Self-Organized Criticality Simulation (SocSim)

Feb 28, 2020

Readme file content

CHAPTER 1

Self-Organized Criticality Simulation (SocSim)

FACULTY

of Physics. University of Warsaw

Documentation Status Build Status codecov

Project is created as part of subject: Team student projects Faculty of Physics

Programs in Python that simulate dynamical systems that have a critical point as an attractor. So called **self-organized criticality**(SOC)

1.1 1. Theoretical problem description

Self-organized criticality wiki:

> In physics, self-organized criticality (SOC) is a property of **dynamical systems** that have a **critical point** as an attractor. Their macroscopic behavior thus displays the spatial or temporal scale-invariance characteristic of the critical point of a phase transition, but without the need to tune control parameters to a precise value, because the system, effectively, tunes itself as it evolves towards criticality.
>
> The concept was put forward by Per Bak, Chao Tang and Kurt Wiesenfeld ("BTW") in a paper published in 1987 in Physical Review Letters, and is considered to be one of the mechanisms by which complexity arises in nature. Its concepts have been enthusiastically applied across fields as diverse as geophysics, physical cosmology, evolutionary biology and ecology, bio-inspired computing and optimization (mathematics), economics, quantum gravity, sociology, solar physics, plasma physics, neurobiology and others.
>
> SOC is typically observed in slowly driven non-equilibrium systems with a large number of degrees of freedom and strongly nonlinear dynamics. Many individual examples have been identified since BTW's

original paper, but to date there is no known set of general characteristics that guarantee a system will display SOC.

1.2 2. Program structure, installation and use cases

1.2.1 2.1 Project folder structure

Project folder structure is inspired by these sources: sources1 source2 and Kwant project.

socsim:

- **docsrc** - holds Sphinx scripts used for documentation generation.
- **docs** - GitHub configuration folder, which holds web-page of project.
- **resource** - Non executable files.
- **results** - folder used for holding results of simulation, *Jupiter* notebooks and different use cases.
- **SOC** - main project folder, which holds all source code.
  - **models** - contains different SOC models, like: Abelian sandpile model, forest-fire model, etc..
  - **common** - common code between all models
  - **tests** - unit tests of code

1.2.2 2.2 Installation and dependencies

Dependencies

Mostly numerical libraries, visualsation, web page generation etc. For whole list take a look at requirements.txt

1.2.3 2.3 Use cases

Running program

Program is designed in next way:

- Framework part - placed under `SOC` folder.
- Research part - consists of jupyter notebooks(which can be easily deployed to web-page) and is placed under `research folder`

Developing the program

use `python setup.py develop` to install a basic set of dependencies and link the package to be importable in your current Python environment.

Running test cases

To make folder SOC an import package, run only once:

```
python setup.py develop
```

After that, simply use `pytest SOC` to automatically find and execute all existing test cases.

Web-page generation

Web page is generated using Sphinx library.

Under terminal enter into `docsrc` folder and type:

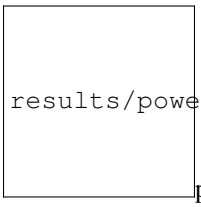
```
make html
```

Web-page will be generated into `./docsrc/build/html` folder. If you want to update web-page, copy generated web-page into `/docs` folder.

1.3 Results

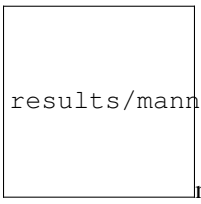
Click the model's name for more examples:

1.3.1 BTW



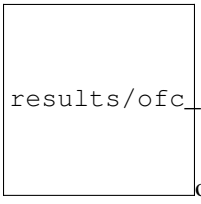
power-law

1.3.2 Manna model(Abelian/non-Abelian)



mannaexample

1.3.3 OFC



ofcexample

1.3.4 Forest fire

1.4 Summary of intial goals:

- Make wide coverage of all self-organized criticality models.
- Figuring out the best algorithms.
  - Easy scaling on many processors machine(multithreading, GPGPU, CUDA, numba).
- Using some best practice of programming:
  - Coding conventions
  - Unit Tests.
  - Creation of common modules.
  - Automatic documentation generation.
  - Readability of code and easy of use(between clarity and speed, we should choose clarity).

1.4.1 0.2 Commitments

Here are described code formatting style and other conventions, to make code more *uniform*. Also this section is for newcomers and contributors.

Unit Tests

SocSim uses the lovely PyTest for its unit testing needs. Tests are run automatically on every commit using TravisCI.

Documentation

Most popular documentation generator for Python - Sphinx. Good tutorial about using Sphinx here. Here is example of good Google style docstring standardized by PEP-484.

```
pip install -U sphinx
pip install sphinx_rtd_theme
pip install nbsphinx
```

pandoc

Dependencies of sphinx: `recommonmark`.

1.5 What's next?

- How we can apply Keras? Predictions, finding hidden parameters, etc.
- More tests for the batch processes running (Dask)
- Convenient selection of different boundary conditions of a system (lattice)
- Parametrization of the earthquake model for the coverage of wider selection of submodels (by including eg.: drive with random loading, toppling to the neighbours in a specific state (crack model), delay of the fracture initiation, threshold for the fracture propagation) like in Lomnitz-Adler (1993)

- Database of the simulations

1.6 3. Links/References

- Bak, P., Tang, C. and Wiesenfeld, K. (1987). "Self-organized criticality: an explanation of 1/f noise". Physical Review Letters. 59 (4): 381–384. Bibcode:1987PhRvL..59..381B. doi:10.1103/PhysRevLett.59.381. PMID 10035754. Papercore summary: http://papercore.org/Bak1987.
- Abelian sandpile model
- Forest-fire model
- Theoretical Models of Self-Organized Criticality (SOC) Systems
- Pink noise
- Introduction to Self-Organized Criticality & Earthquakes
- 25 Years of Self-Organized Criticality: Solar and Astrophysics
- SOC computer simulations
  - Studies in self-organized criticality
- Theoretical Models of SOC Systems

CHAPTER 2

Manna model

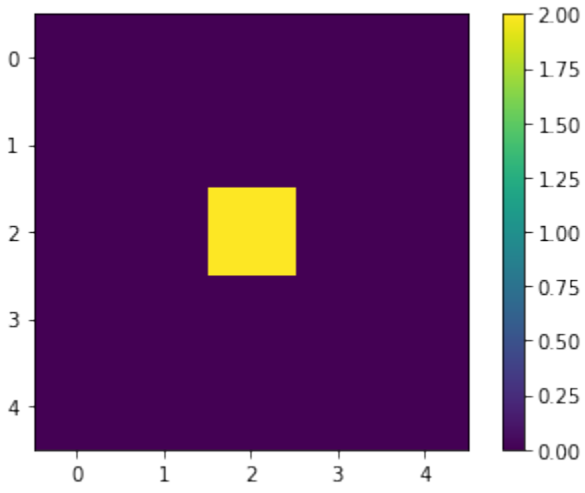
The Manna model is similar in concept to the BTW model. However, where BTW dissipates its "sand grains" deterministically, the Manna model introduces some **randomness**. Let's take a look:

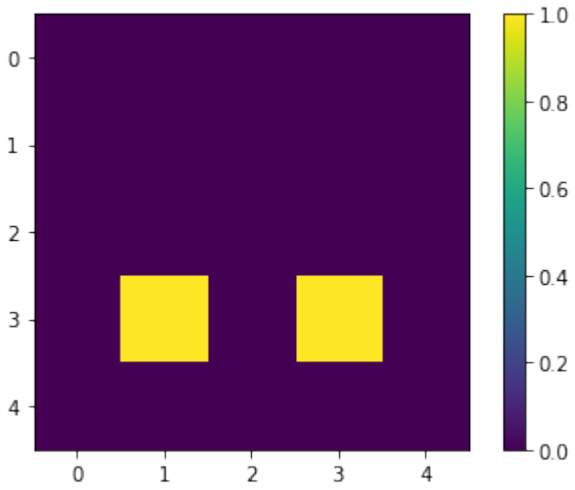
[3]:
```
from SOC import Manna
model = Manna(L=3, save_every=1)
model.critical_value
```

