## Appendix N.II - Quickstart guide

quickstart.md

# CAPyLE Quickstart Guide

Home

*This guide assumes that have installed the software as instructed in the installation guide*

## Running an example CA

1. Run CAPyLE (python main.py or run.[bat/sh])
2. Use the menu bar to select File -> Open. This will open in the folder `./ca_descriptions` .
3. Open one of the example files;
   - `wolframs_1d.py` is Wolfram's elementary 1D automata
   - `gol_2d.py` is Conway's 2D game of life
4. The main GUI elements will now load, feel free to customise the CA parameters on the left hand panel
5. Run the CA with your parameters by clicking the bottom left button 'Apply configuration & run CA'
6. The progress bar will appear as the CA is run
7. After the CA has been run, use the playback controls at the top and the slider at the bottom to run through the simulation.
8. You may save an image of the currently displayed output using the 'Take screenshot' button

To customise the CA please look at the CA Description Guide

# Appendix N.III - CA Descriptions guide

**cadescription.md**

# CA Descriptions Guide

Home

*This guide assumes a working installation of CAPyLE*

## Contents:

## The CAConfig Class

This class is available in `capyle.ca` and is used to transport CA parameters round the internal system. A whole instance of the class can be serialised using pickle and it simplifies dealing with 5+ Grid constructor arguments. Below are the variables that you may need to understand, if you wish to override the GUI within your description.

All functions and methods are accessed by using the . operator on an instance of CAConfig. Eg. `config.states = (0,1)`

| Variable | Type | Description | Example |
|---|---|---|---|
| title | str | the human readable title for your reference | `"Conway's game of life"` |
| dimensions | int | indicate a 1D or 2D CA | 2 |
| states | tuple of ints or floats | define which states are present in the model | `(0,1,2)` |
| num_generations | int | the number of timesteps to simulate | `100` |
| grid_dims | tuple | the dimensions of the grid | `(200,200)` |
| nhood_arr | list or numpy.ndarray | Integer or floating point array of the neighbourhood | `[[0,1,0],[1,1,1],[0,1,0]]` |
| initial_grid | np.ndarray | State of the grid at t = 0 | `np.zeros((200,200))` |
| state_colors | list | Default rgb color for each state (in order as defined in CAConfig.states) | `[(0,0,0),(0.2,0.3,1), (1, 0.2, 0.2)]` |
| rule_num* | int | The rule number for wolframs 1D CA | `30` |
| wrap | bool/float/int | The wrapping setting for whether you wish to wrap the grid around, pacman style. Setting this value to True => wrapping behaviour, False => surrounded by 'dead' state -100, int/float => surrounded by this value. You may supply your own value if you need to handle interactions with state -100 in your | `False` |

| | | | |
|---|---|---|---|
| | | model, otherwise just set the boolean value. This defaults to True. | |

**only applicable for 1D CA*

Instead of setting the variables directly, you may find it more useful to use the functions:

| Name | Usage |
|---|---|
| set_grid_dims | `set_grid_dims(dims=(200,200))` or `set_grid_dims(num_generations=100)` for 1D |
| set_initial_grid | `set_initial_grid(grid)` |

### Overriding the GUI

The GUI passes a CAConfig object to your CA description, filled with the values from the GUI. You can plug this directly into a Grid `X` D object, or alternatively you can override the variables in the CAObject before passing it to the Grid `X` D object to completely ignore the GUI. Setting these values will also mean that you can 'save' configurations (for example setting the values in the initial_grid so that they persist).

It is reccomended that you set these defaults in the `setup` function, as then when the CA description is initially loaded, your defaults will be shown in the GUI. You may also set them in `main` if you prefer, just as long as they are set before the GridXD constructor.

For an example, see the setup function guide below. For more detail, see the API documentation.

## CA descriptions location

The example descriptions are located in `[path_to_main_folder]/ca_descriptions/`. This is also where user defined descriptions should be created. These files are .py files that describe how a CA should be executed by the main software.

