init__()
        self.space = mg.GeoSpace(warn_crs_conversion=False)

        # SIR model parameters
        self.pop_size = pop_size
        self.mobility_range = mobility_range
        self.initial_infection = init_infection
```

(continues on next page)

(continued from previous page)

```
        self.exposure_distance = exposure_dist
        self.infection_risk = max_infection_risk
        self.recovery_rate = max_recovery_time

        # Set up the Neighbourhood patches for every region in file
        ac = mg.AgentCreator(NeighbourhoodAgent, model=self)
        neighbourhood_agents = ac.from_file(self.geojson_regions)

        # Add neighbourhood agents to space
        self.space.add_agents(neighbourhood_agents)

        # Generate random location, add agent to grid
        for i in range(pop_size):
            # assess if they are infected
            if self.random.random() < self.initial_infection:
                agent_type = "infected"
            else:
                agent_type = "susceptible"
            # determine movement range
            mobility_range = self.random.randint(0, self.mobility_range)
            # determine agent recovery rate
            recover = self.random.randint(1, self.recovery_rate)
            # determine agents infection risk
            infection_risk = self.random.uniform(0, self.infection_risk)
            # determine agent death probability
            death_risk = self.random.random()

            # Generate PersonAgent population
            unique_person = mg.AgentCreator(
                PersonAgent,
                model=self,
                crs=self.space.crs,
                agent_kwargs={
                    "agent_type": agent_type,
                    "mobility_range": mobility_range,
                    "recovery_rate": recover,
                    "infection_risk": infection_risk,
                    "death_risk": death_risk,
                },
            )

            x_home, y_home = self.find_home(neighbourhood_agents)

            this_person = unique_person.create_agent(Point(x_home, y_home))
            self.space.add_agents(this_person)

    def find_home(self, neighbourhood_agents):
        """Find start location of agent"""

        # identify location
        this_neighbourhood = self.random.randint(
            0, len(neighbourhood_agents) - 1
```

(continues on next page)

(continued from previous page)

```
        )  # Region where agent starts
        center_x, center_y = neighbourhood_agents[
            this_neighbourhood
        ].geometry.centroid.coords.xy
        this_bounds = neighbourhood_agents[this_neighbourhood].geometry.bounds
        spread_x = int(
            this_bounds[2] - this_bounds[0]
        )  # Heuristic for agent spread in region
        spread_y = int(this_bounds[3] - this_bounds[1])
        this_x = center_x[0] + self.random.randint(0, spread_x) - spread_x / 2
        this_y = center_y[0] + self.random.randint(0, spread_y) - spread_y / 2

        return this_x, this_y

    def step(self):
        """Run one step of the model."""
        # Activate PersonAgents in random order
        self.agents_by_type[PersonAgent].shuffle_do("step")
        # For NeighbourhoodAgents the order doesn't matter, since they update␣
↪independently from each other
        self.agents_by_type[NeighbourhoodAgent].do("step")
```

Run The Base Model

#explanatory

This cell is fairly simple

1 - We instantiate the SIR model by call the class name "GeoSIR" into the object `model`.

2 - Then we call the step function to see if it prints out the Agent IDs, infection status, death_risk, and recovery rate as called in the PersonAgent class.

You can also see all the person agents are called and then the neighbourhood agents. This will become important later as we want to update the neighbourhood status later based on its PersonAgent status.

If you are curious about the numbers for the neighbourhood agents, you can open up the GeoJSON in the data folder and see that each neighborhood gets a `HOODNUM` attribute assigned.

```
model = GeoSIR()
model.step()
```

3.1.2 Part 2 Add Agent and Model Complexity

Increase PersonAgent Complexity

In this section we add behaviors to the PersonAgent to build the necessary SIR dynamics.

To create the SIR dynamics we need the agents move, determine if they have been exposed and if they have process the probability of them being infected and possibly dying.

To do this we will update our step function. The step function logic uses the agent's `atype` to determine what actions to process

Part 1

If the PersonAgent `atype` is susceptible, then we need to identify all PersonAgent's neighbors within the exposure distance. To do this, we will use Mesa-Geo's `get_neighbors_within_distance` function which takes 2 parameters, the agent, and a distance, which in this case is the model parameter for exposure distance in meters. This creates a list of PersonAgents within that distance.

The `get_neighbors_within_distance` function has two keyword arguments `center` and `relation`. `center` takes `True` or `False` on whether to include the center, it is set to `False` and measures as a buffer around the agent's geometry. If `True` it measures from the Center of the point. `relation` is defaulted to `intersects` but can take any common spatial relationship, such as `contains`, `within`, `touches`, `crosses`

The step function then iterates through the list of neighbors to see if any agents are infected. If so it does a probabilistic comparison of a random float compared to the agents infection risk and if `True` the agent becomes infected and the iteration ends.

Part 2

If the agent `atype` is infected, then the step function does comparisons. First, it sees how many steps the agent has been infected. To track this the PersonAgent got a new attribute counter which is `steps_infected`. If the steps are greater than or equal to their recovery rate, the agent is recovered, if not then the function does a probabilistic comparison with the agents death risk to see if the agent dies. If neither of these things happen the `steps_infected` increases by one.

Part 3

The next part is if the agent `atype` is not dead then the agent moves. For this we randomly get an integer for the x any (lat and long) between their negative `mobility_range` and positive `mobility range`. We pass these two integers into the helper function `move_point` and then update the agents geometry with this new point.

Finally, we update the counts of agent types.