[3]:
```
1
```

This means that the model begins toppling its "sandpiles" once we put two grains somewhere. Let's try that:

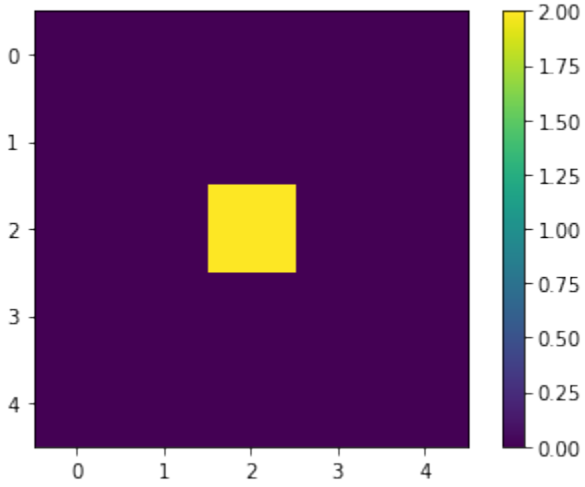
[20]:
```
model = Manna(L=3, save_every=1)
model.values[2,2] = 2
model.plot_state(with_boundaries=True);
model.AvalancheLoop()
model.plot_state(with_boundaries=True);
```

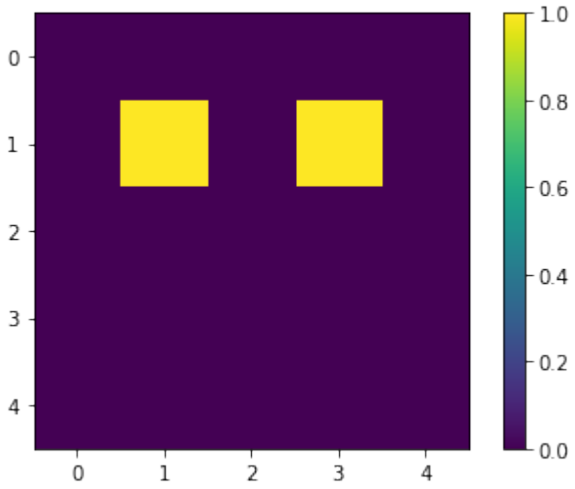




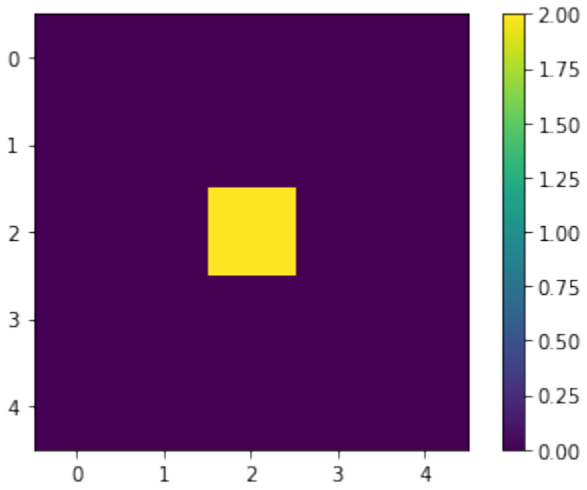
Why these two target locations in particular? It actually is random! Let's rerun that:

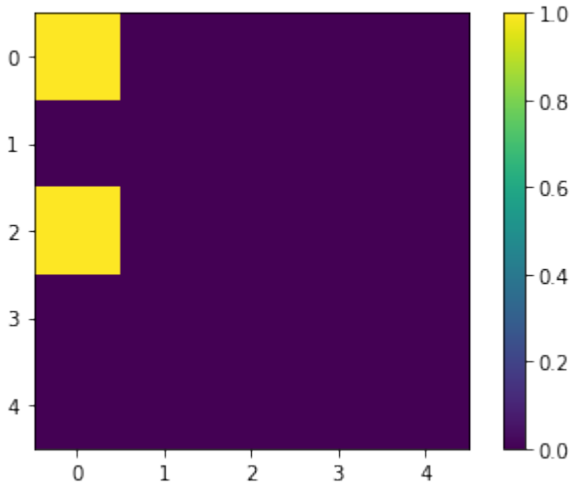
[21]:
```
model = Manna(L=3, save_every=1)
model.values[2,2] = 2
model.plot_state(with_boundaries=True);
model.AvalancheLoop()
model.plot_state(with_boundaries=True);
```





[22]:
```
model = Manna(L=3, save_every=1)
model.values[2,2] = 2
model.plot_state(with_boundaries=True);
model.AvalancheLoop()
model.plot_state(with_boundaries=True);
```





Oh, that's a bit weird, isn't it? These seem to have moved awfully far. The trick is that the two grains that fall from the toppling location pick their location at random **independently**, and here they both picked (1, 1) at first.

Let's run it for some more time:

[25]:
```
model = Manna(L=5, save_every=1)
model.run(1000)
model.animate_states(notebook=True)
```

```
Waiting for wait_for_n_iters=10 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=1010.0), HTML(value='')))
```

```

```

```
<IPython.core.display.HTML object>
```

Let's run a larger simulation instead:

[37]:
```
model = Manna(L=10, save_every=1)
model.run(1000)
model.animate_states(notebook=True)
```

```
Waiting for wait_for_n_iters=10 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=1010.0), HTML(value='')))
```

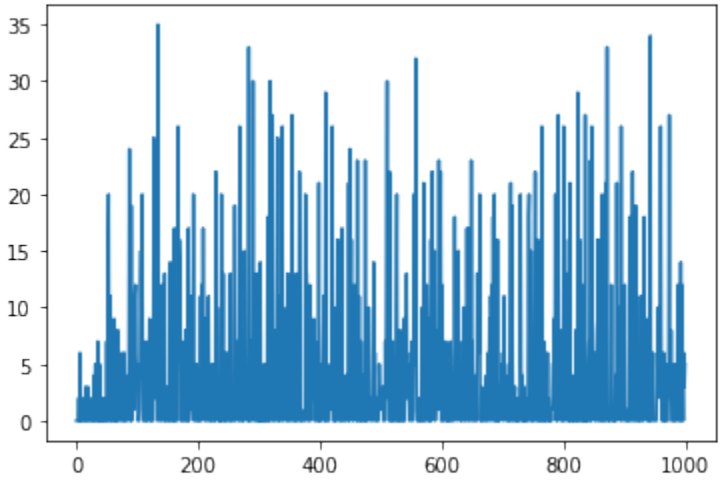
```

```

```
<IPython.core.display.HTML object>
```

If you look closely, you'll see that the system begins to exhibit very large avalanches very soon:

[38]:
```
model.data_df.AvalancheSize.plot()
```

[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2828870050>
```



Let's take a look at how modifying the critical value affects the simulation. We'll do some more iterations, so the system has the opportunity to "fill up" better. We'll also skip some animation frames.

[47]:
```
model = Manna(L=10, critical_value=4, save_every=10)
model.run(5000, wait_for_n_iters = 1000)
model.animate_states(notebook=True)
```

