mrrvis

Release 0.1.0

Alexander Pasha

Sep 16, 2022

CONTENTS:

CHAPTER

ONE

MRRVIS

1.1 mrrvis package

1.1.1 Submodules

1.1.2 mrrvis.cell module

The cell module defines the Cell abstract class and 4 subclasses: Square, Hex, Tri and Cube.

The Cell subclasses define the shape and characteristics of a module in a lattice bearing that name. For instance, if there is a square lattice then a cell.Square(coord) object would know the location of adjacent cells, given a connectivity level ('edge' or 'vertex'), in this case assume 'edge', and the 'compass' with which those neighbors are related to this cell, which for edge connected square cells is ['N','W','S','E'].

class mrrvis.cell.**Cell**(*coord: ndarray*)

Bases: ABC

A Cell class

Parameters
coord (*np.ndarray or list or tuple*) – The coordinate of the cell

Raises
ValueError – If coord is not a valid coordinate of the lattice used by this subclass

coord

The coordinate of the cell

Type
np.ndarray

__getitem__(*key*)

access adjacent cell coordinates as keys of this object

abstract classmethod adjacent_transformations(*connectivity: str*) → dict

Get the translations required to go to neighboring cells

Parameters
connectivity (*{'edge', 'vertex', 'face'}*) – The level of connectivity required for a cell to be considered adjacent

Returns
the translation to the adjacent cell indexed by compass direction

Return type
dict of np.ndarrays

Raises
ValueError – if connectivity is not in this class.connectivity_types

adjacents(*connectivity=None*) → dict

Get the neighbors of this cell

Parameters
connectivity (*{'edge', 'vertex', 'face'}, optional*) – The level of connectivity required for a cell to be considered adjacent if left blank, will use the broadest set allowed: face>vertex>edge

Returns
A dictionary of adjacent cells indexed by compass directions

Return type
dict of np.ndarrays

classmethod compass(*connectivity='edge'*)

The keys which are available for a given connectivity level

Parameters
connectivity (*{'edge', 'vertex', 'face'}, default 'edge'*) – The level of connectivity where a cell is considered adjacent

Returns
The list of compass directions for that level of connectivity

Return type
list of str

abstract class property connectivity_types: set

a list of the connectivity types supported by the cell

abstract class property dimensions: int

The number of dimensions of the cell.

abstract class property n_parameters: int

The expected number of parameters

class property rotation_angle: float

The rotation angle of the cell.

abstract classmethod valid_coord(*coord: ndarray*) → bool

Verify that a coordinate exists in the cell lattice

Parameters
coord (*np.ndarray or list or tuple*) – The coordinate of the cell to check

Returns
Is True if the cell is a valid coord

Return type
bool

class mrrvis.cell.**Cube**(*coord: ndarray*)

Bases: *Cell*

static adjacent_transformations(*connectivity: Literal['edge', 'vertex', 'face']*) → dict

Get the translations required to go to neighboring cells

Parameters
connectivity (*{'edge', 'vertex', 'face'}*) – The level of connectivity required for a cell to be considered adjacent

Returns
the translation to the adjacent cell indexed by compass direction

Return type
dict of np.ndarrays

Raises
ValueError – if connectivity is not in this class.connectivity_types

connectivity_types = {'edge', 'face', 'vertex'}

dimensions = 3

n_parameters = 3

classmethod valid_coord(*coord: array*) → bool
Verify that a coordinate exists in the cell lattice

Parameters
coord (*np.ndarray or list or tuple*) – The coordinate of the cell to check

Returns
Is True if the cell is a valid coord

Return type
bool

class mrrvis.cell.**Hex**(*coord: ndarray*)
Bases: *Cell*

static adjacent_transformations(*_) → dict
Get the translations required to go to neighboring cells

Parameters
connectivity (*{'edge', 'vertex', 'face'}*) – The level of connectivity required for a cell to be considered adjacent

Returns
the translation to the adjacent cell indexed by compass direction

Return type
dict of np.ndarrays

Raises
ValueError – if connectivity is not in this class.connectivity_types

connectivity_types = {'edge'}

dimensions = 2

n_parameters = 3

classmethod valid_coord(*coord: array*) → bool
remark. On a hex grid, valid coords must be in the plane x+y+z=0

class mrrvis.cell.**Square**(*coord: ndarray*)
Bases: *Cell*

static adjacent_transformations(*connectivity: Literal['edge', 'vertex']*) → dict

Get the translations required to go to neighboring cells

Parameters
connectivity (*{'edge', 'vertex', 'face'}*) – The level of connectivity required for a cell to be considered adjacent

Returns
the translation to the adjacent cell indexed by compass direction

Return type
dict of np.ndarrays

Raises
ValueError – if connectivity is not in this class.connectivity_types

connectivity_types = {'edge', 'vertex'}

dimensions = 2

n_parameters = 2

classmethod valid_coord(*coord: ndarray*) → bool

Verify that a coordinate exists in the cell lattice

Parameters
coord (*np.ndarray or list or tuple*) – The coordinate of the cell to check

