mrrvis

Release 0.1.0

Alexander Pasha

Sep 15, 2022

CONTENTS:

CHAPTER

ONE

MRRVIS

1.1 mrrvis package

1.1.1 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

1.1.2 Submodules

1.1.3 mrrvis.cell module

Defines the abstract class Cell and its 4 concrete subclasses: Square, Hex, Tri, and Cube. The Cell class's concrete types are the basis for generating the information for lattice operations in the configuration Graph

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

Bases: ABC

abstract classmethod adjacent_transformations(*connectivity*, *point_up=None*) → dict

make a dictionary of the neighbors to a hypothetical cell at the origin :param connectivity: str: the connectivity type for the lattice :param point_up: bool: (for triangles) whether this cell points up :return: the dictionary of adjacent transformations

adjacents(*connectivity=None*) → dict

the neighbors of the cell :param connectivity: the connectivity type of the cell, connectivity will default to vertex except for hexagons where edge is the only valid type :type connectivity: None or Literal['edge', 'vertex', 'face'] :return: adjacents to the cell coordinate given the connectivity for all compass directions :rtype: dict[str, np.ndarray]

classmethod compass(*connectivity='edge'*)

The keys which are available for a given connectivity level :param connectivity: The connectivity rule for defining adjacency :return: a list of directions

abstract class property connectivity_types: set

a list of the connectivity types supported by the cell :return: a set of connectivity types

abstract class property dimensions: int

The number of dimensions of the cell.

abstract class property n_parameters: int

The expected number of parameters

classmethod point_up(*coord=None*) → bool

Returns True if the cell is pointing up, False if pointing down, only implemented for triangular cells.

class property rotation_angle: float

The rotation angle of the cell. note. valid for all space-filling lattices ('Square', 'Tri', 'Hex', 'Cube').

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

Returns True if the coord is valid for the cell type :param coord: the coordinate to test :type coord: np.ndarray :return: the truth value of the coord's validity :rtype: bool

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

Bases: *Cell*

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

The transformations of a cell situated at the origin :param connectivity: the connectivity type of the lattice :param point_up: is true if the cell at the origin points up :return: a dictionary with the adjacent cells :rtype: dict[‘str’, np.ndarray]

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

dimensions = 3

n_parameters = 3

classmethod valid_coord(*coord: array*) → bool

tests that the coordinate is in the lattice

Parameters
coord – a sequence with 3 elements (x,y,z), where x,y,z are integers

Returns
a boolean stating whether the coordinate is valid

all discrete cartesian coordinates are valid

class mrrvis.cell.**Hex**(*coord: array*)

Bases: *Cell*

static adjacent_transformations(**_*) → dict

The transformations of a cell situated at the origin :param connectivity: the connectivity type of the lattice :param point_up: is true if the cell at the origin points up :return: a dictionary with the adjacent cells :rtype: dict[‘str’, np.ndarray]

connectivity_types = {'edge'}

dimensions = 2

n_parameters = 3

classmethod valid_coord(*coord: array*) → bool

tests that the coordinate is in the lattice

Parameters
coord – a sequence with 3 elements (x,y,z), where x,y,z are integers

Returns
a boolean stating whether the coordinate is valid

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

Returns a dictionary of transformations for a cell situated at the origin :param connectivity: the connectivity rule for adjacency, either ‘edge’ or ‘vertex’ adjacency :return: a dictionary of coordinates which are adjacent to the origin

connectivity_types = {'edge', 'vertex'}

dimensions = 2

n_parameters = 2

classmethod valid_coord(*coord: ndarray*) → bool

tests the validity of a coordinate

Parameters
coord – a sequence with 2 elements: (x,y)

Returns
a boolean signifying whether the coordinate exists within the cell lattice