```
class PersonAgent(mg.GeoAgent):
    """Person Agent."""

    def __init__(
        self,
        model,
        geometry,
        crs,
        agent_type,
        mobility_range,
        infection_risk,
        recovery_rate,
        death_risk,
    ):
        super().__init__(model, geometry, crs)
        # Agent attributes
        self.atype = agent_type
        self.mobility_range = mobility_range
        self.infection_risk = (infection_risk,)
        self.recovery_rate = recovery_rate
        self.death_risk = death_risk
        self.steps_infected = 0
        self.steps_recovered = 0

    def __repr__(self):
        return f"Person {self.unique_id}"

    # Helper function for moving agent
```

(continues on next page)

(continued from previous page)

```
    def move_point(self, dx, dy):
        """
        Move a point by creating a new one
        :param dx:  Distance to move in x-axis
        :param dy:  Distance to move in y-axis
        """
        return Point(self.geometry.x + dx, self.geometry.y + dy)

    def step(self):

        # Part 1 - find neighbors based on infection distance
        if self.atype == "susceptible":
            neighbors = self.model.space.get_neighbors_within_distance(
                self, self.model.exposure_distance
            )
            for neighbor in neighbors:
                if (
                    neighbor.atype == "infected"
                    and self.random.random() < self.model.infection_risk
                ):
                    self.atype = "infected"
                    break  # stop process if agent becomes infected

        # Part -2 If infected, check if agent recovers or agent dies
        elif self.atype == "infected":
            if self.steps_infected >= self.recovery_rate:
                self.atype = "recovered"
                self.steps_infected = 0
            elif self.random.random() < self.death_risk:
                self.atype = "dead"
            else:
                self.steps_infected += 1

        elif self.atype == "recovered":
            self.steps_recovered += 1
            if self.steps_recovered >= 2:
                self.atype = "susceptible"
                self.steps_recovered = 0

        # Part 3 - If not dead, move
        if self.atype != "dead":
            move_x = self.random.randint(-self.mobility_range, self.mobility_range)
            move_y = self.random.randint(-self.mobility_range, self.mobility_range)
            self.geometry = self.move_point(move_x, move_y)  # Reassign geometry

        self.model.counts[self.atype] += 1  # Count agent type
```

Increase NeighbourhoodAgent Complexity

For the NeighbourhoodAgent we want to change their color based on the number of infected PersonAgents in their neighbourhood.

To do this we will create a helper function called `color_hotspot`. We will then use mesa-geo's

`get_intersecting_agents` function. We will then iterate through that list to get the agents with `atype` infected if the list is longer than our `hotspot_threshold` equal to 1 (so if two agents in the neighborhood are infected) then the `atype` will change to `hotspot`.

We then update our model counts.

```
class NeighbourhoodAgent(mg.GeoAgent):
    """Neighbourhood agent. Changes color according to number of infected inside it."""

    def __init__(self, model, geometry, crs, agent_type="safe", hotspot_threshold=1):
        super().__init__(model, geometry, crs)
        self.atype = agent_type
        self.hotspot_threshold = (
            hotspot_threshold  # When a neighborhood is considered a hot-spot
        )

    def __repr__(self):
        return f"Neighbourhood {self.unique_id}"

    def color_hotspot(self):
        # Decide if this region agent is a hot-spot
        # (if more than threshold person agents are infected)
        neighbors = self.model.space.get_intersecting_agents(self)
        infected_neighbors = [
            neighbor for neighbor in neighbors if neighbor.atype == "infected"
        ]
        if len(infected_neighbors) > self.hotspot_threshold:
            self.atype = "hotspot"
        else:
            self.atype = "safe"

    def step(self):
        """Advance agent one step."""
        self.color_hotspot()
        self.model.counts[self.atype] += 1  # Count agent type
```

Increase model complexity

For this section will add data collection where we collect the status of the PersonAgents and the NeighbourhoodAgents but counting the different `atypes`.

As we run our SIR model, we want to ensure we are collecting information about the status of the disease.

To do this we will create helper functions that get this information. In this case we will put them in a separate cell, but depending on the developers preference they could also put them in the model class or collect the information in a handful of other ways.

In this case, we set up an attribute in the model called counts and these functions just get the total number from Mesa's data collector of each of our statuses.

