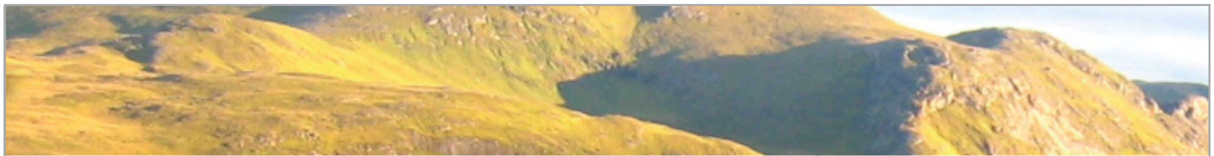

CellLab-CTS 2015 Users Manual

Created: August 2015, Greg Tucker

Last updated: August 2015

Introduction

CellLab-CTS is a Landlab module for building pairwise, continuous-time stochastic (CTS) cellular automata. Like other cellular automata, pairwise CTS models represent natural phenomena using a regular grid of cells; in the case of CellLab-CTS, the user can choose between square and hexagonal cells. Each cell has a given state, which is an integer code representing something about the nature or contents of that cell. Cells change state at random time intervals according to a user-defined transition rules. The transitions depend on the states of the neighboring cells, and in particular, of the states of each *pair* of adjacent cells. For example a pair with states 0 and 1 might undergo a transition to 1 and 1, or 3 and 0, etc.

This Users Manual provides instructions on how to write a model using CellLab-CTS, along with reference information about the classes and methods that CellLab-CTS provides. For further information about the theory, implementation, and design of CellLab-CTS, see Tucker et al. (2015 in prep). For background information on the theory of pairwise CTS models and example applications, see Narteau et al. (2001, 2009) and Rozier and Narteau (2014). For background on cellular automata in general, see Chopard and Droz (1998).

Note on terminology: In a CellLab-CTS model, the computational points—the objects that are normally called **cells** in a cellular automaton model—actually correspond with the **nodes** in a Landlab grid. Although Landlab grids also contain *cells*, which are defined as polygons that contain a node, Landlab grids do not have cells along the outmost ring of nodes around the grid. For example, a 4-row by 5-column Landlab raster grid has 20 nodes but only 6 cells (2 inner rows x 3 inner columns). For CellLab-CTS models, it is useful to include the perimeter nodes as “cells” for the purpose of handling boundary conditions. Therefore, CellLab-CTS treats all the **nodes** in the grid as if they were cells in a cellular automaton. This includes the perimeter nodes, for which Landlab does not formally define cells. For practical purposes, the distinction doesn’t make much difference, but it is important to understand that CellLab-CTS works with arrays of grid nodes rather than the (shorter) arrays of grid cells. Henceforth, to avoid confusion, we will refer to **nodes**, which you should read as being synonymous with the usual meaning of “cell” in a cellular automaton.

Prerequisites: This manual assumes working knowledge of the Python

Table Of Contents

programming language (any version), including basic familiarity with Python classes and objects. It also assumes a basic familiarity with Landlab grids. In addition, it will be helpful to have some familiarity with the Matplotlib and/or Pylab plotting libraries.

Writing a CellLab-CTS model

What is a CellLab-CTS model?

A CellLab-CTS model is a Python code that creates and initializes one of four types of `CellLabCTSModel` object, defines the possible cell states and transition rules, and calls the `run` method to execute the model. A CellLab-CTS model can be written in one of two basic ways. The first option is to write a simple Python script that imports the necessary ingredients. This approach is easy and versatile, and is recommended for first-time users, and/or those who are relatively unfamiliar with Python classes. The second option is to write your model as a subclass of one of the four existing CellLabModel subclasses (more on these below). The subclass approach is useful when you wish to use the *dynamic property updating* capability of CellLab-CTS—that is, for example, when you want to attach some form of additional data to the grid, and update the data at each transition event according to the state of the grid. In this manual, we will focus on the example of a simple script-based model.

Basic ingredients of a CellLab-CTS model

The basic steps in most CellLab-CTS models are as follows:

1. Import the necessary CTS classes (and any other necessary packages). These include: one of the four CellLab-CTS model classes (described below), the `Transition` class, and (optionally) the `CAPlotter` class for graphical display.
2. Create and initialize a Landlab `ModelGrid` object: either a `RasterModelGrid` or a `HexModelGrid`
3. Create a dictionary that defines the node states to be used
4. Create a list of Transition objects: one for each transition type in your model
5. Create an array with the initial state values in the grid
6. Instantiate and initialize one of the four CellLab-CTS classes
7. Set up plotting (if desired)
8. Run the model by calling the CellLab-CTS object's `run` method (perhaps pausing every so often to display the grid and/or write output to file)
9. Clean up

We will illustrate each of these steps using a simple example called `isotropic_turbulent_suspension.py`. This program simulates the random motion of neutrally buoyant sediment particles that are immersed in a turbulent fluid: think of tea leaves in a jar of tea that you are stirring with an invisible spoon. Each random motion is simulated by simply swapping a fluid state and a particle state.