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

Bases: *Cell*

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

The transformations of a cell situated at the origin :param connectivity: the connectivity type of the lattice :param point_up: is true if the cell at the origin points up :return: a dictionary with the adjacent cells :rtype: dict[‘str’, np.ndarray]

connectivity_types = {'edge', 'vertex'}

dimensions = 2

n_parameters = 3

classmethod point_up(*coord: Optional[ndarray] = None*) → bool

check that the cell is an upward facing triangle

Parameters
coord – a sequence with 3 elements (x,y,z) where x,y,z are integers

Returns
a boolean representing whether the cell is an upward pointing triangle (true) or not

remark. The coordinate system used for triangular grids can be represented as a face connected ‘staircase of cubes’ in 3D cartesian space (with the z axis inverted). If the triangular cell is pointing down then its centroid is on the plane x+y+z=1, if it is facing up then it is on the plane x+y+z=0

classmethod valid_coord(*coord: array*) → bool

tests that the coordinate is in the lattice

Parameters
coord – a sequence with 3 elements (x,y,z), where x,y,z are integers

Returns
a boolean stating whether the coordinate is valid

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

1.1.4 mrrvis.configuration module

This module contains the module graph class

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

Bases: object

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

Parameters

- **in_vertices** – the vertices to add
- **check_connectivity** – if true, check if the resulting graph is connected the resulting graph

Returns
configuration graph with additional vertices

Raises
ValueError – if in_vertices are of wrong shape

In order for vertices to be valid, they must: 1. be of the correct shape, 2. be valid coordinates. 3. if the graph is required to be connected, then the vertices must form a connected graph when concatenated with existing vertices.

property edges

returns the edges of the graph object

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

edges which are connected to a particular vertex :param vertex: the coordinate to find edges from :param connectivity: the connectivity level required of neighbors :param omit_self: returns a list of edges if false, or the indices of connecting vertices if true :return: a list of undirected edges if omit_self if False :return: a list of connecting node indices if omit_self :rtype: List[set] or List[int]

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

get_index(*vertex: ndarray*) → int

get the index of a vertex in the graph :param vertex: the vertex coordinate in the graph :return: an integer index of the coordinate in self.V

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

test graph connectivity. :param connectivity: the level of connectivity required, depending on self.Cell, could be any of 'edge', 'vertex', 'face' :return: bool representing the status of graph connectivity This is done by performing a breadth first search of every module from the zeroth module to see if every module is reachable from the zeroth module, if so, then the graph is connected

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 :param u: index of the first cell :param v: index of the other cell :return: bool representing whether or not a path exists used for checking connectivity in graphs

isomorphic(*other:* ConfigurationGraph) → bool

Check if two graphs are isomorphic

Parameters
other – the graph to compare

Returns
bool representing whether the two graphs are isomorphic or not

accessible as ConfigurationGraph.__eq__

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

note, this is currently a brute force method, which is fast enough for two dimensional cell types who's rotation groups have an cardinality equal to the number of vertices in the shape, but slow for cubes. This is because the order of the rotation group of a shape on a 3D discrete lattice is 24, which is still small enough to be brute forced, but in future a more efficient method might need to be considered if possible

property n

number of modules in the graph

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

Remove vertices in the graph

Parameters

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

Raises
ValueError – if in_vertices are of wrong shape

Returns
the graph with the vertices removed

In order for vertices to be removed, they must: 1. be of the correct shape. Additionally, 2. If the graph is required to be connected, then the reduced graph must be connected.

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

Add vertices to the graph

Parameters

- **graph** – the graph to edit
- **in_vertices** – the vertices to add
- **check_connectivity** – if true, check if the resulting graph is connected the resulting graph

Returns
configuration graph with additional vertices

Raises
ValueError – if in_vertices are of wrong shape

In order for vertices to be valid, they must: 1. be of the correct shape, 2. be valid coordinates. 3. if the graph is required to be connected, then the vertices must form a connected graph when concatenated with existing vertices.