```
# Functions needed for datacollector
def get_infected_count(model):
    return model.counts["infected"]
```

(continues on next page)

(continued from previous page)

```
def get_susceptible_count(model):
    return model.counts["susceptible"]


def get_recovered_count(model):
    return model.counts["recovered"]


def get_dead_count(model):
    return model.counts["dead"]


def get_hotspot_count(model):
    return model.counts["hotspot"]


def get_safe_count(model):
    return model.counts["safe"]
```

Now to finish the model so we can add the interface we add datacollection and a stop condition. As these updates are interspersed throughout the class. The comment `#added` is used to make the changes easier to identify.

First, we add an attribute called `self.counts` which will track our the agent types (e.g. infected). We will initialize it as None. We then initialize the counts in our next line `self.reset_counts()`. This helper function located directly above the step function, resets the counts of each type of agent so it is always based on the current situation in the Model.

We are then going to add the attribute self.running so we can input the stop condition. Next we set our our data collector that call our functions from the previous cell which collects our agent types

With these added we can now call `self.reset_counts` and `self.datacollector.collect` in our step function so it collect our agent states each step.

Finally we add a stop condition. If no PersonAgent is infected the pandemic is over and we stop the model.

```
class GeoSIR(mesa.Model):
    """Model class for a simplistic infection model."""

    # Geographical parameters for desired map
    geojson_regions = "data/TorontoNeighbourhoods.geojson"

    def __init__(
        self,
        pop_size=30,
        mobility_range=500,
        init_infection=0.2,
        exposure_dist=500,
        max_infection_risk=0.2,
        max_recovery_time=5,
    ):
        super().__init__()
        # Space
        self.space = mg.GeoSpace(warn_crs_conversion=False)
        # Data Collection
```

(continues on next page)

(continued from previous page)

```
        self.counts = None  # added
        self.reset_counts()  # added

        # SIR model parameters
        self.pop_size = pop_size
        self.mobility_range = mobility_range
        self.initial_infection = init_infection
        self.exposure_distance = exposure_dist
        self.infection_risk = max_infection_risk
        self.recovery_rate = max_recovery_time
        self.running = True  # added
        # added
        self.datacollector = mesa.DataCollector(
            {
                "infected": get_infected_count,
                "susceptible": get_susceptible_count,
                "recovered": get_recovered_count,
                "dead": get_dead_count,
                "safe": get_safe_count,
                "hotspot": get_hotspot_count,
            }
        )

        # Set up the Neighbourhood patches for every region in file
        ac = mg.AgentCreator(NeighbourhoodAgent, model=self)
        neighbourhood_agents = ac.from_file(self.geojson_regions)

        # Add neighbourhood agents to space
        self.space.add_agents(neighbourhood_agents)

        # Generate random location, add agent to grid
        for i in range(pop_size):
            # assess if they are infected
            if self.random.random() < self.initial_infection:
                agent_type = "infected"
            else:
                agent_type = "susceptible"
            # determine movement range
            mobility_range = self.random.randint(0, self.mobility_range)
            # determine agent recovery rate
            recover = self.random.randint(1, self.recovery_rate)
            # determine agents infection risk
            infection_risk = self.random.uniform(0, self.infection_risk)
            # determine agent death probability
            death_risk = self.random.uniform(0, 0.05)

            # Generate PersonAgent population
            unique_person = mg.AgentCreator(
                PersonAgent,
                model=self,
                crs=self.space.crs,
                agent_kwargs={
```

(continues on next page)

(continued from previous page)

```
                    "agent_type": agent_type,
                    "mobility_range": mobility_range,
                    "recovery_rate": recover,
                    "infection_risk": infection_risk,
                    "death_risk": death_risk,
                },
            )

            x_home, y_home = self.find_home(neighbourhood_agents)

            this_person = unique_person.create_agent(Point(x_home, y_home))
            self.space.add_agents(this_person)

    def find_home(self, neighbourhood_agents):
        """Find start location of agent"""

        # identify location
        this_neighbourhood = self.random.randint(
            0, len(neighbourhood_agents) - 1
        )  # Region where agent starts
        center_x, center_y = neighbourhood_agents[
            this_neighbourhood
        ].geometry.centroid.coords.xy
        this_bounds = neighbourhood_agents[this_neighbourhood].geometry.bounds
        spread_x = int(
            this_bounds[2] - this_bounds[0]
        )  # Heuristic for agent spread in region
        spread_y = int(this_bounds[3] - this_bounds[1])
        this_x = center_x[0] + self.random.randint(0, spread_x) - spread_x / 2
        this_y = center_y[0] + self.random.randint(0, spread_y) - spread_y / 2

        return this_x, this_y

    # added
    def reset_counts(self):
        self.counts = {
            "susceptible": 0,
            "infected": 0,
            "recovered": 0,
            "dead": 0,
            "safe": 0,
            "hotspot": 0,
        }

    def step(self):
        """Run one step of the model."""

        self.reset_counts()  # added

        # Activate PersonAgents in random order
        self.agents_by_type[PersonAgent].shuffle_do("step")
        # For NeighbourhoodAgents the order doesn't matter, since they update␣
```

(continues on next page)

(continued from previous page)

```
↪independently from each other
        self.agents_by_type[NeighbourhoodAgent].do("step")

        self.datacollector.collect(self)  # added

        # Run until no one is infected
        if self.counts["infected"] == 0:
            self.running = False
```

To test our code we will run the model through 5 steps and then call the model dataframe via data collector with `get_model_vars_dataframe()`. This will show a Pandas DataFrame.