Previous topic

Landlab Standard Names

Next topic

How do I set the boundary codes for the edges of a grid?

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

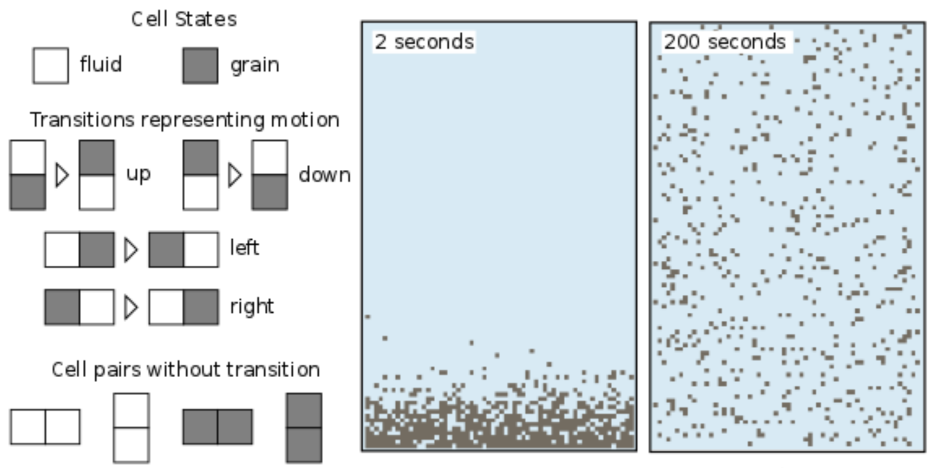

Figure 1: A CellLab-CTS model of suspended sediment particles in an isotropic turbulent fluid.

Before diving into the example, however, it's useful to look at the four different types of CellLab-CTS model.

Types of CellLab-CTS model

A CellLab-CTS grid can be either raster (regular grid of square cells) or hexagonal (triagonal grid of nodes with hexagonal cells). In addition, a CellLab-CTS model can be either *oriented* or *non-oriented*. An oriented model is one in which the spatial orientation of a node pair influences the types and/or of transition. For example, in an oriented raster model, a horizontal pair with states 0 and 1 might have a different set of potential transitions than a vertical pair with states 0 and 1. A non-oriented pair treats the sequence 0-1 the same regardless of whether the pair is vertical, horizontal, or (in the case of a hex grid) at any other angle.

With these different possibilities in mind, the four CellLab-CTS model types are:

1. `RasterCTS:` A non-oriented grid of square cells.
2. `OrientedRasterCTS:` An oriented grid of square cells, with two orientations (horizontal and vertical).
3. `HexCTS`: A non-oriented grid of hexagons.
4. `OrientedHexCTS:` An oriented grid of hexagons, with three orientations. These can be: (1) vertical, (2) +30 degrees from horizontal (angling down/left to up/right), and (3) -30 degrees from horizontal (angling up/left to down/right). Or, alternatively, the three axes can be horizontal and +/-30 degrees from vertical (one determines this when instantiating the grid object, as illustrated below).

These four types are implemented as subclasses of the base class `CellLabCTSModel`, as illustrated in Figure 2.

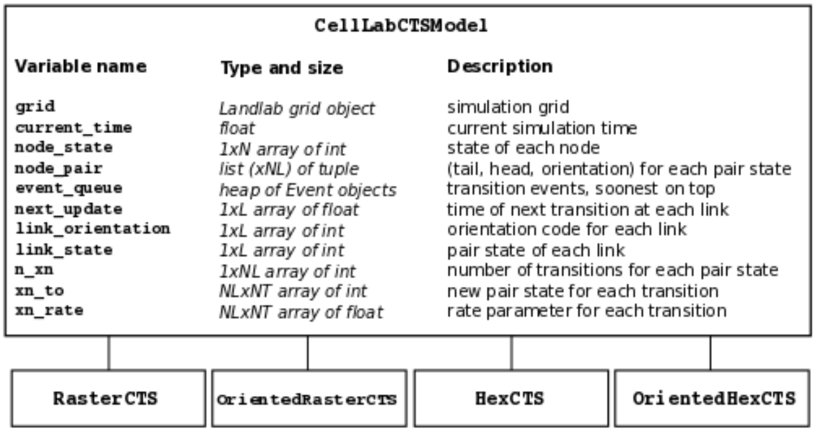

Figure 2: CellLab-CTS class hierarchy and main data structures. N = number of grid nodes, L = number of grid links, NL = number of possible link (node pair) states, NT = maximum number of transitions for any link state.