## Required in a CA description file

At the head of the file, it is critical to define where the module 'capyle' and its submodules are (by editing sys.path). This is automatically inserted on creation of a new description.

### Functions

The main body of the file must contain 3 functions with the following arguments:

1. `setup(args)`
   - The function executed to set up the config object
2. `transition_function(grid, neighbourstates, neighbourcounts)`
   - Define transition rules for the CA
   - To keep track of variables (for example a counter for each grid item) append these arguments to the function arguments (eg. to keep track of decaygrid `def transition_function(grid, neighbourstates, neighbourcounts, decaygrid):`. To see an example of this, see Extra variables as discussed later.)
3. `main()`
   - For main() to be executed when `python yourdescription.py` is executed the following must be placed at the *bottom* of the file, as is standard practice in python files.

   ```
   if __name__ == "__main__":
       main()
   ```

   - This function will only require editing if initialising variables to keep track of (eg. the initial state of `decaygrid` will be initialised in main. See Extra variables below for an example of this)

#### Setup

# CA Descriptions guide

The main program runs these description files in two stages:

- 'Pre-run' runs through only the `setup` function of the module and saves the CAConfig object to disk, which is then instantly loaded by the main software.
  - This pre-run is executed when the CA file is initialy 'opened' into the main GUI
  - During this pre-run 3 parameters *must* be set:

  ```
  config.title = "Your title here"  #Give the CA a name
  config.dimensions = 2             # Define as a 2D CA
  config.states = (0,1)             # States used are only 0 and 1
  ```

  - Additionally the other config parameters may also be set to static values as below:

  *Please note that as these additional parameters are being overridden at runtime, the GUI will no longer have an effect on the config parameters set by the user, in the description. To leave all parameters to be set by the gui, do not define any constants here*

  ```
  # Always run 100 generations
  config.num_generations = 100
  # Initialise the colours to be black for state 0 and
  # a slightly off-white for state 1
  config.state_colors = [(0,0,0), (0.8,0.8,0.8)]
  ```

- 'Full-run' runs the entire file all the way through, applying the transition function for every generation, and saving the output to a 'timeline' file, which is saved to disk along with the config.
  - Runs when user presses 'Run' in the GUI

### Transition function

As the transition function will be invoked every timestep, it is important that it is written with performance as the number one priority; only numpy array operations be used. Examples of these can be seen in `wolframs_1d.py` and `gol_2d.py`.

A minimum of three arguments must be passed to the transition function;

1. `grid` the current state of the grid
2. `neighbourstates` the states of each neighbour
3. `neighbourcounts` the number of neighbours of each state.

*The transition funciton must return the new state of the grid*

#### Example 1 - Using neighbourcounts

For example in a 2D CA (with states 0 and 1), if we wish to create a rule where the cell must be in state 1 and change to state 0 only if any 3 of its neighbours are in state 0 we would write:

```
cells_in_state_1 = (grid == 1) # cells that are currently in state 1
three_zero_neighbours = (neighbourcounts[0] == 3) # cells that have 3 neighbours in state 0
to_zero = cells_in_state_1 & three_zero_neighbours
# cells that are currently in state 1 and have exactly three neighbours in state 0
# are set to zero
grid[to_zero] = 0
```

The `cells_in_state_1` computation creates a boolean array where only where 1 exists on the grid is True. Another boolean array is computed where the `neighbourcounts` of state 0 is 3, so only those cells that have 3 surrounding neighbours currently in state 0 are set to True, and False otherwise. These two boolean arrays are then union-ed together so only the cells where True is in the same position in both arrays becomes True, this result is then stored in `to_zero`. This boolean array is then used as indices for which cells need to undergo the change (numpy ignores the 0 or False entries when a boolean array of matching shape is supplied as indices). So only those indices where *both* conditions of the rule are satisfied undergo the change.