```
Waiting for wait_for_n_iters=1000 iterations before collecting data. This should let␣
↪the system thermalize.
```

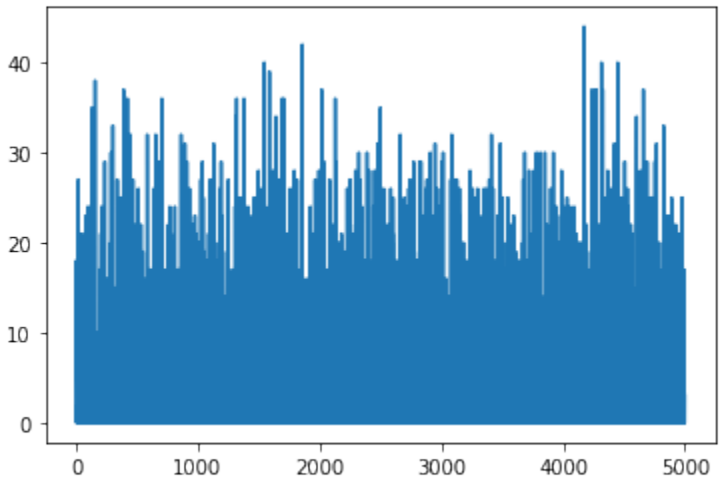
```
HBox(children=(FloatProgress(value=0.0, max=6000.0), HTML(value='')))
```

```

```

```
<IPython.core.display.HTML object>
```

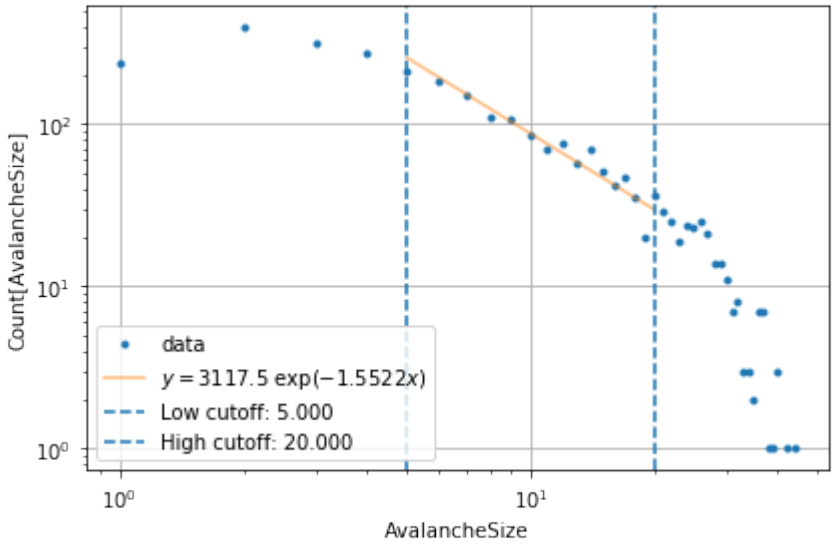
[48]:
```
model.data_df.AvalancheSize.plot()
```

[48]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f281fc7fb90>
```



Note how the avalanche size grows until a certain period, and then starts to fluctuate randomly at pretty large values. We can try to investigate the histogram of those avalanche sizes. We'll also fit a line to the linear segment (picked purely subjectively, visually and heuristically).

[53]:
```
model.get_exponent(low=5, high=20)
```




```
y = 3117.481 exp(-1.5522 x)
```

[53]:
```
{'exponent': -1.552219845297347, 'intercept': 3.4938038036304393}
```

One thing for sure, there is a region where the scaling in log-log scale is linear. The line fits rather well.

Let's run a larger simulation and try to estimate the scaling exponent. We'll wait for a good while so that the system can thermalize well:

[61]:
```
model = Manna(L=40, save_every=100)
model.run(100000, wait_for_n_iters=50000)
```

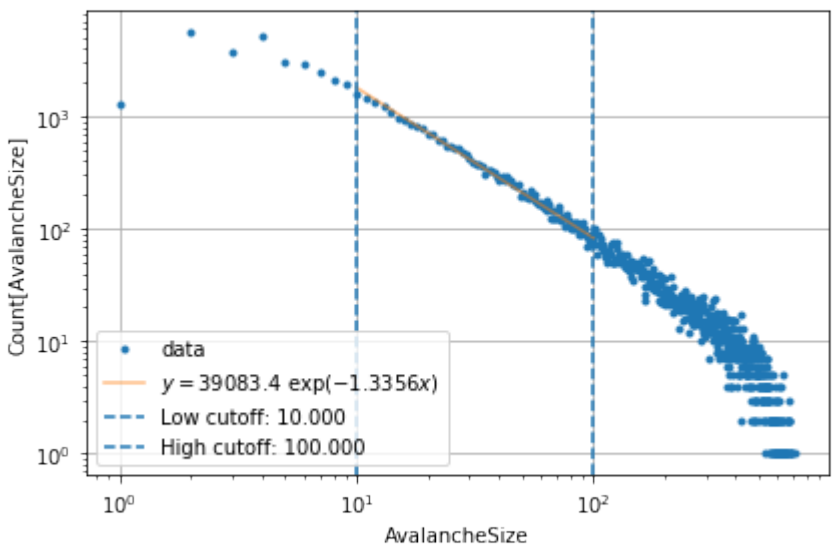
```
Waiting for wait_for_n_iters=50000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=150000.0), HTML(value='')))
```

[62]:
```
model.animate_states(notebook=True)
```

```
<IPython.core.display.HTML object>
```

[63]:
```
model.get_exponent(low = 10, high=100)
```




```
y = 39083.385 exp(-1.3356 x)
```

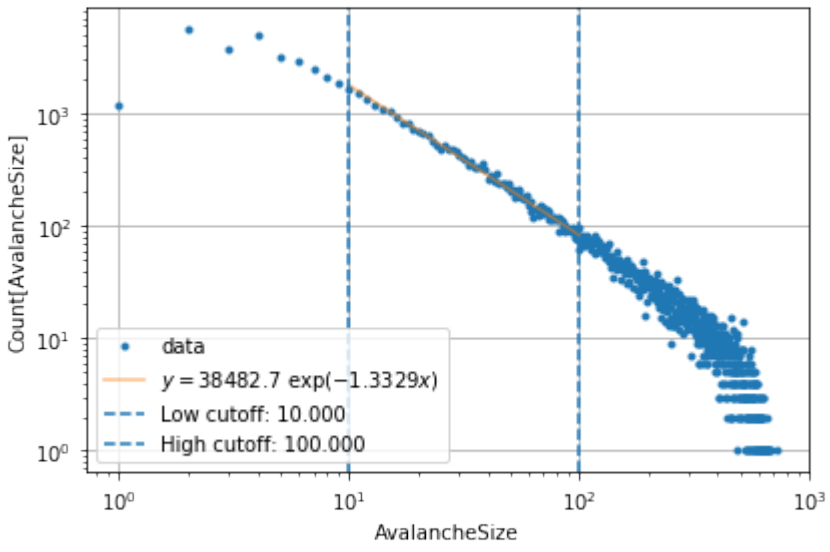
[63]:
```
{'exponent': -1.3356064470598588, 'intercept': 4.591992174363189}
```

Let's see how reproducible this is:

[65]:
```
model = Manna(L=40, save_every=10000)
model.run(100000, wait_for_n_iters=50000)
model.get_exponent(low = 10, high=100)
```

```
Waiting for wait_for_n_iters=50000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=150000.0), HTML(value='')))
```