Step 1: Importing CellLab-CTS

A CellLab-CTS application normally starts by importing the appropriate type of CTS model class, along with any other packages needed. Thus, our suspended-sediment model starts out as follows:

```
#!/usr/env/python

"""
isotropic_turbulent_suspension.py

Example of a continuous-time, stochastic, pair-based cellular automaton model,
which simulates the diffusion of suspended, neutrally buoyant particles in a
turbulent fluid.

Written by Greg Tucker, February 2015
"""

import time
import matplotlib
from numpy import where
from landlab import RasterModelGrid
from landlab.components.cellular_automata.celllab_cts import Transition, CAPlotter
from landlab.components.cellular_automata.raster_cts import RasterCTS
```

Here, we're using a raster model, so we import Landlab's `RasterModelGrid` class. It will be a non-oriented raster model, so we import the `RasterCTS` class (rather than `OrientedRasterCTS`). We also import the `CAPlotter` class for help with graphical display (more on that below), as well as the `Transition` class. We need the Transition class to set up our pair transitions, which we explore next.

Setting up transitions

Sequence matters!

A particular pair state is described by the two node states, and optionally by the pair's orientation. A key thing to understand here is that any particular pair sequence, such as 0 and 1, is *different from the sequence in reverse*. The pair 0-1 is not the same as the pair 1-0! This is true for all four types of model. So then which is which? To answer this question, we first need to recall that each pair corresponds to the two ends of a *link* in the Landlab grid. A link is simply a directed line segment that connects two neighboring nodes. Every link has a *tail* and a *head* (like the head of an arrow); the direction of the link is from tail to head. The rule for CellLab-CTS pairs is that the first number refers to the tail of the corresponding link, and the second refers to its head. Thus, the pair state 0-1 means that the tail node has state 0 and the head node has state 1.

By default, the links in a raster grid always run from down to up (for vertical links) or left to right (for horizontal links) (Figure 3). For example, with a 0-1 pair in a raster grid, the 0 is either the left-hand node (if it's a horizontal pair) or the bottom node (if the pair is vertical). In a default hex grid, the links point either (1) upward, (2) angling right and up 30 degrees, or (3) angling right and down 30 degrees. (Note that you also have the option of switching the grid orientation so that one of the principal axes is horizontal instead of vertical; in that case, the three orientations are horizontal, 30 degrees clockwise from vertical, and 30 degrees counter-clockwise from vertical).

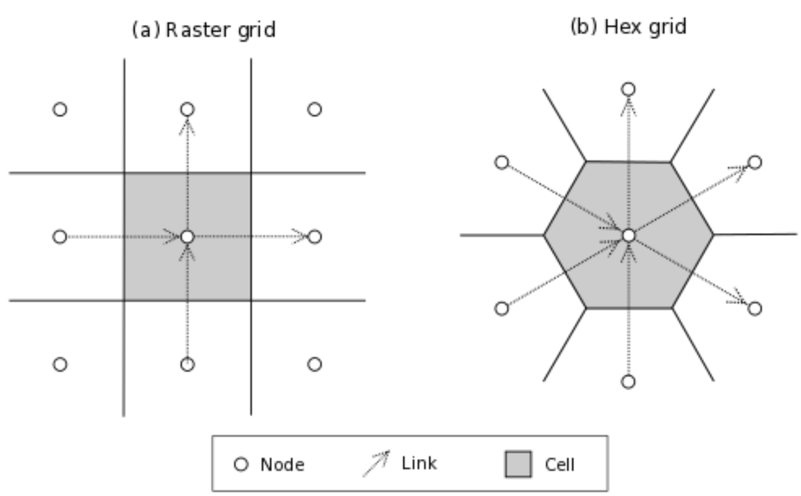

Figure 3: Illustration of nodes, links, and cells in a raster and hex grids. Note directions of links, which always "point" toward the upper-right hemisphere. The hex illustration shows a hex grid cell in vertical orientation; in horizontal orientation, links point rightward, up and right, and up and left.