```
model = GeoSIR()
for i in range(5):
    model.step()

model.datacollector.get_model_vars_dataframe()
```

3.1.3 Part 3 - Add Interface

Adding the interface requires three steps:

1. Define the agent portrayal
2. Set the sliders for the model parameters
3. Call the model through the Mesa-Geo visualization model

Visualizing agents is done through a function that is is passed in as a parameter. By default agents they are Point geometries are rendered as circles. However, Mesa uses ipyleaflet Users can pass through any Point geometry for their Agent (i.e. Marker, Circle, Icon, AwesomeIcon). To show this we will use different colors for the PersonAgent base don infection status and if they die, we will use the Font Awesome Icons and represent them with an x, in the traditional ipyleaflet marker.

We will also change the color of the NeighbourhoodAgent based whether or not it is a hotspot

Next we will build Sliders for each of our input parameters. These use the Solara's input approach. This is stored in a dictionary of dictionaries that is then passed through in the model instantiation.

If you want the model to fill the entire screen you can hit the expand button in the upper right.

```
def SIR_draw(agent):
    """
    Portrayal Method for canvas
    """
    portrayal = {}
    if isinstance(agent, PersonAgent):
        if agent.atype == "susceptible":
            portrayal["color"] = "Green"
        elif agent.atype == "infected":
            portrayal["color"] = "Red"
        elif agent.atype == "recovered":
            portrayal["color"] = "Blue"
        else:
            portrayal["marker_type"] = "AwesomeIcon"
```

(continues on next page)

(continued from previous page)

```
            portrayal["name"] = "times"
            portrayal["icon_properties"] = {
                "marker_color": "black",
                "icon_color": "white",
            }

    if isinstance(agent, NeighbourhoodAgent):
        if agent.atype == "hotspot":
            portrayal["color"] = "Red"
        else:
            portrayal["color"] = "Green"

    return portrayal


model_params = {
    "pop_size": {
        "type": "SliderInt",
        "value": 80,
        "label": "Population Size",
        "min": 0,
        "max": 100,
        "step": 1,
    },
    "mobility_range": {
        "type": "SliderInt",
        "value": 500,
        "label": "Max Possible Agent Movement",
        "min": 100,
        "max": 1000,
        "step": 50,
    },
    "init_infection": {
        "type": "SliderFloat",
        "value": 0.4,
        "label": "Initial Infection",
        "min": 0.0,
        "max": 1.0,
        "step": 0.1,
    },
    "exposure_dist": {
        "type": "SliderInt",
        "value": 800,
        "label": "Exposure Distance",
        "min": 100,
        "max": 1000,
        "step": 50,
    },
    "max_infection_risk": {
        "type": "SliderFloat",
        "value": 0.7,
        "label": "Maximum Infection Risk",
```

(continues on next page)

(continued from previous page)

```
        "min": 0.0,
        "max": 1.0,
        "step": 0.1,
    },
    "max_recovery_time": {
        "type": "SliderInt",
        "value": 7,
        "label": "Maximum Number of Steps to Recover",
        "min": 1,
        "max": 10,
        "step": 1,
    },
}
```

To create the model with the interface we use Mesa's GeoJupyterViz module. First we pass in the model class and next the parameters. We then switch to key word arguments. First measures, in this case of list of lists, where the first list will be a chart of the PersonAgent statuses and the second chart will be the NeighbourhoodAgent statuses. We also pass in a name, our agent portrayal function a zoom level and in this case set the scroll wheel zoom to false.