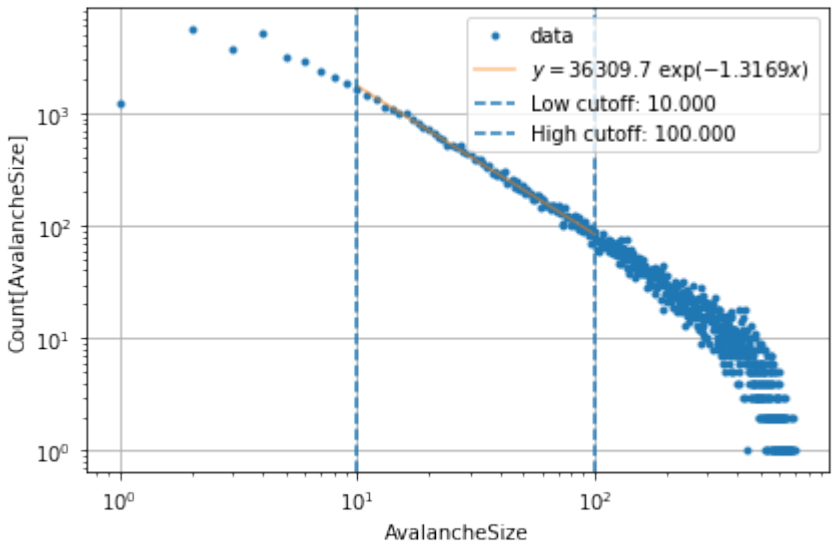

```
y = 38482.710 exp(-1.3329 x)
```

[65]:
```
{'exponent': -1.3328829112845815, 'intercept': 4.585265642606212}
```

[66]:
```
model = Manna(L=40, save_every=10000)
model.run(100000, wait_for_n_iters=50000)
model.get_exponent(low = 10, high=100)
```

```
Waiting for wait_for_n_iters=50000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=150000.0), HTML(value='')))
```



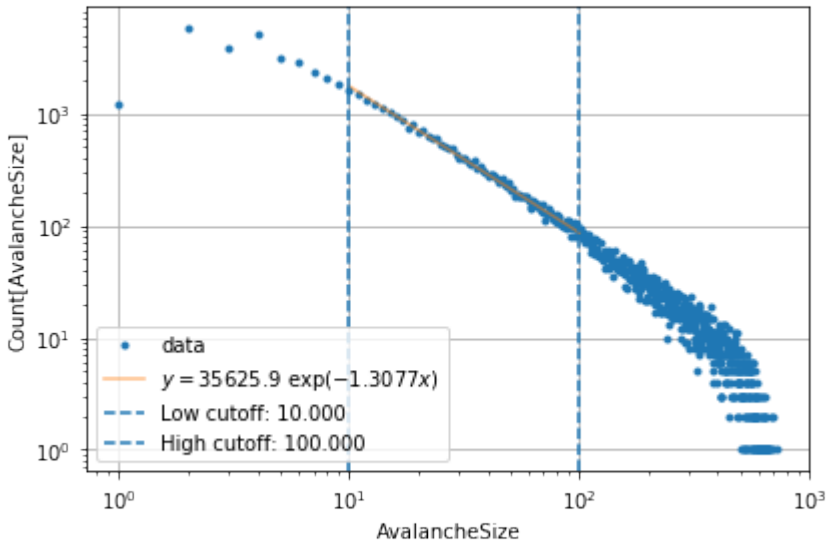

```
y = 36309.665 exp(-1.3169 x)
```

[66]:
```
{'exponent': -1.316892681945754, 'intercept': 4.5600222427865065}
```

[67]:
```
model = Manna(L=40, save_every=10000)
model.run(100000, wait_for_n_iters=50000)
model.get_exponent(low = 10, high=100)
```

```
Waiting for wait_for_n_iters=50000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=150000.0), HTML(value='')))
```




```
y = 35625.884 exp(-1.3077 x)
```

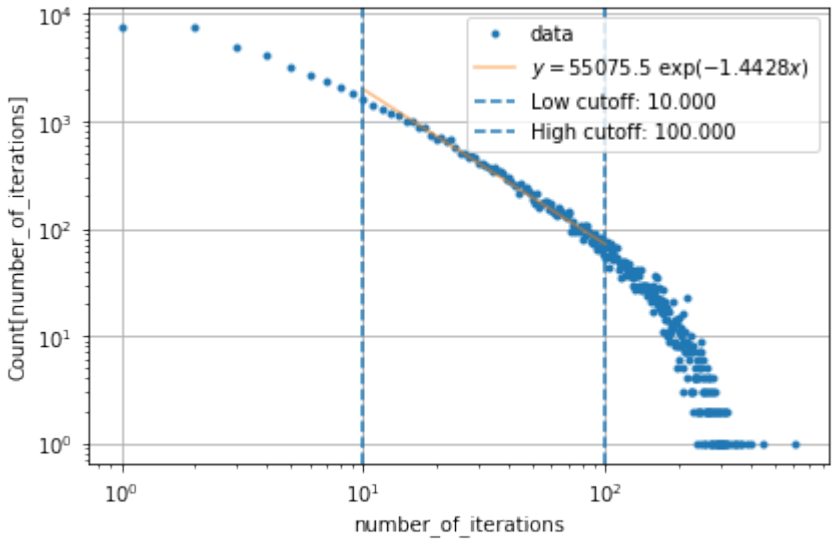
[67]:
```
{'exponent': -1.3077133473189786, 'intercept': 4.551765652488615}
```

And the exponent for this model was reported by others (see Table 6.1 in Pruessner) to be around 1.25-1.30.

Which is pretty darn close!

There is another quantity we could calculate here, the number of iterations for an avalanche to finish:

[68]:
```
model.get_exponent("number_of_iterations", low = 10, high=100)
```




```
y = 55075.511 exp(-1.4428 x)
```

[68]:
```
{'exponent': -1.4428414522987587, 'intercept': 4.7409585345785565}
```

CHAPTER 3

Olami–Feder–Christensen (OFC) Earthquake Model

Olami Z., Feder H., Christensen K., Self-organized criticality in a continuous, nonconservative cellular automaton modeling earthquakes, Phys. Rev. Lett. 68, 1992, https://doi.org/10.1103/PhysRevLett.68.1244

modified as in eqns (1) from: Grassberger P., 1994. Efficient large-scale simulations of a uniformly driven system, Phys. Rev. E, 49, 2436–2444, https://doi.org/10.1103/PhysRevE.49.2436

[2]:
```
from SOC.models import OFC
```

[3]:
```
sim0 = OFC(conservation_lvl=0.2, L=30, save_every = 1)
sim0.run(1000, wait_for_n_iters=1000)
```

```
Waiting for wait_for_n_iters=1000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=2000.0), HTML(value='')))
```

[4]:
```
sim0.animate_states(notebook=True)
```

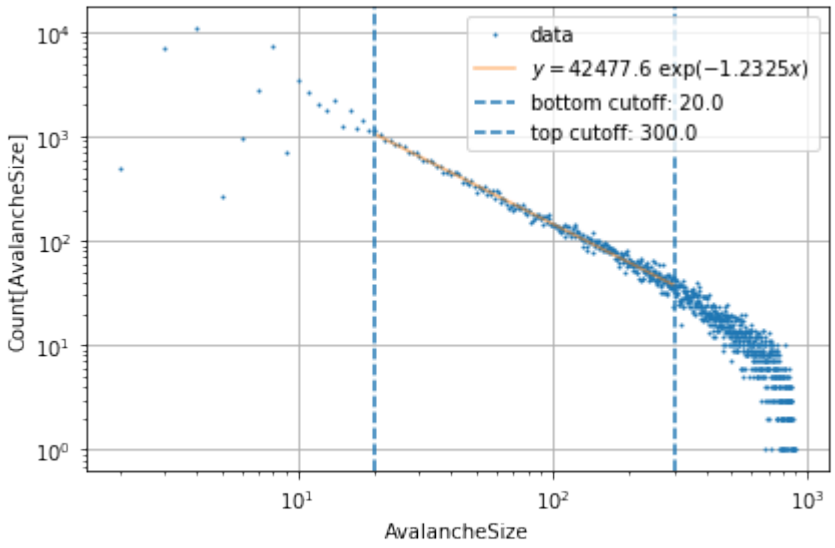
```
<IPython.core.display.HTML object>
```

[17]:
```
sim1 = OFC(L=30)
sim1.run(100000, wait_for_n_iters = 10000)
```

```
Waiting for wait_for_n_iters=10000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=110000.0), HTML(value='')))
```