Returns
Is True if the cell is a valid coord

Return type
bool

class mrrvis.cell.Tri(*coord: ndarray*)

Bases: *Cell*

classmethod adjacent_transformations(*connectivity: Literal['edge', 'vertex']*, *point_up: bool*) → dict

Get the translations required to go to neighboring cells

Parameters

- **connectivity** (*{'edge', 'vertex'}*) – The level of connectivity required for a cell to be considered adjacent
- **point_up** (*bool*) – Is the cell an upward facing triangle

Returns
the translation to the adjacent cell indexed by compass direction

Return type
dict of np.ndarrays

Raises
ValueError – if connectivity is not in this class.connectivity_types

classmethod compass(*point_up: bool*, *connectivity='edge'*) → list

The keys which are available for a given connectivity level

Parameters

- **point_up** (*bool*) – If True, get the compass of a north pointing tri, or a south pointing one otherwise

- **connectivity** (*{'edge', 'vertex', 'face'}, default 'edge'*) – The level of connectivity where a cell is considered adjacent

Returns
The list of compass directions for that level of connectivity

Return type
list of str

connectivity_types = {'edge', 'vertex'}

dimensions = 2

n_parameters = 3

classmethod point_up(*coord: ndarray*) → bool

Decide if the cell is upward_facing

Only implemented for triangle cells

Parameters
coord (*np.ndarray*) – The coord to check

Returns
returns true if and only if the triangle at this cell is upward facing

Return type
bool

Raises
ValueError – If the input coordinate is not valid

classmethod valid_coord(*coord: array*) → bool

valid coords must be in the plane x+y+z=0 or x+y+z=1

1.1.3 mrrvis.configuration module

The configuration Module defines the ConfigurationGraph object and some operations which can be performed on ConfigurationGraphs

The configuration graph is a representation of a configuration in an MRR system. It represents the collection of vertices, which are the centroids of the cells in the system, A cell type which shows how to interpret this set of vertices as a shape, the level of connectivity at which two cells are considered connected An edge set constructed from the above information.

It also allows us to perform two essential operations:

- Check if there exists an isomorphic relationship between two different configurations
- Check to see if a configuration forms a single connected component

class mrrvis.configuration.**ConfigurationGraph**(*CellPrototype: Literal['Square', 'Cube', 'Tri', 'Hex']*, *vertices: Optional[ndarray] = None*, *connect_type='edge'*)

Bases: object

A graph based representation of a reconfigurable robotics system

Parameters

- **CellPrototype** (Cell *or {'Square', 'Cube', 'Tri', 'Hex'}*) – The type of cell lattice (or alias thereof) this graph represents

- **vertices** (*np.ndarray or array-like object, optional*) – The vertices in the configuration
- **connect_type** (*{'edge', 'vertex', 'face'}*) – The level of connectivity required for two cells to be considered neighbors

Raises

- **KeyError** – if cellPrototype is invalid
- **ValueError** – if connect_type is not suitable for the given cell_type

Warns

UserWarning – called whenever one of the vertices in the vertices array is invalid

vertices

the set of vertices as an array where each row is a coordinate

Type

ndarray

__eq__(*other*)

(==) tests for isomorphism between this graph and another

__contains__(*other*)

(in) tests to see if the other item, a Sequence, is a vertex in self.V

__getitem__(*key*)

ConfigurationGraph()[key] obtains item from self.V by an integer index

See also:

Cell

mrrvis.cell.Cell

property E

the edges of the graph

property V

the vertices of the graph

add_vertices(*in_vertices: array*, *check_connectivity=True*) → *ConfigurationGraph*

Add vertices to the graph (not in-place)

Parameters

- **in_vertices** (*ndarray*) – the vertices to add
- **check_connectivity** (*bool, default=True*) – if true, check if the resulting graph is connected the resulting graph

Raises

ValueError – If in_vertices is of the wrong shape for this graph's cell type

Warns

UserWarning – warning is raised if there are invalid cells in in_vertices or if the configuration is disconnected and check_connectivity==True

Returns

The edited graph or, if the connectivity check fails, self

Return type
ConfigurationGraph

property edges

returns the edges of the graph object

edges_from(*vertex: ndarray*, *connectivity: Optional[Literal['edge', 'vertex', 'face']] = None*, *omit_self=False*)

find the list of edges which are connected to a particular vertex

Parameters

- **vertex** (*np.ndarray*) – The coordinate to find the edges of
- **connectivity** (*{'edge', 'vertex','face'}, optional*) – The level of connectivity at which two cell are considered connected, will default to this graph's connectivity
- **omit_self** (*bool, default False*) – Selects return type. If False will provide a list of edges represented as sets, if true then return just the list of the indices of the adjoining cells