```
model = GeoSIR()
page = SolaraViz(
    model,
    name="GeoSIR",
    model_params=model_params,
    components=[
        make_geospace_component(SIR_draw, zoom=12, scroll_wheel_zoom=False),
        make_plot_component(["infected", "susceptible", "recovered", "dead"]),
        make_plot_component(["safe", "hotspot"]),
    ],
)
# This is required to render the visualization in the Jupyter notebook
page
```

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: 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: CRS | None

Return the coordinate reference system of the object.

abstractmethod 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(*model*, *geometry*, *crs*)

Base class for a geo model agent.

Create a new agent.

Parameters

- **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: 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

classmethod create_agents(*model: Model*, *n: int*, **args*, ***kwargs*) → AgentSet[T]

Create N agents.

Args:

model: the model to which the agents belong args: arguments to pass onto agent instances

each arg is either a single object or a sequence of length n

n: the number of agents to create kwargs: keyword arguments to pass onto agent instances

each keyword arg is either a single object or a sequence of length n

Returns:

AgentSet containing the agents created.

property crs: CRS | None

Return the coordinate reference system of the object.

classmethod from_dataframe(*model: Model*, *df: pd.DataFrame*, ***kwargs*) → AgentSet[T]

Create agents from a pandas DataFrame.

Each row of the DataFrame represents one agent. The DataFrame columns are mapped to the agent's constructor as keyword arguments. Additional keyword arguments (***kwargs*) can be used to set constant attributes for all agents.

Args:

model: The model instance. df: The pandas DataFrame. Each row represents an agent. **kwargs: Constant values to pass to every agent's constructor.

Only non-sequence data is allowed in kwargs to avoid ambiguity with DataFrame columns.

Returns:

AgentSet containing the agents created.

Note:

If you need to pass variable data or sequences, add them as columns to the DataFrame before calling this method.

property random: Random

Return a seeded stdlib rng.

remove() → None

Remove and delete the agent from the model.

Notes:

If you need to do additional cleanup when removing an agent by for example removing it from a space, consider extending this method in your own agent class.

property rng: Generator

Return a seeded np.random rng.

property scenario

Return the scenario associated with the model.

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.

property crs

Return the coordinate reference system of the GeoAgents.

create_agent(*geometry*)

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

Parameters
geometry – The geometry of the agent.

Returns
The created agent.

Return type
self.agent_class

from_GeoDataFrame(*gdf*, *set_attributes=True*)

Create a list of agents from a GeoDataFrame.

Parameters

- **gdf** – The GeoDataFrame to create agents from.
- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

from_file(*filename*, *set_attributes=True*)

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

Parameters

- **filename** – The vector data file to create agents from.
- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

from_GeoJSON(*GeoJSON*, *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.
- **set_attributes** – Set agent attributes from GeoDataFrame columns. Default True.

3.3.3 GeoSpace

class GeoSpace(*crs='epsg:3857'*, ** (Keyword-only parameters separator (PEP 3102))*, *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* | GeoDataFrame]

Return a list of all layers in the Geospace.

property total_bounds: ndarray | None

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

add_layer(*layer:* ImageLayer | RasterLayer | *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=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: 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: ndarray | None

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

Returns
Bounds of the raster layer 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] | None = None*, ***, *rowcol: tuple[int, int] | None = None*, *xy: tuple[float, float] | ndarray[tuple[Any, ...], dtype[float]] | None = None*) → bool

Determine whether a coordinate is outside the raster extent.

Exactly one selector must be provided.

Parameters

- **pos** (*Coordinate | None*) – Grid position in `(grid_x, grid_y)` format with origin at lower left.
- **rowcol** (*Coordinate | None*) – Raster indices in `(row, col)` format with origin at upper left.
- **xy** (*FloatCoordinate | None*) – Continuous `(x, y)` coordinate in CRS units.

Returns
True if the selected coordinate is off the raster, False otherwise.

Return type
bool

Raises
ValueError – If selector arguments are invalid.

property crs: CRS | None

Return the coordinate reference system of the object.

class Cell(*model*, *pos=None*, *indices=None*, ***, *rowcol=None*, *xy=None*)

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

Deprecated:
Cell.indices is deprecated. Use *Cell.rowcol* instead.

Initialize a cell.

Parameters

- **pos** – Grid position of the cell in (grid_x, grid_y) format. Origin is at lower left corner of the grid
- **indices** – (Deprecated) Indices of the cell in (row, col) format. Origin is at upper left corner of the grid. Use rowcol instead.
- **rowcol** – Indices of the cell in (row, col) format. Origin is at upper left corner of the grid
- **xy** – Geographic/projected (x, y) coordinates of the cell center in the CRS.

property pos: tuple[int, int] | None

Grid position in (grid_x, grid_y) format with origin at lower left of the grid.

property indices: tuple[int, int] | None

Deprecated alias of *rowcol*.

property rowcol: tuple[int, int] | None

Raster indices in (row, col) format with origin at upper left of the grid.

property xy: tuple[float, float] | ndarray[tuple[Any, ...], dtype[float]] | None

Geographic/projected (x, y) coordinates of the cell center in the CRS.

step()

A single step of the agent.

advance() → None

classmethod create_agents(*model: Model*, *n: int*, **args*, ***kwargs*) → AgentSet[T]

Create N agents.

Args:

model: the model to which the agents belong args: arguments to pass onto agent instances

each arg is either a single object or a sequence of length n

n: the number of agents to create kwargs: keyword arguments to pass onto agent instances

each keyword arg is either a single object or a sequence of length n

Returns:

AgentSet containing the agents created.

classmethod from_dataframe(*model: Model*, *df: pd.DataFrame*, ***kwargs*) → AgentSet[T]

Create agents from a pandas DataFrame.

Each row of the DataFrame represents one agent. The DataFrame columns are mapped to the agent's constructor as keyword arguments. Additional keyword arguments (***kwargs*) can be used to set constant attributes for all agents.

Args:

model: The model instance. df: The pandas DataFrame. Each row represents an agent. **kwargs: Constant values to pass to every agent's constructor.

Only non-sequence data is allowed in kwargs to avoid ambiguity with DataFrame columns.

Returns:

AgentSet containing the agents created.

Note:

If you need to pass variable data or sequences, add them as columns to the DataFrame before calling this method.