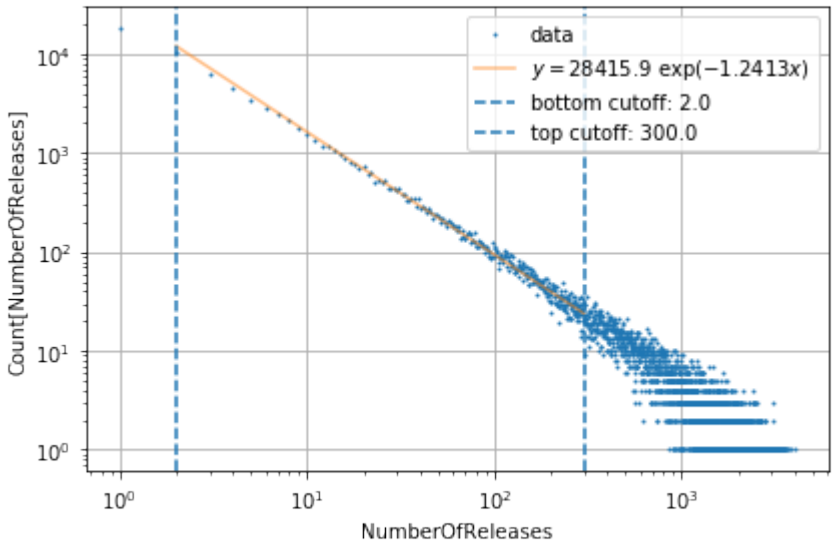
[22]:
```
sim1.get_exponent(low=20, high=300)
```




```
y = 42477.635 exp(-1.2325 x)
```

[22]:
```
{'exponent': -1.232457941524679, 'intercept': 4.628160328647776}
```

[23]:
```
sim1.get_exponent(column='NumberOfReleases', low=2, high=300)
```




```
y = 28415.887 exp(-1.2413 x)
```

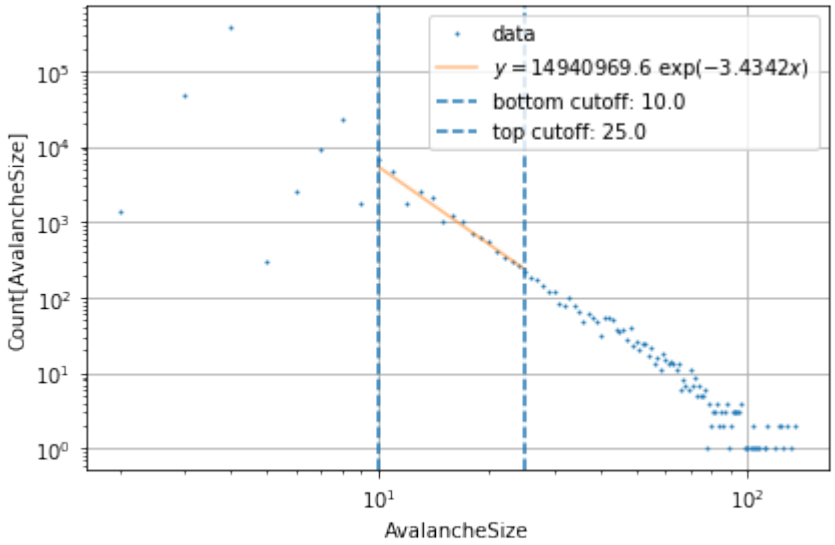
[23]:
```
{'exponent': -1.2412823768665933, 'intercept': 4.453561223266139}
```

[8]:
```
sim2 = OFC(conservation_lvl=0.1, L=30)
sim2.run(500000, wait_for_n_iters = 10000)
```

```
Waiting for wait_for_n_iters=10000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=510000.0), HTML(value='')))
```

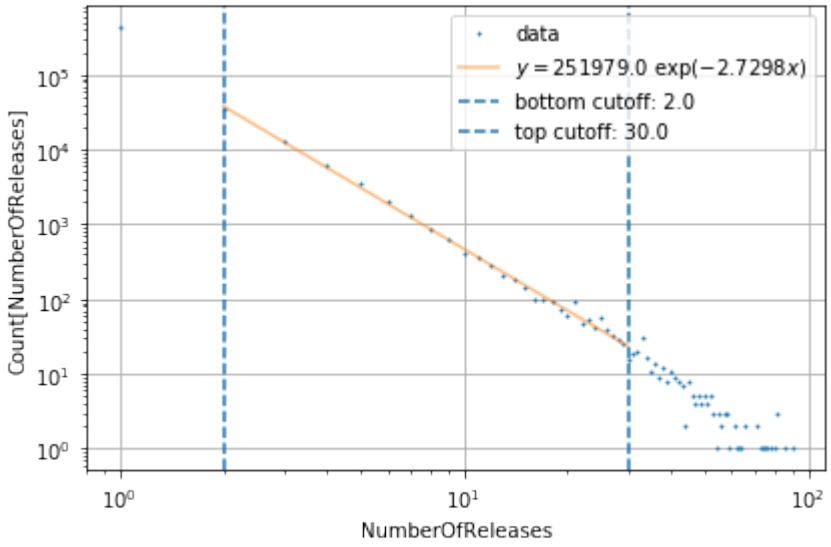
[9]:
```
sim2.get_exponent(low=10, high=40)
```




```
y = 14940969.613 exp(-3.4342 x)
```

[9]:
```
{'exponent': -3.4341725493293325, 'intercept': 7.1743787824846805}
```

[16]:
```
sim2.get_exponent(column='NumberOfReleases', low=2, high=30)
```




```
y = 251979.007 exp(-2.7298 x)
```

[16]:
```
{'exponent': -2.7297970986317637, 'intercept': 5.401364360728629}
```

[ ]:

[ ]:

CHAPTER 4

The BAK–TANG–WIESENFELD Sandpile

source page 85

4.1 General features

- First published by Bak, Tang, and Wiesenfeld (1987).
- Motivated by avalanching behaviour of a real sandpile.
- In one dimension rules represent downward movement of sand grains.
- Defined in any dimension, exactly solved (trivial) in one.
- Stochastic (bulk) drive, deterministic relaxation.
- Non-Abelian in its original definition.
- Many results actually refer to Dhar's (1990a) Abelian sandpile, Sec. 4.2.
- Simple scaling behaviour disputed, multiscaling proposed.
- Exponents listed in Table 4.1, p. 92, are for the Abelian BTW Model.

4.2 Rules

- d dimensional (usually) hyper-cubic lattice and q the coordination number (on cubic lattices q = 2d).
- Choose (arbitrary) critical slope z^c = q 1.
- Each site n {1,. . . , L}^d has slope z_n.
- *Initialisation*: irrelevant, model studied in the stationary state.
- *Driving*: add a grain at n0 chosen at random and update all uphill nearest neighbours n‘0 of n0: z_n0 →z_n0 + q/2 z_n0 →z_n‘0 1.