#### Example 2 - Using neighbourstates

If you wish to create rules that require more precision than simply 'any x cells in the neighbourhood' and need to check the state of specific cells, this can be achieved through using the `neighbourstates` argument.

In a 2D CA the `neighbourstates` variable contains the states of the surrounding 8 cells in the order [NW, N, NE, W, E, SW, S, SE] (like reading the neighbourhood left to right).

For example in a 2D CA with states (0,1,2,3) To create a rule where if the cell is in state 3 *and* the corner neighbours are all in states above 1, then transition to state 1:

```
# unpack the state arrays
NW, N, NE, W, E, SW, S, SE = neighbourstates
in_state_3 = (grid == 3) # cells currently in state 3
all_corners_above_1 = (NW > 1) & (NE > 1) & (SW > 1) & (SE > 1) # corner states > 1
to_one = in_state_3 & all_corners_above_1 # union the results
grid[to_one] = 1
```

Of course you are not limited to using one or the other, feel free to mix and match if it is required.

#### Important notes

- You may wish to experiment creating and manipulating numpy arrays in this manner in a separate file before writing your transition function.
- It is recommended that you perform all of the rule calculations, before performing the rule applications. For example if I wished to perform both of the rules above, I would calculate both `to_zero` and `to_one` first, then apply the rules using `grid[to_zero] = 0; grid[to_one] = 1` then return the grid with `return grid`. This ensures that you are applying your rules on the current time step t.
    - You may instead make a copy of the initial grid and apply your transitions to the copy on the fly, then return it, but ensure to make a deep copy with np.copy()
- You might like to consider that you 'clear' your grid before applying any changes to avoid mistakes. For example `grid[:, :] = 0` then `grid[to_one] = 1`, to set everything to 0 then only those which have been calculated to 1.

#### Extra variables

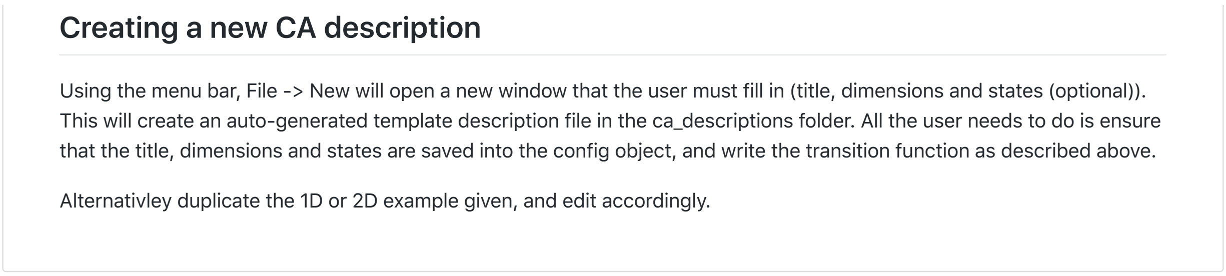
It is common to keep track of and manipulate variables during runtime, like staying in one state for a set number of generations, and then changing state after these generations have passed. To achieve this, add these extra variables to the arguments of your transition function, and instead of passing your transition_function to the Grid constructor, pass a tuple of your transition function and these extra variables eg. `(transition_function, extravar1, extravar2,...)`. See the example below:

After 2 timesteps cells in state 1 decay to state 0 in a 2D CA with states (0,1):

```
...
def transition_function(grid, neighbourstates, neighbourcounts, decaygrid):
  cells_in_state_1 = (grid == 1) # cells in state 1
  decaygrid[cells_in_state_1] -= 1 # take one off their decay value
  # the above line *must* save any changes to the decaygrid else they will not persist
  # to the next timestep

  decayed_to_zero = (decaygrid == 0) # find those which have decayed to 0
  grid[decayed_to_zero] = 0 # switch their state to 0
  return grid
...
def main():
  ...
  # initialise the decay grid
  decaygrid = np.zeros(config.grid_dims)
  decaygrid.fill(2)
  grid = Grid2D(config, (transition_function, decaygrid))
  # the usual constructor would be Grid2D(config, transition_function)
  timeline = grid.run()
  ...
...
```

## CA Descriptions guide



### Creating a new CA description

Using the menu bar, File -> New will open a new window that the user must fill in (title, dimensions and states (optional)). This will create an auto-generated template description file in the ca_descriptions folder. All the user needs to do is ensure that the title, dimensions and states are saved into the config object, and write the transition function as described above.