property random: Random

Return a seeded stdlib rng.

remove() → None

Remove and delete the agent from the model.

Notes:

If you need to do additional cleanup when removing an agent by for example removing it from a space, consider extending this method in your own agent class.

property rng: Generator

Return a seeded np.random rng.

property scenario

Return the scenario associated with the model.

class RasterLayer(*width*, *height*, *crs*, *total_bounds*, *model*, *cell_cls: type[Cell] = 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: ndarray*, *attr_name: str | Sequence[str] | None = None*) → None

Apply raster data to the cells.

Parameters

- **data** (*np.ndarray*) – 3D numpy array with shape (bands, height, width).
- **attr_name** (*str | Sequence[str] | None*) – Attribute name(s) to be added to the cells. For multi-band rasters, pass a list of names with length equal to the number of bands, or a single base name to be suffixed per band. If None, names are generated. Default is None.

Raises
ValueError – If the shape of the data does not match the raster.

get_raster(*attr_name: str | Sequence[str] | None = None*) → ndarray

Return the values of given attribute.

Parameters
attr_name (*str | Sequence[str] | None*) – Name(s) of attributes to be returned. If None, returns all attributes. Default is None.

Returns
The values of given attribute(s) as a numpy array with shape (bands, height, width).

Return type
np.ndarray

get_random_xy(*cell: Cell | None = None*, ***, *pos: tuple[int, int] | None = None*, *rowcol: tuple[int, int] | None = None*) → tuple[float, float] | ndarray[tuple[Any, ...], dtype[float]]

Generate random continuous (x, y) coordinates within a specific raster cell.

Exactly one of `cell`, `pos`, or `rowcol` must be provided.

Parameters

- **cell** (*Cell | None*) – Cell to sample from.
- **pos** (*Coordinate | None*) – Grid coordinate in `(grid_x, grid_y)` format with origin at lower left.
- **rowcol** (*Coordinate | None*) – Raster index in `(row, col)` format with origin at upper left.

Returns
Random continuous `(x, y)` coordinate within the selected cell in CRS units.

Return type
FloatCoordinate

Raises
ValueError – If selector arguments are invalid or out of bounds.

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*) – Grid coordinate tuple (grid_x, grid_y) for the neighborhood to get. Origin is at lower left corner of the grid.
- **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 (grid_x, grid_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]

property crs: 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

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*) – Grid coordinate tuple (grid_x, grid_y) for the neighborhood to get. Origin is at lower left corner of the grid.
- **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 (grid_x, grid_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*]

out_of_bounds(*pos: tuple[int, int] | None = None*, ***, *rowcol: tuple[int, int] | None = None*, *xy: tuple[float, float] | ndarray[tuple[Any, ...], dtype[float]] | None = None*) → bool

Determine whether a coordinate is outside the raster extent.

Exactly one selector must be provided.

Parameters

- **pos** (*Coordinate | None*) – Grid position in `(grid_x, grid_y)` format with origin at lower left.
- **rowcol** (*Coordinate | None*) – Raster indices in `(row, col)` format with origin at upper left.
- **xy** (*FloatCoordinate | None*) – Continuous `(x, y)` coordinate in CRS units.

Returns
True if the selected coordinate is off the raster, False otherwise.

Return type
bool

Raises
ValueError – If selector arguments are invalid.

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: ndarray | None

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

Returns
Bounds of the raster layer 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

iter_cell_list_contents(*positions*) → Any

get_cell_list_contents(*positions*) → Any

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

Return a list of cell coordinates that are in the neighborhood of a certain point.

Parameters

- **pos** (*Coordinate*) – Grid coordinate tuple (grid_x, grid_y) for the neighborhood to get. Origin is at lower left corner of the grid.
- **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 (grid_x, grid_y) cell as well. Otherwise, return surrounding cells only. Default is False.
- **radius** (*int*) – Radius, in cells, of the neighborhood. Default is 1.

Returns
A list of 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
List[Coordinate]

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 raster layer to a new coordinate reference system.

Parameters

- **crs** – The coordinate reference system to transform to.

- **inplace** – Whether to transform the raster layer in place or return a new raster layer. Defaults to False.

Returns
The transformed raster layer if not inplace.

Return type
RasterLayer | None

to_image(*colormap*) → *ImageLayer*

Returns an ImageLayer colored by the provided colormap.

classmethod from_file(*raster_file: str*, *model: Model*, *cell_cls: type[Cell] = Cell*, *attr_name: str | Sequence[str] | None = None*, *rio_opener: Callable | 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 | Sequence[str] | None*) – Attribute name(s) to use for the cell values. For multi-band rasters, pass a list of names with length equal to the number of bands, or a single base name to be suffixed per band. If None, names are generated. Default is None.
- **rio_opener** (*Callable | None*) – A callable passed to Rasterio open() function.

to_file(*raster_file: str*, *attr_name: str | Sequence[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 | Sequence[str] | None*) – The name(s) of attributes 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: 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] | None = None*, ***, *rowcol: tuple[int, int] | None = None*, *xy: tuple[float, float] | ndarray[tuple[Any, ...], dtype[float]] | None = None*) → bool

Determine whether a coordinate is outside the raster extent.

Exactly one selector must be provided.

Parameters

- **pos** (*Coordinate | None*) – Grid position in `(grid_x, grid_y)` format with origin at lower left.
- **rowcol** (*Coordinate | None*) – Raster indices in `(row, col)` format with origin at upper left.
- **xy** (*FloatCoordinate | None*) – Continuous `(x, y)` coordinate in CRS units.

Returns
True if the selected coordinate is off the raster, False otherwise.

Return type
bool

Raises
ValueError – If selector arguments are invalid.

`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: ndarray | None`

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

Returns
Bounds of the raster layer 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 image layer to a new coordinate reference system.

Parameters

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

Returns
The transformed image layer if not inplace.

Return type
ImageLayer | 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

class LeafletViz(*style: dict[str, str | bool | int | float] | None = None*, *popupProperties: dict[str, str | bool | int | float] | 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

class MapModule(*portrayal_method*, *tiles*)

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.
- **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")
```