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

checks if the graph is edge connected alias of graph.is_connected('edge')

mrrvis.configuration.**equals**(*graph1:* ConfigurationGraph, *graph2:* ConfigurationGraph) → bool

Check if two graphs are equal

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

get the index of a vertex in a set of vertices

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

find the maximum coordinate in a set of vertices

parameters: :param vertices: the set of vertices to search :return: the index of the vertex in the set of vertices

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

find the canonical minimum coordinate in a set of vertices

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

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

Remove vertices in the graph

Parameters

- **graph** – the graph to edit
- **rm_vertices** – the vertices to remove
- **check_connectivity** – if true, check if the resulting graph is connected

Raises

ValueError – if in_vertices are of wrong shape

Returns

the graph with the vertices removed

In order for vertices to be removed, they must: 1. be of the correct shape. Additionally, 2. If the graph is required to be connected, then the reduced graph must be connected.

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

Checks if the graph is vertex connected alias of graph.is_connected('vertex')

1.1.5 mrrvis.env module

A module defining the control environment of a reconfiguration problem

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`

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

get the current action space

Returns
a dictionary of the action space

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 (actions_for_module_and_move over all moves)

Parameters
module – the coordinate of a vertex in the configuration

Returns
the dictionary of the actions for that module

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

return the actions for a specified module and move params: :param module: the coordinate of a vertex in the configuration :param move_name: the name of a move from this environment's moveset

add_move(*move:* Move, *alias: Optional[str] = None*)

add a move to the moveset params: :param move: is a move to add to the moveset :param alias: the key to give this move in the moveset dictionary

auto_step(*next_state:* ConfigurationGraph) → Tuple[*ConfigurationGraph*, int, bool]

step from an already specified action (useful for automation) params: :param: next_state: the configuration of the next state returns: :return: reward: reward for this move, by default -1 on all non-terminal moves :return: done: whether the next state matches the environment target state (is a terminal state)

remove_move(*move_name: str*)

remove a move from the moveset with the given key params: :param move_name: the key of the move to remove

render(*axes=False*, *show=True*, *save=False*, *filename=None*, ***style*)

render the current state of the environment :param axes: whether to show axes :param show: whether to show graph :param save: whether to save graph :param filename: what name to save the graph under any matplotlib kwargs can be added using the **style attribute

render_history(*speed: int = 200*, *show=True*, *save=False*, *filename=None*, ***style*) → FuncAnimation

render the history of this environment as an animation, can be saved as a gif or mp4 :param speed: the length of each frame in milliseconds :param show: whether to show graph :param save: whether to save graph :param filename: what name to save the graph under any matplotlib kwargs can be added using **style

reset() → *ConfigurationGraph*

reset the environment to its original configuration

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

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

Parameters
steps – the number of steps to revert by

Returns
the configuration graph k steps ago

reward(*state_next:* ConfigurationGraph) → int

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

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

take a step in the environment, returning the new state, reward, and a flag to indicate if the environment has reached the target state

params: :param move_name: the name of the move, which must be a key in this environment's moveset :param module: either the id or coordinate location of a module in this environment's state :param direction: a string signifying a direction in the compass of the chosen move returns: :return: the configuration of the next state :return: reward for this move, by default -1 on all non-terminal moves :return: 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 params: :param state_next: the state to compare against the target, leave blank to use this env's current state :return: bool stating if the input state is isomorphic to the target state

1.1.6 mrrvis.geometry_utils module

Utilities for performing geometric operations on matrices defines the following functions

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

params: :param coords: the array of coordinates to transform

returns: :return: a generator for the 24 different discrete 90degree rotations of a 3D shape

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 – the array of coordinates to rotate

Returns
a generator containing the 4 different rotations of the vertices

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

generate a normal vector from a string representing an axis params: :param axis: some combination of the characters ['x','y','z'] returns: :return: normal vector

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

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

Parameters: :param angle: angle to rotate by in radians :param normal: the vector normal to the plane of rotation :param discrete: whether to discretise the rotation matrix (only use if the angle is definitel) returns: :return: rotation matrix

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: :param array: array of points to rotate :param turns: number of turns to rotate by :param base_angle: angle to rotate by :param around: point to rotate around :param axis: axis to rotate around either as a combination of ['x','y','z'] or as a vector normal to the desired plane of rotation returns: :return: the rotated array

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 – the array of coordinates to rotate