Alternativley duplicate the 1D or 2D example given, and edit accordingly.

# Appendix N.IV - Troubleshooting guide

troubleshooting.md

## Troubleshooting guide

### My CA is really slow

Every step has in the internal system has been taken to ensure that it runs things as fast as possible. If your CA is very slow, then the cause will be your transition function. Refer to the CA description guide for specifics, but you must try to create your transition rules using Numpy array operations, as these operations are compiled C and run in parallel if possible, resulting in them being exceptionally fast.

- Some for loops are acceptable as there may be occasions when it is computationally quicker (or equal) to perform a low step count for loop than prepare an array for a numpy operation.
- The above holds for nested for loops, but these must be used with extreme care
- *Never* traverse the grid using a for row in grid, for cell in row like operation. There will be alternative, but perhaps less obvious solutions.
- If possible put any intensive or repetitive processing in the `main` function, and pass the part or full pre-processed information to the transition function. Even if it is a very simple operation, remember that your transition function will be called every iteration and will have an impact.

If you have taken the steps above, try reducing your grid dimensions or generations to speed up the simulation.

### My GUI parameters are not doing anything

Check the CA description you are trying to run, the parameter will be being overwritten in there (more than likely in the `setup` function.

### How do I save my configuration?

Set the values in your CA description file before the Grid object is created (for example in `setup`). For example:

- To always use 200 by 200 grid:

```
config.grid_dims = 200, 200
```

- To set default state colours for my three states:

```
...
config.states = (0,1,2)
...
config.state_colors = [(0.5, 0, 0.5), (0, 0.5, 0.63), (1, 0, 0)]
```

- To set an initial grid (this example fills with state 0 then fills central 5x5 square with state 1):

```
config.initial_grid = np.zeros(config.grid_dims)            # zero grid
halfr, halfc = config.grid_dims[0]//2, config.grid_dims[1]//2   # calc central square indices
config.initial_grid[halfr:halfr+5, halfc:halfc+5] = 1          # fill square with state 1
```

- etc...

### The state colours are the same

You can set default colours for each state, see the example above. Alternatively you can set these in the GUI.

### It crashed

- If using macOS, and you scrolled on the window, this will make it crash as the native scrolling intertia is not handled at all by the GUI toolkit tkinter. Unfortunatley there was nothing to be done to fix this, so just refrain from scrolling! (There is no reason to scroll in this software)
- Bugs in your code should not cause the main GUI to crash, but if it does crash, check your CA description thoroughly for what could have caused it to terminate. Otherwise please report what you were doing to cause the crash to a demonstrator.

### I don't like this software at all

Sorry about that, please report why to the feedback form here (https://goo.gl/H6SW2q).

## Possible FAQs

### What is Numpy?

You should have used this library in COM2004 Data driven computing, if not, please familiarise your self using the Numpy documentation and simple examples.

### Can I add feature x so that it does y?

Go right ahead, but at your own risk. Make sure to save your CA descriptions somewhere safe. If you have a really good feature, create a github pull request and we will look into adding it to the main tool.

### Can I provide some feedback?

Please ( please ) do! Any feedback is welcomed, positive, negative or neutral. Use this link (https://goo.gl/H6SW2q) to fill in the google form.

### Why are the code lines so short?

The python programming standard PEP8 requires the line length to be 79 characters or less. It is recommended to follow this standard where possible.