make_geospace_component(*agent_portrayal*, *view=None*, *tiles=xyzservices.providers.OpenStreetMap.Mapnik*, ***kwargs*)

Create a Solara component that displays a Leaflet map for a model's GeoSpace.

This function returns a factory callable that can be supplied to Mesa's *SolaraViz* to embed an interactive Leaflet map showing the model's *GeoSpace*. The map is rendered using ipyleaflet and will draw raster layers, vector layers, and agents with their portrayals, using a user-provided *agent_portrayal* function.

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.

Parameters

- **agent_portrayal** – 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** – Initial map center as `(latitude, longitude)`. If not provided, the map is centered from `model.space.total_bounds`.
- **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")
```

- ****kwargs** – Extra keyword arguments forwarded to `ipyleaflet.Map` (e.g., `zoom=`, `scroll_wheel_zoom=`). The available options can be found at: https://ipyleaflet.readthedocs.io/en/latest/api_reference/index.html#ipyleaflet.leaflet.Map

Returns

A factory callable to be passed as a SolaraViz component.

Return type

Callable[[mesa.Model], solara.Element]

Warning

When using this component with `SolaraViz`, pass the list of components via the `components=` keyword argument (not as a positional argument). See the SolaraViz docs: https://mesa.readthedocs.io/latest/apis/visualization.html

Example

Define a custom portrayal for agents and add a map component to SolaraViz:

```
import mesa_geo as mg
from mesa.visualization import SolaraViz, make_plot_component
from mesa_geo.visualization import make_geospace_component

def agent_portrayal(agent):
    # Return Leaflet style options or RGBA tuple
    if isinstance(agent, mg.GeoAgent):
        return {"radius": 4, "color": "blue"}
    elif isinstance(agent, mg.Cell):
        return (255, 0, 0, 1)  # Red color for raster cells