Returns
if omit_self is true, gives a list of indices, otherwise return a list of edges represented as sets

Return type
list of sets of ints or list of ints

edges_from_i(*index*, *connectivity=None*, *omit_self=False*)

Obtain the edges from the cell at a given index in self.V

get_index(*vertex: ndarray*) → int

get the index of a vertex in the graph

Parameters
vertex (*np.ndarray*) – A vertex, which should be in self.V

Warns
UserWarning – If the vertex is not in the graph

Returns
The index of the vertex in self.V

Return type
int

is_connected(*connectivity: Optional[Literal['edge', 'vertex', 'face']] = None*) → bool

test graph connectivity

Performs a breadth first search for a path from the first coordinate in self.V to every other coordinate. is_connected==True if and only if every cell is_reachable from the first cell

Parameters
connectivity (*{'edge', 'vertex','face'}, optional*) – The level of connectivity required, by default will use this graph's declared cell_type

Returns
True if and only if the graph is connected

Return type
bool

is_reachable(*u: int*, *v: int*) → bool

identifies if there exists a path from the module at index u to index v using breadth first search

Parameters

- **u** (*int*) – The index of the first module in self.V
- **v** (*int*) – The index of the second module in self.V

Returns
True if there exists a path u ~> v

Return type
bool

isomorphic(*other:* ConfigurationGraph) → bool

Check if two graphs are isomorphic

Parameters
other (ConfigurationGraph) – Another graph to compare

Returns
True if and only if self and other are isomorphic

Return type
bool

Notes

if the graphs are of different connectivity types, this equation will use the type of the first operand

property n

number of modules in the graph

remove_vertices(*rm_vertices: ndarray*, *check_connectivity=True*) → *ConfigurationGraph*

Remove vertices in the graph

Parameters

- **rm_vertices** (*ndarray*) – the vertices to remove
- **check_connectivity** (*bool*) – if true, check if the resulting graph is connected

Raises
ValueError – If in_vertices is of the wrong shape for this graph's cell type

Warns

- **warning is raised if the configuration is disconnected**
- **and check_connectivity==True**

Returns
the graph with the vertices removed, or, if the connectivity check fails, self

Return type
ConfigurationGraph

mrrvis.configuration.**add_vertices**(*graph:* ConfigurationGraph, *in_vertices: ndarray*, *check_connectivity=True*) → *ConfigurationGraph*

Add vertices to the graph

See also:

add_vertices
ConfigurationGraph.add_vertices

mrrvis.configuration.**edge_connected**(*graph:* ConfigurationGraph) → bool

checks if the graph is edge connected

See also:

is_connected
ConfigurationGraph.is_connected

Notes

this is used for checking moves

mrrvis.configuration.**get_index**(*vertex: array*, *vertices: ndarray*) → int

get the index of a vertex in a set of vertices

See also:

get_index
graph.get_index

mrrvis.configuration.**max_coord**(*vertices: ndarray*) → ndarray

find the maximum coordinate in a set of vertices

Parameters
vertices (*ndarray*) –

Returns
The maximum coordinate in the input array

Return type
ndarray

Notes

This maximum works by finding the largest x, largest y and then largest z in vertices

mrrvis.configuration.**min_coord**(*vertices: ndarray*) → ndarray

find the canonical minimum coordinate in a set of vertices

Parameters
vertices (*ndarray*) –

Returns
The minimum coordinate in the input array

Return type
ndarray

Notes

This minimum works by finding the smallest x, smallest y and then smallest z in vertices

mrrvis.configuration.**remove_vertices**(*graph:* ConfigurationGraph, *rm_vertices: ndarray*, *check_connectivity=True*) → *ConfigurationGraph*

Remove vertices in the graph

See also:

rm_vertices
ConfigurationGraph.rm_vertices

mrrvis.configuration.**vert_connected**(*graph:* ConfigurationGraph) → bool

Checks if the graph is vertex connected

See also:

is_connected
ConfigurationGraph.is_connected

Notes

this is used for checking moves

1.1.4 mrrvis.env module

The env module defines the Environment object which acts as the control environment of an mrrvis system

The Environment allows us to perform the following key operations: - Initialize an environment with a history and a target state

class mrrvis.env.**Environment**(*state_0: Union[ndarray,* ConfigurationGraph*]*, *cell_type: Literal['Square', 'Cube', 'Tri', 'Hex']*, *state_target: Optional[Union[ndarray,* ConfigurationGraph*]] = None*, *moveset: Optional[dict] = None*, *connectivity: Optional[Literal['edge', 'vertex', 'face']] = None*, *additional_rules=None*)