- *Toppling*: for each site n with z_n > z^c distribute q grains among its nearest neighbours n' : z_n →z_n q n'.nn.n z_n →z_n + 1. In one dimension site n = L relaxes according to z_L → z_L 1 z_L1 → z_L1 + 1.
- *Dissipation*: grains are lost at open boundaries.
- *Parallel update*: discrete microscopic time, sites exceeding zc at time t topple at t + 1 (updates in sweeps).
- *Separation of time scales*: drive only once all sites are stable, i.e. z_n z^c (quiescence).
- *Key observables* (see Sec. 1.3): avalanche sizes, the total number of topplings until quiescence; avalanche duration T , the total number of parallel updates until quiescence

[20]:
```
from SOC.models import BTW
```

4.3 Empty model

[21]:
```
b = BTW(L = 50, save_every = 50)
```

4.4 Running model

[22]:
```
b.run(200000, wait_for_n_iters = 100)
```

```
Waiting for wait_for_n_iters=100 iterations before collecting data. This should let␣
↪the system thermalize.
```

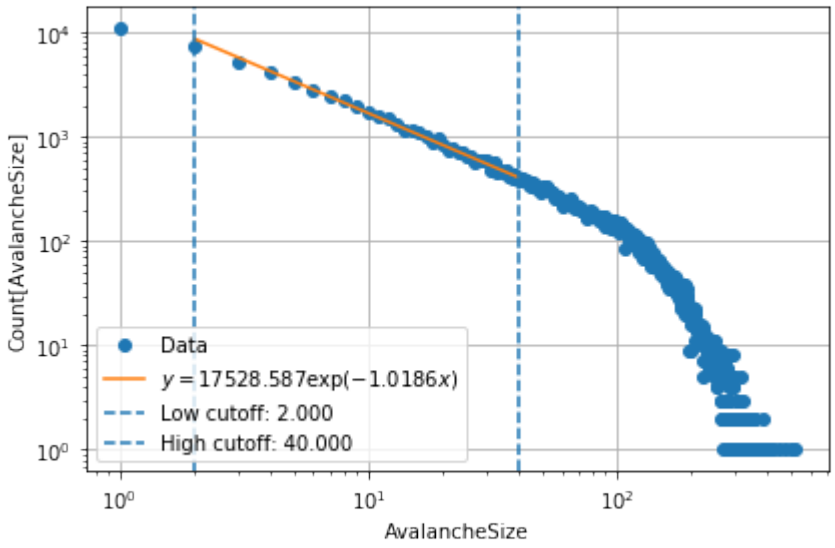
```
HBox(children=(FloatProgress(value=0.0, max=200100.0), HTML(value='')))
```

[23]:
```
b.data_df.describe()
```

[23]:

	AvalancheSize	NumberOfReleases	number_of_iterations
count	200000.000000	200000.000000	200000.000000
mean	87.584035	92.235385	13.013605
std	251.254297	344.621080	32.636649
min	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000
75%	27.000000	12.000000	7.000000
max	2337.000000	10060.000000	519.000000

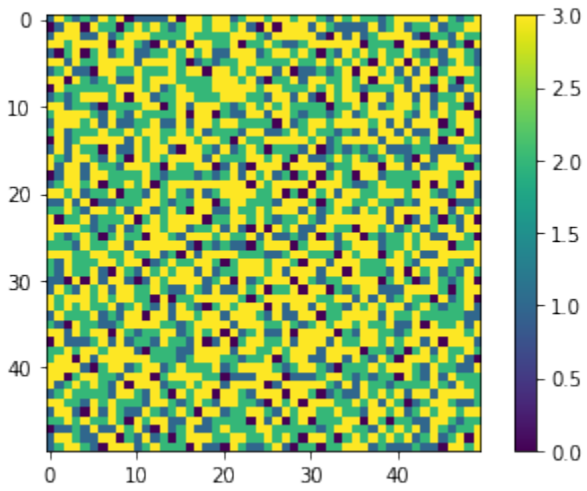
[24]:
```
b.get_exponent(low = 2, high = 40)
```




```
y = 17528.587 exp(-1.0186 x)
```

[24]:
```
{'exponent': -1.018594541688584, 'intercept': 4.243746902353518}
```

[25]:
```
b.plot_state();
```



[26]:
```
%matplotlib inline
a = BTW(30, save_every = 1)
a.run(10000)
```

```
Waiting for wait_for_n_iters=10 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=10010.0), HTML(value='')))
```

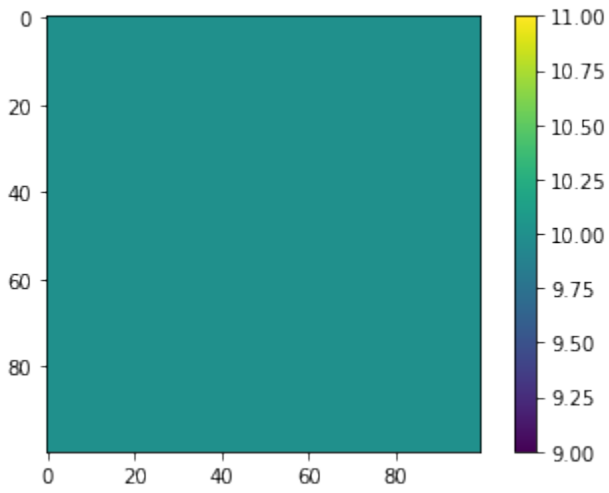
[27]:
```
a.animate_states(notebook = True)
```

```
<IPython.core.display.HTML object>
```

4.5 Uniform model load

[28]:
```
c = BTW(100)
c.values[1:-1,1:-1] = 10
```

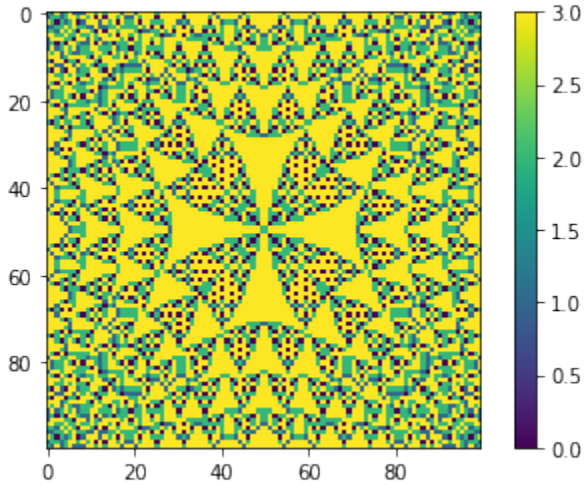
[29]:
```
c.plot_state();
```



[30]:
```
c.AvalancheLoop()
```

[30]:
```
{'AvalancheSize': 10000,
 'NumberOfReleases': 28191892,
 'number_of_iterations': 6920}
```

[31]:
```
c.plot_state();
```



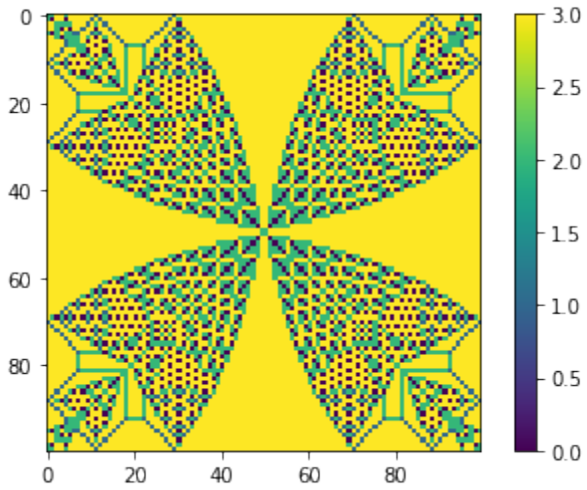
4.5.1 Does pattern depend on height of load?