How transitions are represented

Each transition type is described by the states of the tail and head nodes, and by the orientation of the pair. This information is encoded in a 3-element tuple. Recall that each pair is associated with a link. The first number is the state of the link's tail node, the second is the state of the link's head node, and the third is an *orientation code* that represents the pair's spatial orientation (Figure 4). In a non-oriented model, the orientation code is always zero. In an oriented raster, the orientation code is either 0 (horizontal) or 1 (vertical). For example, the code (0, 1, 0) in an oriented raster model would represent a vertical pair in which the left node has state 0 and the right state 1.

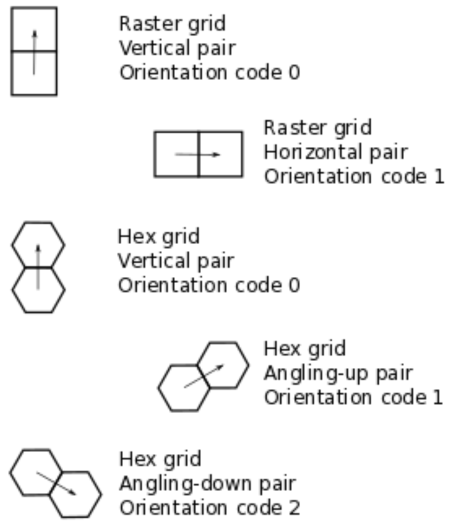

Figure 4: Pair orientation codes in a raster (top 2 panels) and vertical hex (bottom 3 panels) grid.

In an oriented hex, the orientation codes depend on the orientation of the grid itself. A Landlab `HexModelGrid` can be oriented such that one of the three principal axes is either horizontal (the default) or vertical. The choice is controlled by the optional keyword argument `orientation` (either `'vertical'` or `'horizontal'`) in the `HexModelGrid` initialization function. For a vertically aligned hex grid, the CellLab-CTS orientation codes are: 0 for vertical, 1 for right and upward, and 2 for right and downward (Figure 4). For example, the code (1, 0, 2) would represent a down-and-right pair, with a state of 1 in the upper-left node and 0 in the lower-right node. For a horizontally aligned hex grid, the CellLab-CTS orientation codes are: 0 for upward and left, 1 for upward and right, and 2 for right. For example, the code (1, 0, 2) would represent a left-to-right pair, with a state of 1 in the left node and 0 in the right node.

Example of a transition setup function

It can be helpful to put the transition setup procedure inside a function of its own. Here is the transition setup function for our turbulent suspension example (notice that the function itself has only four lines of code; all the rest is documentation):

```
def setup_transition_list():
        """
        Creates and returns a list of Transition() objects to represent state
        transitions for an unbiased random walk.

        Parameters
        ----------
```

```
    (none)

    Returns
    -------
    xn_list : list of Transition objects
            List of objects that encode information about the link-state transitions.

    Notes
    -----
    State 0 represents fluid and state 1 represents a particle (such as a
    sediment grain, tea leaf, or solute molecule).

    The states and transitions are as follows:

    Pair state      Transition to       Process             Rate (cells/s)
    ==========      =============       =======             ==============
    0 (0-0)         (none)              -                   -
    1 (0-1)         2 (1-0)             left/down motion    10.0
    2 (1-0)         1 (0-1)             right/up motion     10.0
    3 (1-1)         (none)              -                   -

    """

    # Create an empty transition list
    xn_list = []

    # Append two transitions to the list.
    # Note that the arguments to the Transition() object constructor are:
    #  - Tuple representing starting pair state
    #    (left/bottom cell, right/top cell, orientation)
    #  - Tuple representing new pair state
    #    (left/bottom cell, right/top cell, orientation)
    #  - Transition rate (cells per time step, in this case 1 sec)
    #  - Name for transition
    xn_list.append( Transition((0,1,0), (1,0,0), 10., 'left/down motion') )
    xn_list.append( Transition((1,0,0), (0,1,0), 10., 'right/up motion') )

    return xn_list
```

In this example, state 0 represents the fluid and state 1 represents a particle. Motion is represented by a transition from a 0-1 pair to a 1-0, or vice versa.

Your transition setup function should create and return a list of `Transition` objects. A Transition object contains (and is initialized with) the 3-element tuples for the starting and ending transitions, a transition rate (in units of cell-widths per time), and (optionally) a name. Two other optional parameters are used when you want to track properties associated with moving particles: a boolean flag (`swap_properties`) indicating whether the transition involves an exchange of properties, and the name of a user-defined callback function (`prop_update_fn`) to invoke whenever a transition of that type occurs.