Bases: `object`

The control environment of a Reconfigurable Robotics system

Parameters

- **state_0** (*ConfigurationGraph or ndarray*) – The initial state of the configuration
- **cell_type** (*{'Square', 'Cube', 'Tri', 'Hex'}*) – The type of cell lattice used
- **state_target** (*ConfigurationGraph or ndarray, optional*) – The target configuration
- **moveset** (*dict, optional*) – A dictionary of moves, indexed by human readable names, defaults for the cell_type will be used if left blank
- **connectivity** (*{'edge', 'vertex', 'face'}*) – The level of connection at which two cells are considered neighbors, defaults to most generous afforded by cell type
- **additional_rules** (*list of func*) – Any additional configuration rules to be applied to all moves in this environment. These should be functions which take a ConfigurationGraph as input and return a boolean truth value

state

The current state of the environment

Type
ConfigurationGraph

target_state

The target state of the environment

Type
ConfigurationGraph

moveset

A dictionary containing the moves cells can carry out and their names for reference

Type
dict of Move

history

A history containing the history of states since the environment was initialised/reset

Type
mrrvis.history.History

property action_space: Dict[str, Dict[str, *Move*]]

get the current action space

Notes

Preferably, if you know the module and the move name, just use env.actions_for_module_and_move(module, move_name)[direction] or, if you know the module, use env.actions_for_module(module)[move_name][direction] to save time and memory

actions_for_module(*module: ndarray*) → Dict[str, Dict[str, *Move*]]

return the actions for a particular module

Parameters
module (*ndarray or int*) – the coordinate or index of a vertex in the configuration

Returns
The dictionary of the actions for that module

Return type
dict

See also:

actions_for_module_and_move
actions_for_module_and_move

actions_for_module_and_move(*module: ndarray*, *move_name: str*) → Dict[str, *Move*]

return the actions for a specified module and move

Parameters
module (*ndarray or int*) – the coordinate or index of a vertex in the configuration

Returns
The dictionary of the actions for the specified move and module

Return type
dict

See also:

actions_for_module
actions_for_module

add_move(*move:* Move, *alias: Optional[str] = None*)
add a move to the moveset

Parameters

- **move** (Move) – The move to add
- **alias** (*str, optional*) – The name of the move, default is to use name of the move

auto_step(*next_state:* ConfigurationGraph) → Tuple[*ConfigurationGraph*, int, bool]
request to make a step with a move (force step)

Parameters
next_state – the configuration of the next state

Returns

- *int* – reward for this move, by default -1 on all non-terminal moves
- *bool* – flag to say if the state matches the target state

remove_move(*move_name: str*)
remove a move from the moveset with the given key

Parameters
move_name (*str*) – name of the move to remove

Raises
KeyError – If the move_name is not in the moveset

render(*axes=False*, *show=True*, *save=False*, *filename=None*, ***style*)
render the current state of the environment

Parameters

- **axes** (*bool*) – whether to show axes
- **show** (*bool*) – whether to show graph
- **save** (*bool*) – whether to save graph
- **filename** (*str*) – what name to save the graph under
- **style** (*dict*) – see matplotlib.pyplot reference for valid kwargs

render_history(*speed: int = 200*, *show=True*, *save=False*, *filename=None*, ***style*) → str
render the current state of the environment

Parameters

- **speed** (*int*) – The length of each frame in milliseconds
- **show** (*bool*) – whether to show graph
- **save** (*bool*) – whether to save graph
- **filename** (*str*) – what name to save the graph under

- **style** (*dict*) – see matplotlib.pyplot reference for valid kwargs

Returns
A HTML representation of the video

Return type
str

reset() → *ConfigurationGraph*

reset the environment to its original configuration.

Returns
The first state in this environment's history

Return type
ConfigurationGraph

revert(*k: int = 1*) → *ConfigurationGraph*

revert the environment to its state k steps ago, returning the new state

Parameters
k (*int*) – the number of steps to revert by

Return type
the configuration graph k steps ago

reward(*state_next:* ConfigurationGraph) → int

simple reward function, for automation. gives -1 for all non-terminal states

Parameters
state_next (ConfigurationGraph) – The configuration to reward

Returns
The reward provided to the state

Return type
int

step(*move_name: str*, *module: Union[ndarray, int]*, *direction: str*) → Tuple[*ConfigurationGraph*, int, bool]

request to take a step in the environment, performing move in place

Parameters

- **move_name** (*str*) – the name of the move
- **module** (*ndarray or int*) – The identity of the module to move
- **direction** (*str*) – A direction in this move's compass

Warns
UserWarning – If the move is infeasible, will cause state not to update

Returns

- *ConfigurationGraph* – the configuration of the next state
- *int* – reward for this move, by default -1 on all non-terminal moves
- *bool* – whether the next state matches the environment target state (is a terminal state)

property t

returns the number of time steps in this environment

verify(*state_next: Optional[ConfigurationGraph] = None*) → bool

verify that a state matches the target state

Parameters
state_next (*ConfigurationGraph, optional*) – the state to compare against the target, leave blank to use this env's current state