Returns
a generator containing the 4 different rotations of the vertices

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 – the array of coordinates to rotate

Returns
a generator containing the 4 different rotations of the vertices

1.1.7 mrrvis.history module

defines a history object for storing a series of configurations

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

Bases: object

append(*state:* ConfigurationGraph)

Append a move to the history

Parameters
state – the state to add

revert(*n: int = 1*)

revert the history by n steps params: :param n: the number of steps to revert by returns: :return: The last state after the reversion

property t

The amount of items (time steps) in the history

1.1.8 mrrvis.move module

contains the Move class and its dependencies

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`

`bind(f: callable) → Checkwrapper`

Bind any function that takes a ConfigurationGraph and returns a boolean :param f: the function to bind :return: the checkwrapper after the bound function

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

`get()`

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 have some evaluation rule when called :param cases: the collection of cases to evaluate :param eval_rule: the rule governing the evaluation of this collision

`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 single collision case

`empty: ndarray`

Alias for field number 0

evaluate_case(*graph:* ConfigurationGraph) → bool

evaluate collision for this case :param graph: graph to check collision against :return: a bool showing true if no collision detected

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 :param turns: the number of times the base angle to rotate by (counter clockwise) :param base_angle: the angle per turn :param around: the coordinate to rotate around :param axis: the axis or normal vector to rotate around :type axis: str or np.ndarray 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

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

attempt to construct a new graph from the transaction :param transaction: the list of transformations to be performed in this move :return: the graph after this transaction is applied 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

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 :param candidate: the graph to be evaluated :return: candidate if feasible or None, if the move is infeasible

abstract generate_collision(*module*) → *Collision*

Generate the collision for a single transformation must be implemented in subclass remark. 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 must be implemented in subclass

recall, a task contains a module

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** – the module that is being affected

- **transformation** – the resulting location of the module(s) (can include replication)
- **collision** – the Collision object for this transformation

collision: *Collision*

Alias for field number 2

location: ndarray

Alias for field number 0

transformation: ndarray

Alias for field number 1

1.1.9 mrrvis.vistools module

core visualisation tools for visualising configurations provides the following functions:

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.**generate_voxels**(*vertices: ndarray*) → ndarray

generate voxel array for cube visualisation

Parameters
vertices – the vertices to be voxelised

Returns
the voxelised array

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

generator for hex patches from a list of two dimensional vertices :param vertices: the vertices to generate patches for :param style: any matplotlib kwargs for styling the patches :return: a generator containing all of the patches

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

convert an array of hexagon cubic coordinates to cartesian coordinates

Parameters
vertices – a set of hexagonal cubic coords expressed as a numpy array

Returns
an array of the converted vertices

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

Plot a configuration with matplotlib params: :param configuration: the graph to visualise :param show: whether to show the resulting graph :param save: whether to save the resulting graph :param filepath: the filepath for the graph when saving :param axes: whether to show the axes :param style: 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) params: :param history: the history to animate :param speed: the length of each frame in milliseconds :param show: whether to show the resulting graph :param save: whether to save the resulting graph :param filepath: the filepath for the graph when saving :param axes: whether to show the axes :param style: any matplotlib kwargs returns: :return: 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.

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** – the vertices to generate patches for
- **style** – any matplotlib kwargs for styling the patches

Returns
a generator containing all of the patches

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

generator for tri patches from a list of two dimensional vertices :param vertices: the vertices to generate patches for :param style: any matplotlib kwargs for styling the patches :return: a generator containing all of the patches

mrrvis.vistools.**tri_to_cart**(*vertices: ndarray*) → Tuple[ndarray, ndarray]

convert an array of Triangle cubic coordinates to cartesian coordinates

Parameters
vertices – a set of hexagonal cubic coords expressed as a numpy array

Returns
an array of the converted vertices

Returns
a vector of whether each vertex(row) of vertices corresponds to an upward pointing triangle

1.1.10 Module contents

Top-level package for mrrvis.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

F

G

H

I

L

M

N

P

R

S

T

V