(Note that it is also possible to specify a single-integer code for the link state, instead of 3-element tuple. This is a bit more of a headache, however, since it requires you to work out the link-state code corresponding to each pair, and is not recommended.)

Defining parameters

Typical parameters in a CellLab-CTS model, in addition to the transitions and rates, include the dimensions of the grid, the duration of the run, and the time intervals for plotting, writing output to file, and/or reporting progress on screen. In the following example, we have defined these within a `main()` function. They could also be read in from a file, input on a command line, or specified by some other method.

```
def main():

        # INITIALIZE

        # User-defined parameters
        nr = 80  # number of rows in grid
        nc = 50  # number of columns in grid
        plot_interval = 0.5   # time interval for plotting, sec
        run_duration = 20.0   # duration of run, sec
        report_interval = 10.0  # report interval, in real-time seconds

        # Remember the clock time, and calculate when we next want to report
        # progress.
        current_real_time = time.time()
        next_report = current_real_time + report_interval
```

Step 2: Creating a grid

Depending on the type of CTS model to be used, your code will need to instantiate either a `RasterModelGrid` or a `HexModelGrid`. If you wish to modify the default boundary setup, this should be done right after the grid is created. In the example below, we create a raster grid and set each of its four boundaries to act like a wall:

```
# Create grid
mg = RasterModelGrid(nr, nc, 1.0)

# Make the boundaries be walls
mg.set_closed_boundaries_at_grid_edges(True, True, True, True)
```

Step 3: Create a node-state dictionary

The possible node states are defined by creating entries in a dictionary, in which each key is an integer and each value is a string that gives the name for that state. There should be one entry for each state in your model. For example, our isotropic turbulent suspension model defines just two states:

```
ns_dict = { 0 : 'fluid', 1 : 'particle' }
```

Step 4: Create the transition list

If you've already defined a transition setup function, all you need to do here is call that function, as in the following example:

```
xn_list = setup_transition_list()
```

Step 5: Create an array containing the initial node-state values

The node state array should be a 1D numpy array of integers, with length equal to the number of grid rows times the number of grid columns. The easiest way to create such a grid is to use the grid's `add_zeros()` method (or, similarly, `add_ones` or `add_empty`). For example, for the suspended-sediment example we'll create an array of zeros, representing a container filled with fluid:

```
# Create the node-state array and attach it to the grid
node_state_grid = mg.add_zeros('node', 'node_state_map', dtype=int)
```

The first argument here is the name of the grid element to which values should be attached, the second is a name to give the array, and the third sets the data type to integer (instead of the default `float` type).

Depending on the nature of the model, the next step is to set the initial values of the node states. You can do this just as you would with any Landlab grid field. Remember that the coordinates of each node in a Landlab grid are available through the `node_x` and `node_y` arrays. For our working example, we'll set the lower 10% of nodes to state 1, indicating that we are starting with a pile of tea leaves at the bottom of the container:

```
# Initialize the node-state array: here, the initial condition is a pile of
# resting grains at the bottom of a container.
bottom_rows = where(mg.node_y<0.1*nr)[0]
node_state_grid[bottom_rows] = 1

# For visual display purposes, set all boundary nodes to fluid
node_state_grid[mg.closed_boundary_nodes] = 0
```

Note the use of the numpy `where` function, which we imported in Step 1.

Step 6: Instantiate a CellLab-CTS object

Our core model will be an object (a.k.a. *instance*) of one of the four CellLabCTS model classes. We create this just as we would any other Python object: by calling its constructor function, which is simply the name of the class followed by parentheses, with any necessary arguments within the parentheses. There are four required arguments: a grid object (which must be of the correct type, i.e., raster or hex), a dictionary of node states, a list of `Transition` objects, and the initial node state array. Here's what it looks like for our raster-based suspension model:

```
# Create the CA model
ca = RasterCTS(mg, ns_dict, xn_list, node_state_grid)
```

Step 7: Set up plotting

If you want to display your model's progress on screen, you can pause the run every once in a while and use pylab, matplotlib, or whatever your favorite graphics library may be to plot what's going on. For convenience, CellLab-CTS provides a `CAPlotter` class. CAPlotter is smart enough to find your node-state array, and to plot its contents in raster or hex form as appropriate. When you create the CAPlotter object, you pass it your CA model object and optionally a matplotlib colormap object. The CAPlotter has an `update_plot` method to plot the current state of your model, and a `finalize` method to clean up.