Returns
True if and only if the input state is isomorphic to this environment's target state

Return type
bool

1.1.5 mrrvis.geometry_utils module

The geometry_utils module provides utilities for performing geometric operations on matrices

r_from normal:
identify the rotation matrix of a normalised vector

norm_from_str:
generate a norm from a string representing the cartesian axes that the rotation is in

rotate_normal:
rotate an array of vertices using angle and axis rotation

cube_rotation_list/square_rotation_list/hex_rotation_list/tri_rotation_list:
generators for the rotation groups of different shapes in a given lattice type

mrrvis.geometry_utils.**cube_rotation_list**(*coords: ndarray*) → Generator

generator of 24 rotations in the regular octahedral rotation group S_4 for an array of 3D cartesian coordinates

Parameters
coords (*ndarray*) – the array of coordinates to transform

Yields

- *ndarray* – a rotation of the shape formed by the input coordinates
- *Items*
- —
- **algorithm based on https** (*//stackoverflow.com/questions/16452383/how-to-get-all-24-rotations-of-a-3-dimensional-array,*) – although expanded to deal with multiple coordinates at once

mrrvis.geometry_utils.**hex_rotation_list**(*coords: ndarray*) → Generator

generator of the 6 rotations of a set of coordinates in a hexagonal lattice

Parameters
coords (*ndarray*) – the array of coordinates to transform

Yields
ndarray – a rotation of the shape formed by the input coordinates

mrrvis.geometry_utils.**norm_from_str**(*axis: str*) → ndarray

generate a normal vector from a string representing an axis

Parameters
axis (*str*) – some combination of the characters ['x','y','z']

Raises
ValueError – If non of the characters in axis are in ['x','y','z']

Returns
a vector pointing in the positive direction for the given axis

Return type
ndarray

mrrvis.geometry_utils.**r_from_normal**(*angle: float*, *normal: ndarray*, *round=True*) → ndarray

obtain rotation matrix from a unit vector normal to the plane of rotation

Parameters

- **angle** (*float*) – angle to rotate by in radians
- **normal** (*ndarray*) – the vector normal to the plane of rotation
- **round** (*bool*) – round the rotation matrix to Four decimal places if true; to avoid floating-point errors at an expense to accuracy for irregular angles

Returns
a rotation matrix for the given axis and angle

Return type
ndarray

mrrvis.geometry_utils.**rotate_normal**(*array: ndarray*, *turns: int*, *base_angle: float = 1.5707963267948966*, *around: Optional[ndarray] = None*, *axis: Optional[Union[str, array]] = None*) → ndarray

rotate an set of points counterclockwise around a point, by default the origin in discrete space

Parameters

- **array** (*ndarray*) – array of points to rotate
- **turns** (*int*) – number of turns to rotate by
- **base_angle** (*float*) – angle to rotate by
- **around** (*ndarray*) – point to rotate around
- **axis** (*str or ndarray*) – axis to rotate around either as a combination of ['x','y','z'] or as a vector normal to the desired plane of rotation

Returns

- *ndarray* – the rotated array
- *Items*
- ——
- *for a description of axis and angle rotation, see*
- **https** (*//en.wikipedia.org/wiki/Rotation_matrix#Axis_and_angle*)

mrrvis.geometry_utils.**square_rotation_list**(*coords: ndarray*) → Generator

generator of the 4 rotations of a set of coordinates in a square lattice

Parameters
coords (*ndarray*) – the array of coordinates to transform

Yields
ndarray – a rotation of the shape formed by the input coordinates

mrrvis.geometry_utils.**tri_rotation_list**(*coords: ndarray*) → Generator

generator of the 3 rotations of a set of coordinates in a triangular lattice

Parameters
coords (*ndarray*) – the array of coordinates to transform

Yields
ndarray – a rotation of the shape formed by the input coordinates

1.1.6 mrrvis.history module

The history module defines the History object, which stores a sequence of configurations

class mrrvis.history.**History**(*state_0: ConfigurationGraph*)

Bases: object

A History of Configurations

Parameters
state_0 (ConfigurationGraph) – The initial configuration

history

A queue of the configuration history

Type
collections.deque

cell_type

The type of cell used by this object's configurations

Type
str

__iter__()

Iterate through history

__reversed__()

obtain the reverse of the history

__contains__(*other*)

append(*state: ConfigurationGraph*)

Append a move to the history

Parameters
state (ConfigurationGraph) – the state to add

revert(*k: int = 1*)

revert the history by n steps

Parameters
k (*int*) – the number of steps to revert by

Returns
The last state after the reversion

Return type
ConfigurationGraph

property t

The amount of items (time steps) in the history

1.1.7 mrrvis.move module

contains the Move class and its dependancies

Move is a callable object which produces a new graph configuration if possible or returns None if not.

