Documentation of StateSpace class

Class Constructor: `__init__`

Initializes the state space model with specific configurations and parameters, defining the initial setup, boundary conditions, and transition algorithms.

Parameters

`num_colors` **(int)** Number of distinct colors (or states) in the model.

`grid_size` **(int)** Dimensions of the grid (NxN).

`beta` **(float)** Inverse temperature, influences the dynamics of state transitions, controlling the probability of changes and their responsiveness to the system's conditions.

`init` **(int or str, optional)** Specifies the initial state configuration. 0 for uniform, 'random' for random. Default is 0.

`bc` **(int, optional)** Boundary condition for the grid's edges. Defines how edge cases are handled, affecting the model's spatial dynamics. Default is 0 (free boundary conditions).

`algo` **(str, optional)** The algorithm used for state transitions. Options are 'metropolis' or 'glauber'. Default is 'metropolis'.

Example

```
# Class Initialization Example
m = StateSpace(num_colors=3, grid_size=64, beta=1.2, init=0, bc=0,
    algo='metropolis')
```

Methods

`step`

Performs simulation steps, updating the grid state and optionally tracking specified observables.

Parameters

`num_steps` (int, optional) The number of simulation steps to perform. Default is 1.

`progress_bar` (bool, optional) If True, shows a progress bar. Useful for long simulations. Default is True.

`sample_rate` (int, optional) Determines sampling frequency for observables. Lower values increase data resolution. Default is 10k.

`observables` (list of functions, optional) Specific observables to monitor. Default is None.

Returns

None – This method updates the grid and observable states in-place.

Example

```python
# Performing Simulation Steps
m.step(num_steps=1000, progress_bar=True, sample_rate=100,
    observables = [m.avg_links])
```

Utility Methods

Utility methods provide various functionalities from retrieving the current grid configuration to computing statistical measures and checking the grid's state legality.

`get_grid()`

Returns the current grid configuration.

Parameters: None.

Returns: `ndarray` – An array of shape $(num\_colors, grid\_size, grid\_size, 2)$ representing the grid's current state.

`get_local_time(x, y)`

Calculates the local time at a specified grid point, indicating the duration or frequency of a specific state at that point.

Parameters:

`x, y (int, int)` – Coordinates of the grid point.

Returns: `float` – The local time at the specified grid point.

`get_local_time_i(c, x, y)`

Determines the local time for a specific color at a given grid point, useful for analyzing color-specific dynamics.

Parameters:

`c (int)` – Color index.

`x, y (int, int)` – Coordinates of the grid point.

Returns: `float` – The local time for the specified color at the given location.

`max_links(), avg_links(), avg_local_time()`

Compute maximum, average links, and average local time across the grid, providing insights into connectivity and state persistence.

Parameters: None.

Returns:

For `max_links()`: `int` – Maximum number of links.

For `avg_links()`: `float` – Average number of links per grid point.

For `avg_local_time()`: `float` – Average local time across the grid.

`loop_builder()`

Constructs loops based on the current state of the grid, identifying closed paths that may indicate stable or recurring configurations.

Parameters: None.

Returns: `list` – A collection of loops, each represented as a sequence of grid points.

`avg_loop_length()`

Calculates the average length of loops within the grid, offering a measure of the complexity or stability of the grid's structure.

Parameters: None.

Returns: `float` – The average length of the loops found in the grid.

`check_state()`

Validates the current grid configuration, ensuring it adheres to predefined rules or constraints.

Parameters: None.

Returns: `bool` or `str` – True if the state is valid, otherwise returns a description of the validation issue.

Visualization Methods

Visualization methods offer capabilities to plot the grid state, specific colors, loops, and overlay configurations, aiding in the qualitative analysis of the system's behavior.

`plot_one_color(c, cmap, ax, alpha=1.0, linewidth=1.0)`

Visualizes the grid with a single color highlighted, enabling focused analysis on specific states.

Parameters:

`c` **(int)** Color index to be visualized.

`cmap` **(Colormap)** The colormap for the visualization.

`ax` **(matplotlib axis)** Axis object for plotting.

`alpha` **(float, optional)** Transparency of the color. Default is 1.0.

`linewidth` **(float, optional)** Line width for the plot. Default is 1.0.

Returns: None – This method directly visualizes the grid on the given axis.

`plot_loop(c, loop, color='yellow', alpha=0.25)`

Displays a specific loop within the grid, useful for highlighting structures or patterns.

Parameters:

`c` **(int)** Color index of the loop.

`loop` **(sequence)** Sequence of coordinates defining the loop.

`color` **(str, optional)** Color for the loop visualization. Default is 'yellow'.

`alpha` **(float, optional)** Transparency of the loop. Default is 0.25.

Returns: None – Renders the specified loop on the grid visualization.

`plot_grid(figsize=(10,8), linewidth=1.0, colorbar=True, file_name=None)`

Creates a comprehensive plot of the entire grid, showcasing all colors and states.

Parameters:

`figsize` **(tuple, optional)** Figure dimension. Default is (10, 8).

`linewidth` **(float, optional)** Width of grid lines. Default is 1.0.