Here's an example of how to use a CAPlotter:

```
# Set up colors for plotting
grain = '#5F594D'
fluid = '#D0E4F2'
clist = [fluid,grain]
my_cmap = matplotlib.colors.ListedColormap(clist)

# Create a CAPlotter object for handling screen display
ca_plotter = CAPlotter(ca, cmap=my_cmap)

# Plot the initial grid
ca_plotter.update_plot()
```

Step 8: Run the model

Once a CTS model object has been instantiated, you run it forward in time with the `run` method. `run` takes one required argument: the future time to which to run. There are also three optional arguments:

- a node-state array (this is provided so that if you wish you can modify the array and re-run)
- a flag indicating whether to re-plot after each transition occurs
- a plotter object, which is required if the value of the flag is True

If you wish to pause occasionally to plot and/or write data to file, a natural approach is to place the call to the run method inside a loop, as in the following example:

```
# RUN
current_time = 0.0
while current_time < run_duration:

    # Once in a while, print out simulation real time to let the user
    # know that the sim is running ok
    current_real_time = time.time()
    if current_real_time >= next_report:
        print('Current simulation time '+str(current_time)+'  \
              ('+str(int(100*current_time/run_duration))+'%)')
        next_report = current_real_time + report_interval

    # Run the model forward in time until the next output step
    ca.run(current_time+plot_interval, ca.node_state,
           plot_each_transition=False)
    current_time += plot_interval

    # Plot the current grid
    ca_plotter.update_plot()
```

Step 9: Cleanup

There generally isn't much to clean up. If you are using a CAPlotter object, it can be helpful to call its `finalize` method, which turns off matplotlib's interactive mode and calls `show()` to make sure the plot is displayed on screen.

```
ca_plotter.finalize()
```

Reference information

Main data structures in the CellLabCTSModel class

Each of the four types of CTS model inherits from the base class (CellLabCTSModel) the following data structures. These are also illustrated in Figure 2.

`node_state` : 1d array (x number of nodes in grid)

  Node-based grid of node-state codes. This is the grid of cell (sic) states.

`node_pair` : list (x number of possible link states)

  List of 3-element tuples representing all the various link states. Allows you to look up the node states and orientation corresponding to a particular link-state ID.

`event_queue` : heap of Event objects

  Queue containing all future transition events, sorted by time of occurrence (from soonest to latest).

`next_update` : 1d array (x number of active links)

  Time (in the future) at which the link will undergo its next transition. You might notice that the update time for every scheduled transition is also stored in each Event object in the event queue. Why store it twice? Because a scheduled event might be invalidated after the event has been scheduled (because another transition has changed one of a link's two nodes, for example). The way to tell whether a scheduled event is still valid is to compare its time with the corresponding transition time in the *next_update* array. If they are different, the event is discarded.

`link_orientation` : 1d array of ints (x number of active links)

  Orientation code for each link.

`link_state` : 1d array of ints (x number of active links)

  State code for each link.

`n_xn` : 1d array of ints (x number of possible link states)

  Number of transitions ("xn" stands for "transition") from a given link state.

`xn_to` : 2d array of ints (# possible link states x max. # transitions)

  Stores the link-state code(s) to which a particular link state can transition. "max. # transitions" means the maximum number of transitions from a single state. For example, if each link state is associated with one and only one transition, then the maximum is 1, but if there is at least one link state that can have either of two different transitions, then the maximum would be two.

`xn_rate` : 2d array of floats (# possible link states x max. # transitions)

Rate associated with each link-state transition.

Methods and Internal Documentation for the base class: CellLabCTSModel

class **CellLabCTSModel**(*model_grid*, *node_state_dict*, *transition_list*, *initial_node_states*, *prop_data=None*, *prop_reset_value=None*) [source]

A CellLabCTSModel implements a link-type (or doublet-type) cellular automaton model. A link connects a pair of cells. Each cell has a state (represented by an integer code), and each link also has a state that is determined by the states of the cell pair.

assign_link_states_from_node_types() [source]

Assigns a link-state code for each link, and returns a list of these.

Takes lists/arrays of "tail" and "head" node IDs for each link, and a dictionary that associates pairs of node states (represented as a 3-element tuple, comprising the TAIL state, FROM state, and orientation) to link states.

create_link_state_dict_and_pair_list() [source]

Creates a dictionary that can be used as a lookup table to find out which link state corresponds to a particular pair of node states. The dictionary keys are 3-element tuples, each of which represents the state of the TAIL node, the HEAD node, and the orientation of the link. The values are integer codes representing the link state numbers.