CollisionCheck is a NamedTuple which contains two fields:

- empty: the locations which need to be empty for a move to be feasible
- full: the locations which need to be full for a move to be feasible

CollisionCheck represents a single collision case for a move, one move can contain multiple collision cases

Transformation is a NamedTuple which contains three fields:

- location: the index of the module to be moved
- translation: the translation vector of the move
- collisions: a list of CollisionCheck objects

Transformation represents a single module transformation, one move can contain multiple transformations if more than

- one module is relocated

Checkwrapper is a way to wrap a candidate configuration so that additional checks can be made upon it for testing the validity

- of the resulting move
- it simply takes a ConfigurationGraph as a value, wraps it and allows checks to be made on its' feasibility
- these checks are simply functions which take a graph as their only argument and return a boolean

`class mrrvis.move.Checkwrapper(`*value:* ConfigurationGraph`)`

Bases: `object`

Wrap a ConfigurationGraph in this object to pipeline checks on the feasibility of that configuration

Parameters
value (ConfigurationGraph) – The graph to be wrapped

value

The graph within the wrapper

Type
ConfigurationGraph

Notes

CheckWrapper is a variation on the Failure Monad design pattern which allows us to pipeline checks on the feasibility of a configuration.

a useful explanation is given at https://medium.com/swlh/monads-in-python-e3c9592285d6

bind(*f: callable*) → *Checkwrapper*

Bind any function that takes a ConfigurationGraph and returns a boolean to the wrapped value

Parameters
f (*the function to bind*) –

Returns
A new CheckWrapper which contains a value if the move still appears feasible

Return type
CheckWrapper

Notes

if true, then return the current wrapped value of the graph if false, then return wrapped None if a previous check has failed and the value is already None, then return None again

unwrap()

get the unwrapped value

class mrrvis.move.**Collision**(*cases: List[CollisionCase]*, *eval_rule: Literal['or', 'and', 'xor']*)

Bases: NamedTuple

a collection of collision cases which can be evaluated with a number of different behaviours

Parameters

- **cases** (*list*) – the collection of cases to evaluate
- **eval_rule** (*{'or','and', 'xor'}*) – the rule governing the evaluation of this collision

Notes

A Collision is feasible if the list of collision cases meets the given condition:
'or' means that the collision is feasible if any case is true 'xor' means that the collision is feasible if exactly one case is true 'and' means that the collision is feasible if all cases are True

cases: List[*CollisionCase*]

Alias for field number 0

eval_rule: Literal['or', 'and', 'xor']

Alias for field number 1

evaluate_feasible(*graph*) → bool

evaluation of collision :param graph: the graph to evaluate against :return: true if no collision, false if one is detected

class mrrvis.move.**CollisionCase**(*empty: ndarray*, *full: ndarray*)

Bases: NamedTuple

A collision case :param empty: the array of coordinates which need to be empty for the case to be true :type empty: ndarray :param full: the array of coordinates which need to be full for the case to be true :type full: ndarray

empty: ndarray

Alias for field number 0

evaluate_case(*graph: ConfigurationGraph*) → bool

evaluate collision for this case

Parameters
graph (*Configuration Graph*) – graph to check collision against

Returns
true if no collision detected

Return type
bool

full: ndarray

Alias for field number 1

rotate(*turns*, *base_angle=1.5707963267948966*, *around: Optional[ndarray] = None*, *axis=None*) → *CollisionCase*

rotate both arrays in the collision object

Parameters

- **turns** (*int*) – the number of times the base angle to rotate by (counter clockwise)
- **base_angle** (*float*) – the angle to rotate per turn
- **around** (*ndarray, optional*) – the coordinate to rotate around
- **axis** (*str*) – the axis string (some combination of ['x','y','z']) or normal vector to rotate around

Notes

This comes in handy when we need to rotate a basic collision example around a compass for a move

class mrrvis.move.**Move**(*configuration: ConfigurationGraph*, *module: Union[int, ndarray]*, *direction: str*, *additional_checks: Optional[List[Callable]] = None*, *verbose=False*, *check_connectivity=True*)

Bases: ABC

A Move which can be called as a function to generate a new configuration, if one would exist for the move so defined

Parameters

- **configuration** (ConfigurationGraph) – The configuration graph to be edited
- **module** (*int or ndarray*) – The module to be moved
- **direction** (*str*) – The direction for the move to be carried out in (see self.__class__.compass to get the compass of this move)
- **additional_checks** (*list of funcs*) – environment specific checks that the resulting configuration would have to take, on top of those already defined by the move. such should be passed as a list of functions where those functions take a Configuration graph as input and return a bool as output
- **verbose** (*whether to provide warning information to explain the infeasability of a move*) –

checklist

the list of checks to be carried out upon the candidate configuration

Type
list of funcs

__call__()

evaluate the move's feasibility and return the resulting configuration, if one exists

attempt_transaction(*transaction: Iterable[*Transformation*]*) → Optional[*ConfigurationGraph*]

attempt to construct a new graph from the transaction

Parameters
transaction (*list of Transformations*) – the list of transformations to be performed in this move