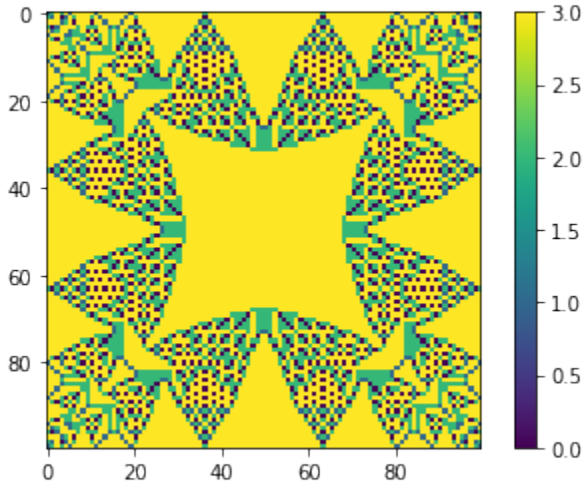
[32]:
```
for i in [4, 5, 6, 10, 20]:
    mdl = BTW(100)
```

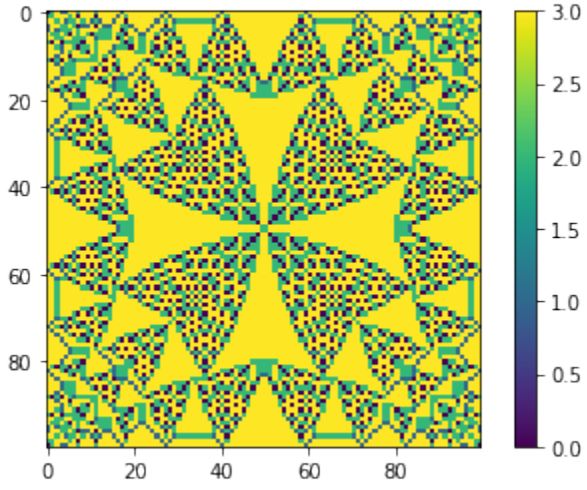
(continues on next page)

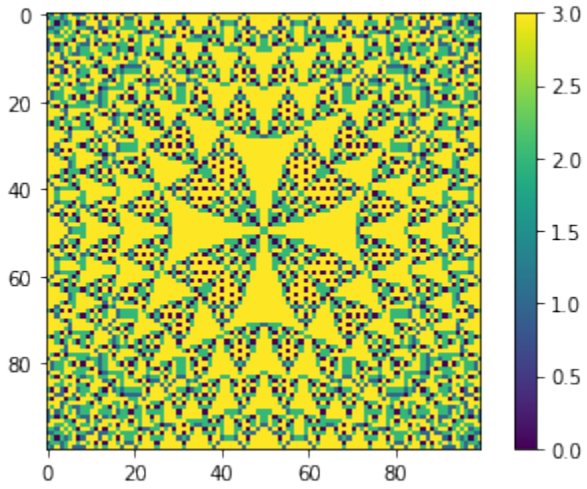
(continued from previous page)

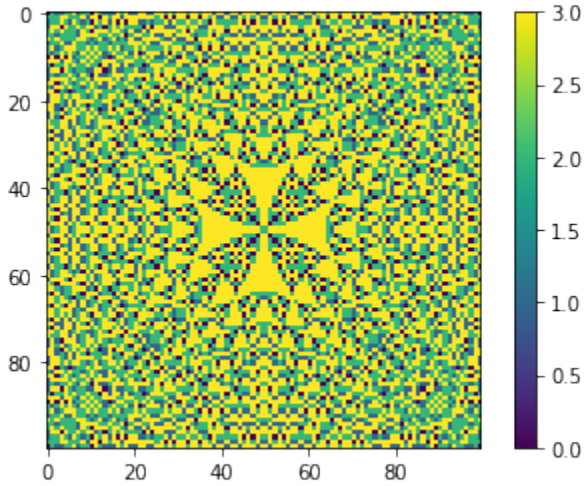
```
    mdl.values[1:-1,1:-1] = i
    mdl.AvalancheLoop()
    mdl.plot_state();
```





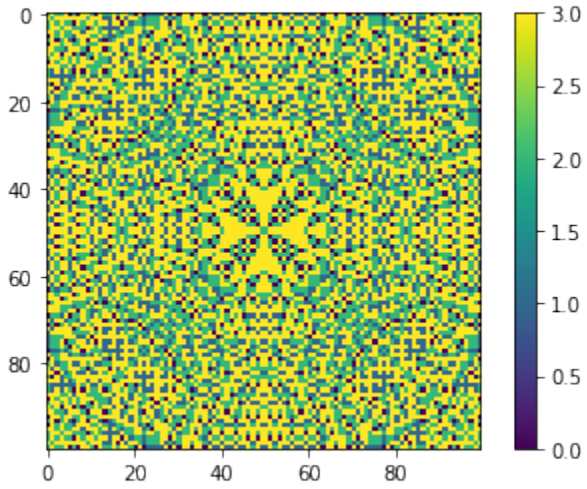




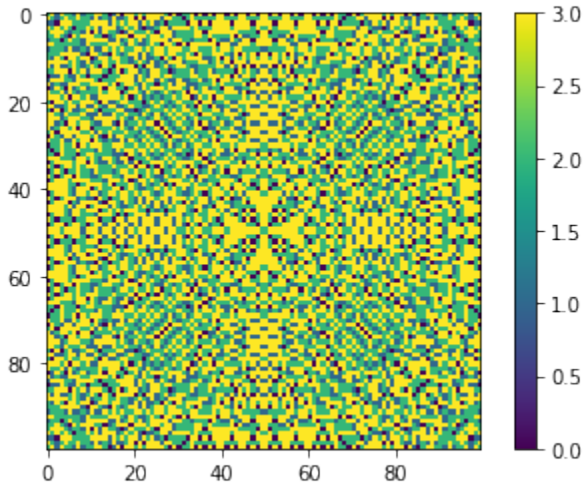


As we can see patter is changeing depending of pile height. **Is there some maximum load**?

[33]:
```
mdl = BTW(100)
mdl.values[1:-1,1:-1] = 40
mdl.AvalancheLoop()
mdl.plot_state();
```



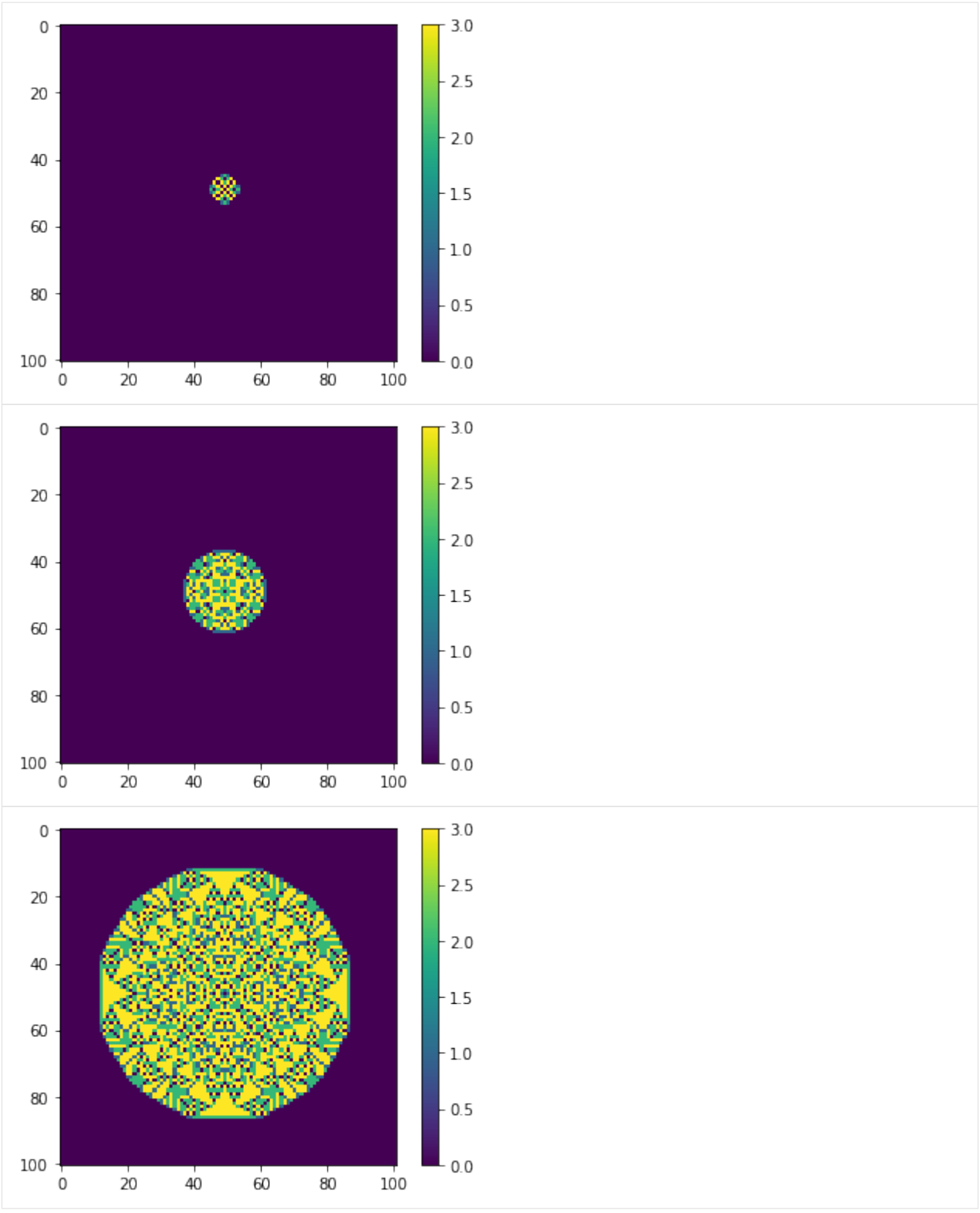
[34]:
```
mdl = BTW(100)
mdl.values[1:-1,1:-1] = 50
mdl.AvalancheLoop()
mdl.plot_state();
```