(Performance note: making self.node_pair a tuple does not appear to change time to lookup values in update_node_states. Changing it to a 2D array of int actually slows it down.)

current_link_state(*link_id*) [source]

Used to determine whether the link state at link *link_id* has changed due to an independent change in the node-state grid. Returns the current state of the link based on the states of its two end nodes; this can be compared to the entry in self.link_state to determine whether the state has changed.

Parameters:	**link_id** : int ID of the active link to test
Returns:	**int** : New link state code

Notes

Vectorizing this might yield some speed.

---

do_transition(*event*, *current_time*, *plot_each_transition=False*, *plotter=None*) [source]

Implements a state transition.

Parameters:	**event** : Event object Event object containing the data for the current transition event **current_time** : float Current time in simulation **plot_each_transition** : bool (optional) True if caller wants to show a plot of the grid after this transition **plotter** : CAPlotter object Sent if caller wants a plot after this transition

---

get_next_event(*link*, *current_state*, *current_time*) [source]

Returns the next event for link with ID “link”, which is in state “current state”.

Parameters:	**link** : int

ID of the link

current_state : int

Current state code for the link

current_time : float

Current time in simulation (i.e., time of event just processed)

Returns:

Event object :

The returned Event object contains the time, link ID, and type of the next transition event at this link.

Notes

If there is only one potential transition out of the current state, a time for the transition is selected at random from an exponential distribution with rate parameter appropriate for this transition.

If there are more than one potential transitions, a transition time is chosen for each, and the smallest of these applied.

Assumes that there is at least one potential transition from the current state.

`push_transitions_to_event_queue()` [source]

Initializes the event queue by creating transition events for each cell pair that has one or more potential transitions and pushing these onto the queue. Also records scheduled transition times in the self.next_update array.

`run`(*run_duration*, *node_state_grid=None*, *plot_each_transition=False*, *plotter=None*) [source]

Runs the model forward for a specified period of time.

Parameters:

run_duration : float

Length of time to run

node_state_grid : 1D array of

ints (x number of nodes) (optional)

Node states (if given, replaces model's current node state grid)

plot_each_transition : bool (optional)

Option to display the grid after each transition

plotter : CAPlotter object (optional)

Needed if caller wants to plot after every transition

`set_node_state_grid`(*node_states*) [source]

Sets the grid of node-state codes to node_states. Also checks to make sure node_states is in the proper format, which is to say, it's a Numpy array of the same length as the number of nodes in the grid.

Parameters:	**node_states** : 1D array of ints (x number of nodes in grid)

Notes

The node-state array is attached to the grid as a field with the name 'node_state'.

`setup_array_of_orientation_codes`() [source]

Creates and configures an array that contain the orientation code for each active link (and corresponding cell pair).

Parameters:	**(none)** :
Returns:	**(none)** :

Notes

The setup varies depending on the type of LCA. The default is non-oriented, in which case we just have an array of zeros. Subclasses will override this method to handle lattices in which orientation matters (for example, vertical vs. horizontal in an OrientedRasterLCA).

setup_transition_data(*xn_list*)[source]

Using the transition list and the number of link states, creates three arrays that collectively contain data on state transitions:

> n_xn: for each link state, contains the number of transitions out of
>
> > that state.
>
> xn_to: 2D array that records, for each link state and each
>
> > transition, the new state into which the link transitions.
>
> xn_rate: 2D array that records, for each link state and each
>
> > transition, the rate (1/time) of the transition.
>
> xn_propswap: 2D array that indicates, for each link state and each
>
> > transition, whether that transition is accompanied by a "property" swap, in which the two cells exchange properties (in order to represent a particle moving)

update_component_data(*new_node_state_array*) [source]

Call this method to update all data held by the component, if, for example, another component or boundary conditions modify the node statuses outside the component between run steps.

This method updates all necessary properties, including both node and link states.

new_node_state_array is the updated list of node states, which must still all be compatible with the state list originally supplied to this component.

update_link_state(*link*, *new_link_state*, *current_time*) [source]

Implements a link transition by updating the current state of the link and (if appropriate) choosing the next transition event and pushing it on to the event queue.

Parameters:

link : int

ID of the link to update

new_link_state : int

Code for the new state

current_time : float

Current time in simulation

update_link_states_and_transitions(*current_time*) [source]

Following an "external" change to the node state grid, updates link states where necessary and creates any needed events.

update_node_states(*tail_node*, *head_node*, *new_link_state*) [source]

Updates the states of the two nodes in the given link.