Returns
the graph after this transaction is applied or none

Return type
ConfigurationGraph or None

abstract class property cell_type: Literal['Square', 'Hex', 'Tri', 'Cube']

the type of the cell that the move is designed for

abstract class property compass: List[str]

a list of valid compass directions for this move

evaluate() → Optional[*ConfigurationGraph*]

evaluate the validity of a move

Returns
graph after having completed the move, or None if the move is infeasible

Notes

to do this we need to complete three steps: 1. generate a transaction (implemented by subclasses) 2. attempt transaction, to assert the local feasibility of the move 3. evaluate the checklist on the resulting configuration

evaluate_checklist(*candidate:* ConfigurationGraph)

evaluate the checklist on the resulting configuration

Parameters
candidate (*ConfigurationGraph the graph to be evaluated*) –

Returns
candidate if feasible or None, if the move is infeasible

Return type
ConfigurationGraph or None

abstract generate_collision(*module: ndarray*) → *Collision*

Generate the collision for a single transformation

must be implemented in subclass

Parameters
module (*ndarray*) – The coordinates of the module to generate the collision for

Returns
A collision to evaluate

Return type
Collision

Notes

must be implemented in subclass

Collisions are objects containing a list of CollisionCase Objects and a collision rule, which affects the evaluation

abstract generate_transaction() → Iterable[*Transformation*]

generate a list of Transformation objects which need to be completed

Returns
The list of Transformations to carry out

Return type
list of Transformation

Notes

must be implemented in subclass

class mrrvis.move.**Transformation**(*location: ndarray*, *transformation: ndarray*, *collision:* Collision)

Bases: NamedTuple

The transformation and collision information of a single module in a transaction

Parameters

- **location** (*ndarray*) – the coordinates of the module that is being transformed
- **transformation** (*ndarray*) – the resulting location of the module(s) (can include replication by inputting a 2D array here)
- **collision** (Collision) – The Collision object to evaluate for this transformation

collision: *Collision*

Alias for field number 2

location: ndarray

Alias for field number 0

transformation: ndarray

Alias for field number 1

1.1.8 mrrvis.vistools module

tools for visualising configurations

hex_to_cart:
convert hexagonal cubic coordinates to cartesian coordinates

tri_to_cart:
convert triangular cubic coordinates to cartesian coordinates

square_patch_generator/tri_patch_generator/hex_patch_generator/cube_patch_generator:
generators which produce a set of matplotlib patches for the given lattice type

plot_configuration:
plot a configuration graph as a static plot

plot_history:
plot a mrrvis.history.History object as an animation represented with jshtml

mrrvis.vistools.**hex_patch_generator**(*vertices: ndarray*, ***style*) → Generator[list, None, None]

generator for hex patches from a list of two dimensional vertices

Parameters

- **vertices** (*ndarray*) – the vertices to generate patches for
- **style** (*dict*) – any matplotlib kwargs for styling the patches

Yields
matplotlib.patches.RegularPolygon – A polygon situated at an input vertex

mrrvis.vistools.**hex_to_cart**(*vertices: ndarray*) → ndarray

convert an array of hexagon cubic coordinates to cartesian coordinates

Parameters
vertices (*ndarray*) – a set of hexagonal cubic coords expressed as a numpy array

Returns
an array of the converted vertices

Return type
ndarray

mrrvis.vistools.**make_voxels**(*vertices: ndarray*) → ndarray

generate voxel array for cube visualisation

Parameters
vertices (*ndarray*) – the vertices to be voxelised

Returns
the voxelised array

Return type
ndarray

mrrvis.vistools.**plot_configuration**(*configuration:* ConfigurationGraph, *show=True*, *save=False*, *filepath=None*, *axes=False*, ***style*) → None

Plot a configuration with matplotlib

Parameters

- **configuration** (ConfigurationGraph) – the graph to visualise
- **show** (*bool*) – whether to show the resulting graph
- **save** (*bool*) – whether to save the resulting graph
- **filepath** (*str*) – the filepath for the graph when saving
- **axes** (*bool*) – whether to show the axes
- **style** (*dict*) – any matplotlib kwargs

mrrvis.vistools.**plot_history**(*history:* History, *speed: int = 200*, *show=True*, *save=False*, *filepath=None*, ***style*) → str

Plot a history object as an animation with matplotlib: currently only works for 2D lattices (hex, square, tri)

Parameters

- **history** (mrrvis.history.History) – the history to animate

- **speed** (*int*) – the length of each frame in milliseconds
- **show** (*bool*) – whether to show the resulting graph
- **save** (*bool*) – whether to save the resulting graph
- **filepath** (*str*) – the filepath for the graph when saving
- **axes** (*bool*) – whether to show the axes
- **style** (*dict*) – any matplotlib kwargs

Returns
if show is set to true, then it returns an html representation of the video, remark. in iPython, use iPython.display.HTML(plot_history(...,show=true,...)) to turn this into an inline video.