page = SolaraViz(
    model,
    name="Geo Model",
    model_params=model_params,
    components=[
        make_geospace_component(agent_portrayal),
        make_plot_component(["happy", "unhappy"]),
    ],
)
```

3.3.6 Tile Layers

class RasterWebTile(*url: str*, *options: dict[str, str | bool | int | float] | 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, str | bool | int | float] | 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, str | bool | int | float] | 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, str | bool | int | float] | None = None

to_dict() → dict

url: str

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

mesa_geo.geo_base, 25
mesa_geo.geoagent, 26
mesa_geo.geospace, 28
mesa_geo.raster_layers, 30
mesa_geo.tile_layers, 43

v

visualization.__init__, 40

INDEX

A

add_agents() (*GeoSpace method*), 29
add_layer() (*GeoSpace method*), 29
advance() (*Cell method*), 32
advance() (*GeoAgent method*), 27
AgentCreator (*class in mesa_geo.geoagent*), 27
agents (*GeoSpace property*), 29
agents_at() (*GeoSpace method*), 30
apply_raster() (*RasterLayer method*), 34
attributes (*RasterLayer property*), 34

C

Cell (*class in mesa_geo.raster_layers*), 32
cells (*RasterLayer attribute*), 34
coord_iter() (*RasterLayer method*), 34
create_agent() (*AgentCreator method*), 28
create_agents() (*Cell class method*), 33
create_agents() (*GeoAgent class method*), 27
crs (*AgentCreator property*), 28
crs (*GeoAgent property*), 27
crs (*GeoBase property*), 26
crs (*GeoSpace property*), 30
crs (*ImageLayer property*), 38
crs (*RasterBase property*), 32
crs (*RasterLayer property*), 35

D

distance() (*GeoSpace method*), 30

F

from_dataframe() (*Cell class method*), 33
from_dataframe() (*GeoAgent class method*), 27
from_file() (*AgentCreator method*), 28
from_file() (*ImageLayer class method*), 40
from_file() (*RasterLayer class method*), 38
from_GeoDataFrame() (*AgentCreator method*), 28
from_GeoJSON() (*AgentCreator method*), 28
from_xyzservices() (*RasterWebTile class method*), 43
from_xyzservices() (*WMSWebTile class method*), 44

G

GeoAgent (*class in mesa_geo.geoagent*), 26
GeoBase (*class in mesa_geo.geo_base*), 25
GeoSpace (*class in mesa_geo.geospace*), 28
get_agents_as_GeoDataFrame() (*GeoSpace method*), 30
get_cell_list_contents() (*RasterLayer method*), 37
get_intersecting_agents() (*GeoSpace method*), 30
get_neighborhood() (*RasterLayer method*), 37
get_neighboring_cells() (*RasterLayer method*), 37
get_neighbors() (*GeoSpace method*), 30
get_neighbors_within_distance() (*GeoSpace method*), 30
get_random_xy() (*RasterLayer method*), 34
get_raster() (*RasterLayer method*), 34
get_relation() (*GeoSpace method*), 30
get_transformed_geometry() (*GeoAgent method*), 26

H

height (*ImageLayer property*), 38
height (*RasterBase property*), 31
height (*RasterLayer property*), 35

I

ImageLayer (*class in mesa_geo.raster_layers*), 38
indices (*Cell property*), 32
iter_cell_list_contents() (*RasterLayer method*), 37
iter_neighborhood() (*RasterLayer method*), 35
iter_neighbors() (*RasterLayer method*), 35

K

kind (*RasterWebTile attribute*), 43
kind (*WMSWebTile attribute*), 44

L

layers (*GeoSpace property*), 29
LeafletViz (*class in visualization.__init__*), 40

M

make_geospace_component() (*in module visualization.__init__*), 41
MapModule (*class in visualization.__init__*), 40

mesa_geo.geo_base
    module, 25
mesa_geo.geoagent
    module, 26
mesa_geo.geospace
    module, 28
mesa_geo.raster_layers
    module, 30
mesa_geo.tile_layers
    module, 43
module
    mesa_geo.geo_base, 25
    mesa_geo.geoagent, 26
    mesa_geo.geospace, 28
    mesa_geo.raster_layers, 30
    mesa_geo.tile_layers, 43
    visualization.__init__, 40

O

options (*RasterWebTile attribute*), 43
options (*WMSWebTile attribute*), 44
out_of_bounds() (*ImageLayer method*), 39
out_of_bounds() (*RasterBase method*), 31
out_of_bounds() (*RasterLayer method*), 36

P

pos (*Cell property*), 32

R

random (*Cell property*), 33
random (*GeoAgent property*), 27
RasterBase (*class in mesa_geo.raster_layers*), 30
RasterLayer (*class in mesa_geo.raster_layers*), 33
RasterWebTile (*class in mesa_geo.tile_layers*), 43
remove() (*Cell method*), 33
remove() (*GeoAgent method*), 27
remove_agent() (*GeoSpace method*), 29
resolution (*ImageLayer property*), 39
resolution (*RasterBase property*), 31
resolution (*RasterLayer property*), 36
rng (*Cell property*), 33
rng (*GeoAgent property*), 27
rowcol (*Cell property*), 32

S

scenario (*Cell property*), 33
scenario (*GeoAgent property*), 27
step() (*Cell method*), 32
step() (*GeoAgent method*), 27

T

to_crs() (*GeoAgent method*), 26
to_crs() (*GeoBase method*), 26
to_crs() (*GeoSpace method*), 29
to_crs() (*ImageLayer method*), 40
to_crs() (*RasterBase method*), 31
to_crs() (*RasterLayer method*), 37
to_dict() (*RasterWebTile method*), 43
to_dict() (*WMSWebTile method*), 44
to_file() (*RasterLayer method*), 38
to_image() (*RasterLayer method*), 38
total_bounds (*GeoAgent property*), 26
total_bounds (*GeoBase property*), 25
total_bounds (*GeoSpace property*), 29
total_bounds (*ImageLayer property*), 39
total_bounds (*RasterBase property*), 31
total_bounds (*RasterLayer property*), 36
transform (*ImageLayer property*), 39
transform (*RasterBase property*), 31
transform (*RasterLayer property*), 37
transformer (*GeoSpace property*), 29

U

url (*RasterWebTile attribute*), 43
url (*WMSWebTile attribute*), 44

V

values (*ImageLayer property*), 39
visualization.__init__
    module, 40

W

width (*ImageLayer property*), 39
width (*RasterBase property*), 30
width (*RasterLayer property*), 37
WMSWebTile (*class in mesa_geo.tile_layers*), 43

X

xy (*Cell property*), 32