Parameters:

tail_node : int

ID of the tail node of the link (cell pair) in question

head_node : int

ID of the head node of the link (cell pair) in question

new_link_state : int

Link state code for the new cell pair

Returns:

(bool, bool) :

Flags indicating whether the tail node and head node, respectively, have changed state

Methods and Internal Documentation for the RasterCTS class

class **RasterCTS**(*model_grid*, *node_state_dict*, *transition_list*, *initial_node_states*, *prop_data=None*, *prop_reset_value=None*) [source]

Class RasterLCA implements a non-oriented raster CellLab-CTS model.

Methods and Internal Documentation for the OrientedRasterCTS class

class **OrientedRasterCTS**(*model_grid*, *node_state_dict*, *transition_list*, *initial_node_states*, *prop_data=None*, *prop_reset_value=None*) [source]

Class OrientedRasterCTS implements an oriented raster CellLab-CTS model.

setup_array_of_orientation_codes() [source]

Creates and configures an array that contain the orientation code for each active link (and corresponding cell pair).

Parameters:	**(none)** :
Returns:	**(none)** :

Notes

This overrides the method of the same name in landlab_ca.py.

Methods and Internal Documentation for the HexCTS class

class **HexCTS**(*model_grid*, *node_state_dict*, *transition_list*, *initial_node_states*, *prop_data=None*, *prop_reset_value=None*) [source]

Class HexCTS implements a non-oriented hex-grid CellLab-CTS model.

Methods and Internal Documentation for the OrientedHexCTS class

class **OrientedHexCTS**(*model_grid*, *node_state_dict*, *transition_list*, *initial_node_states*, *prop_data=None*, *prop_reset_value=None*) [source]

Class OrientedHexCTS implements an oriented hex-grid CellLab-CTS model.

setup_array_of_orientation_codes() [source]

Creates and configures an array that contain the orientation code for each active link (and corresponding cell pair).

Parameters:	**(none)** :
Returns:	**(none)** :

Notes

This overrides the method of the same name in celllab_cts.py. If the hex grid is oriented such that one of the 3 axes is vertical (a 'vertical' grid), then the three orientations are:

> 0 = vertical (0 degrees clockwise from vertical) 1 = right and up (60 degrees clockwise from vertical) 2 = right and down (120 degrees clockwise from vertical)

If the grid is oriented with one principal axis horizontal ('horizontal' grid), then the orientations are:

> 0 = up and left (30 degrees counter-clockwise from vertical) 1 = up and right (30 degrees clockwise from vertical) 2 = horizontal (90 degrees clockwise from vertical)

Methods and Internal Documentation for class Transition

class **Transition**(*from_state*, *to_state*, *rate*, *name=None*, *swap_properties=False*, *prop_update_fn=None*) [source]

Represents a transition from one state (“from_state”) to another (“to_state”) at a link. The transition probability is represented by a rate parameter “rate”, with dimensions of 1/T. The probability distribution of time until the transition event occurs is exponentional with mean 1/rate. The optional name parameter allows the caller to assign a name to any given transition.

Note that from_state and to_state can now be either integer IDs for the standardised ordering of the link states (as before), or tuples explicitly describing the node state at each end, and the orientation. Orientation is 0: horizontal, L-R; 1: vertical, bottom-top. For such a tuple, order is (left/bottom, right/top, orientation).

Methods and Internal Documentation for class CAPlotter

class **CAPlotter**(*ca*, *cmap=None*) [source]

A CAPlotter is an object that handles display of a CellLab-CTS grid.

finalize() [source]

Wraps up plotting by switching off interactive model and showing the plot.

update_plot() [source]

Plots the current node state grid.

References

Chopard, B., & Droz, M. (1998). Cellular automata. Cambridge University Press, Cambridge, UK.

Narteau, C., Le Mouël, J. L., Poirier, J. P., Sepúlveda, E., & Shnirman, M. (2001). On a small-scale roughness of the core–mantle boundary. Earth and Planetary Science Letters, 191(1), 49-60.

Narteau, C., Zhang, D., Rozier, O., & Claudin, P. (2009). Setting the length and time scales of a cellular automaton dune model from the analysis of superimposed bed forms. Journal of Geophysical Research: Earth Surface (2003–2012), 114(F3).

Rozier, O., & Narteau, C. (2014). A real-space cellular automaton laboratory. Earth Surface Processes and Landforms, 39(1), 98-109.

Tucker, G.E., Hobley, D.E.J., Hutton, E., Gasparini, N.M., Istanbulluoglu, E., Adams, J.M., and Nudurupati, S.S. (in review) CellLab-CTS 2015: A Python library for continuous-time stochastic cellular automaton modeling using Landlab. Submitted to Geoscientific Model Development, September 2015.