Return type
str

mrrvis.vistools.**square_patch_generator**(*vertices: ndarray*, ***style*) → Generator[list, None, None]
generator for square patches from a list of two dimensional vertices

Parameters

- **vertices** (*ndarray*) – the vertices to generate patches for
- **style** (*dict*) – any matplotlib kwargs for styling the patches

Yields
matplotlib.patches.RegularPolygon – A polygon situated at an input vertex

mrrvis.vistools.**tri_patch_generator**(*vertices: ndarray*, ***style*) → Generator[list, None, None]
generator for tri patches from a list of two dimensional vertices

Parameters

- **vertices** (*ndarray*) – the vertices to generate patches for
- **style** (*dict*) – any matplotlib kwargs for styling the patches

Yields
matplotlib.patches.RegularPolygon – A polygon situated at an input vertex

mrrvis.vistools.**tri_to_cart**(*vertices: ndarray*) → Tuple[ndarray, ndarray]
convert an array of Triangle cubic coordinates to cartesian coordinates

Parameters
vertices (*ndarray*) – a set of hexagonal cubic coords expressed as a numpy array

Returns

- *ndarray* – an array of the converted vertices
- *ndarray* – a vector of whether each vertex(row) of vertices corresponds to an upward pointing triangle

1.1.9 Subpackages

mrrvis.movesets package

Submodules

mrrvis.movesets.hexmoves module

class mrrvis.movesets.hexmoves.**rotate**(*configuration: ConfigurationGraph*, *module: Union[int, ndarray]*, *direction: str*, *additional_checks: Optional[List[Callable]] = None*, *verbose=False*, *check_connectivity=True*)

Bases: *Move*

A rotation of 60 degrees on a hexagonal configuration

cell_type = 'Hex'

compass = ['N', 'NE', 'SE', 'S', 'SW', 'NW']

generate_collision(*module: ndarray*) → *Collision*

Collision for hexagonal rotation move

remarks. - collision for hex rotation is an xor collision case where one of the neighbors in the rotation direction must be open and the other must be closed - we use the northbound case as a base, which we then rotate around the origin and add to the specific module location

generate_transaction() → Iterable[*Transformation*]

Transaction for hexagonal rotation move

for this we only need one transformation as this is a single component move

mrrvis.movesets.squaremoves module

Standard moveset for a square lattice

class mrrvis.movesets.squaremoves.**rotate**(*configuration: ConfigurationGraph*, *module: Union[int, ndarray]*, *direction: str*, *additional_checks: Optional[List[Callable]] = None*, *verbose=False*, *check_connectivity=True*)

Bases: *Move*

cell_type = 'Square'

compass = ['NE', 'SE', 'SW', 'NW']

generate_collision(*module*) → *Collision*

collision object for the rotation move

- collision for a rotation move requires that exactly one of two cases is true: for a northeast move (the default), either that the the north and northeast neighbors are empty or that the east and northeast neighbors are empty

generate_transaction() → List[*Transformation*]

transactions for this move consist of a single translation in the given direction

class mrrvis.movesets.squaremoves.**slide**(*configuration: ConfigurationGraph*, *module: Union[int, ndarray]*, *direction: str*, *additional_checks: Optional[List[Callable]] = None*, *verbose=False*, *check_connectivity=True*)

Bases: *Move*

slide along the surface of neighbors which form a line paralel to the module

cell_type = 'Square'

compass = ['N', 'S', 'E', 'W']

generate_collision(*module*) → *Collision*

collision for square slide

- the default cases are for a northward move which are then rotated counterclockwise to the appropriate direction
- each move has two possible cases where it is potentially valid, for a north move these would be where there are modules to the west and northwest or east and northeast, so long as one or both of these cases is true the move is valid

generate_transaction() → List[*Transformation*]

transaction for square slide

this is a single transformation where the translation is simply a move of one in a compass direction

class mrrvis.movesets.squaremoves.**slide_line**(*configuration: ConfigurationGraph*, *module: Union[int, ndarray]*, *direction: str*, *additional_checks: Optional[List[Callable]] = None*, *verbose=False*, *check_connectivity=True*)

Bases: *Move*

cell_type = 'Square'

compass = ['N', 'S', 'E', 'W']

generate_collision() → *Collision*

collision for slide line (push) move collision for a slide line move is always true, as each cell pushes the one in front

generate_transaction() → List[*Transformation*]

The transaction for a slide line move this consists of a list of transformations, where each cell moves to the position of the one in front, until the end of the line

Module contents

1.1.10 Module contents

Top-level package for mrrvis.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

U

V