`colorbar` **(bool, optional)** Whether to include a colorbar. Default is True.

`file_name` **(str or None, optional)** Path to save the figure. Default is None.

Returns: None – Outputs a visual representation of the grid.

`plot_overlap(figsize=(12,12), normalized=False, file_name=None, alpha=0.7, linewidth=1.5)`

Overlays multiple grid configurations for comparative analysis, with options for normalization.

Parameters:

`figsize` **(tuple, optional)** Size of the figure. Default is (12, 12).

`normalized` **(bool, optional)** Whether to normalize overlay plots. Default is False.

`file_name` **(str or None, optional)** If specified, saves the figure to this path. Default is None.

`alpha` **(float, optional)** Transparency level for the overlays. Default is 0.7.

`linewidth` **(float, optional)** Line width used in the plot. Default is 1.5.

Returns: None – Displays the overlaid grid configurations.

`summary()`

Generates and prints a summary of the current state space statistics, offering a quick overview of system dynamics.

Parameters: None.

Returns: None – Outputs a summary of key statistical measures directly.

Examples

```
# Plotting a Single Color Example
state_space.plot_one_color(c=1, cmap='viridis', ax=plt.gca(), alpha
    =0.8, linewidth=0.5)

# Plotting Overlaid Grid Configurations Example
state_space.plot_overlap(figsize=(10, 10), normalized=True, alpha
    =0.5, linewidth=0.75)
```

Mistral version

Python Code Documentation

This documentation provides an overview of a Python class named `stateSpace` that simulates a coloring problem on a grid. The class uses various libraries such as numpy, matplotlib, and tqdm for different functionalities.

Class: stateSpace

The `stateSpace` class is used to simulate a coloring problem on a grid. The class contains various methods to initialize the grid, update the grid, and calculate different properties of the grid.

Initialization

The class is initialized with the following parameters:

- `num_colors`: The number of colors to be used in the simulation.
- `grid_size`: The size of the grid.
- `beta`: A parameter used in the acceptance probability calculation.
- `init`: The initialization method for the grid. It can be 'random' or a number.
- `bc`: The boundary condition for the grid. It can be 'random' or a number.
- `algo`: The algorithm to be used for the simulation. It can be 'metropolis' or 'glauber'.

Methods

The class contains various methods to perform different operations on the grid:

- `random_init()`: This method initializes the grid with random colors.
- `uniform_init(k)`: This method initializes the grid with a uniform color `k`.
- `step(num_steps = 1, progress_bar = True, sample_rate = 10_000, observables = None)`: This method updates the grid for a given number of steps. It also calculates and stores different observables during the simulation.
- `minimal_transformations(S)`: This method returns a list of minimal transformations for a given square `S`.
- `get_possible_transformations(S)`: This method returns a list of all possible transformations for a given square `S`.

- `square_transformation(c, s, X)`: This method applies a transformation `X` to a square `s` of color `c`.
- `acceptance_prob(S, M, s, X, c)`: This method calculates the acceptance probability for a transformation `X` on a square `s` of color `c`.
- `get_grid()`: This method returns the current state of the grid.
- `get_local_time(x, y)`: This method calculates the local time for a given square at position `(x, y)`.
- `get_local_time_i(c, x, y)`: This method calculates the local time for a given square at position `(x, y)` for color `c`.
- `max_links()`: This method returns the maximum number of links for each color.
- `avg_links()`: This method returns the average number of links for each color.
- `avg_local_time()`: This method returns the average local time for the grid.
- `loop_builder(v1 = None, v2 = None)`: This method builds loops for each color in the grid.
- `avg_loop_length()`: This method returns the average loop length for the grid.
- `check_state()`: This method checks if the current state of the grid is legal.
- `plot_one_color(c, cmap, ax, alpha = 1.0, linewidth = 1.0)`: This method plots the grid for a given color `c`.
- `plot_loop(c, loop, color = 'yellow', alpha = 0.25, linewidth = 1.5)`: This method plots the longest loop for a given color `c`.
- `plot_grid(figsize = (10,8), linewidth = 1.0, colorbar = True, file_name = None)`: This method plots the grid for all colors.
- `plot_overlap(figsize = (12,12), normalized = False, file_name = None, alpha = 0.7, linewidth = 1.0)`: This method plots the overlap of all colors in the grid.
- `summary()`: This method prints a summary of the current state of the grid.
- `save_data(file_name)`: This method saves the current state of the grid to a file.
- `load_data(file_name)`: This method loads the state of the grid from a file.

Usage Examples

Here are some examples of how to use the `stateSpace` class:

```
# Initialize a grid with 3 colors, size 10, beta 1.0, random initialization,
# random boundary condition, and metropolis algorithm
m = stateSpace(3, 10, 1.0, init='random', bc='random', algo='metropolis')

# Update the grid for 100 steps
m.step(num_steps=100)

# Print a summary of the current state of the grid
m.summary()

# Plot the grid for all colors
m.plot_grid()

# Save the current state of the grid to a file
m.save_data('grid_data.json')

# Load the state of the grid from a file
m.load_data('grid_data.json')
```