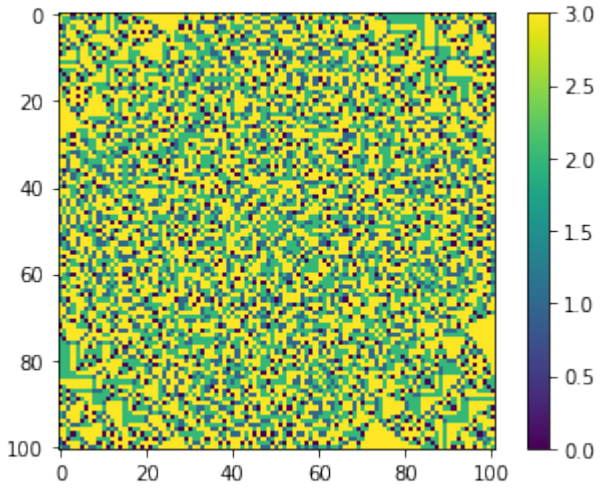


Pattern depends on load height.

4.5.2 Loading model in the single point at centr.

[35]:
```
for i in [10**2, 10**3, 10**4, 10**5]:
    mdl = BTW(101)
    mdl.values[50,50] = i
    mdl.AvalancheLoop()
    mdl.plot_state();
```


0
20
40
60
80
100
0
20
40
60
80
100
3.0
2.5
2.0
1.5
1.0
0.5
0.0



4.6 How exponent dependce on size of system?

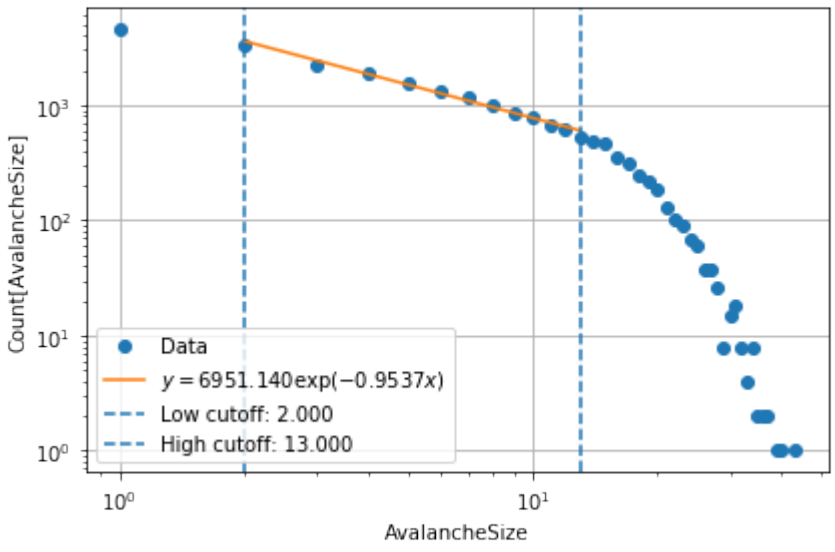
[36]:
```
zip([10, 50, 100, 500],[1,1,1,1])
```

[36]:
```
<zip at 0x7fb61fdc2320>
```

[37]:
```
for l, w in zip([10, 50, 100, 200], [3*100**2, 3*100**2, 3*100**2, 5*100**2]):
    mdl = BTW(L = l, save_every = 100)
    mdl.run(2*w, wait_for_n_iters = w)
    mdl.get_exponent(low = 2, high = 1.3*l)
```

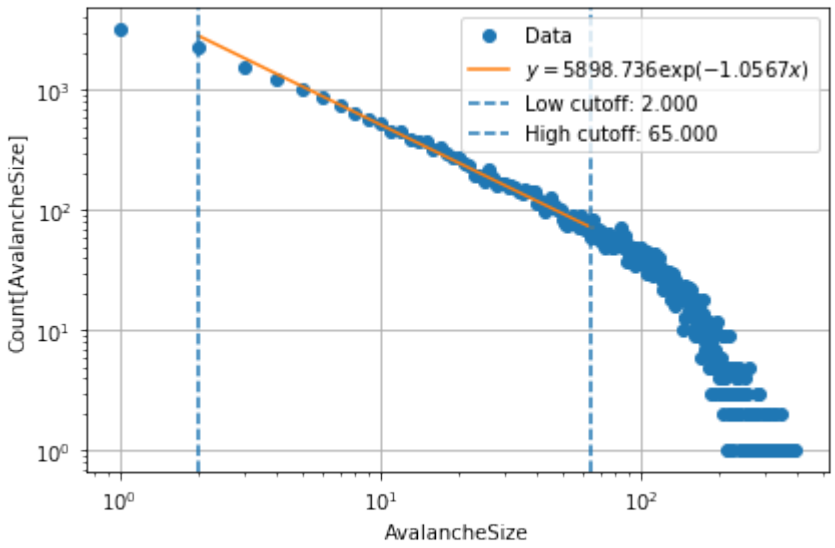
```
Waiting for wait_for_n_iters=30000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=90000.0), HTML(value='')))
```



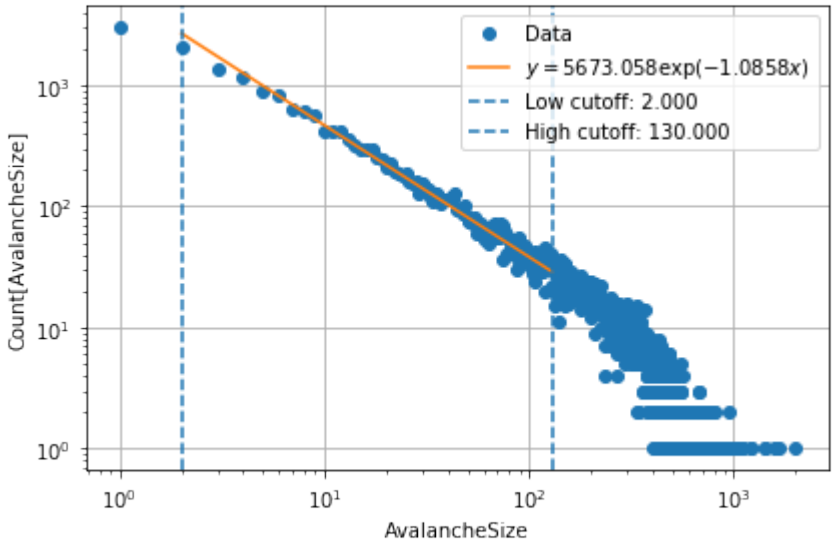
```
y = 6951.140 exp(-0.9537 x)
Waiting for wait_for_n_iters=30000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=90000.0), HTML(value='')))
```



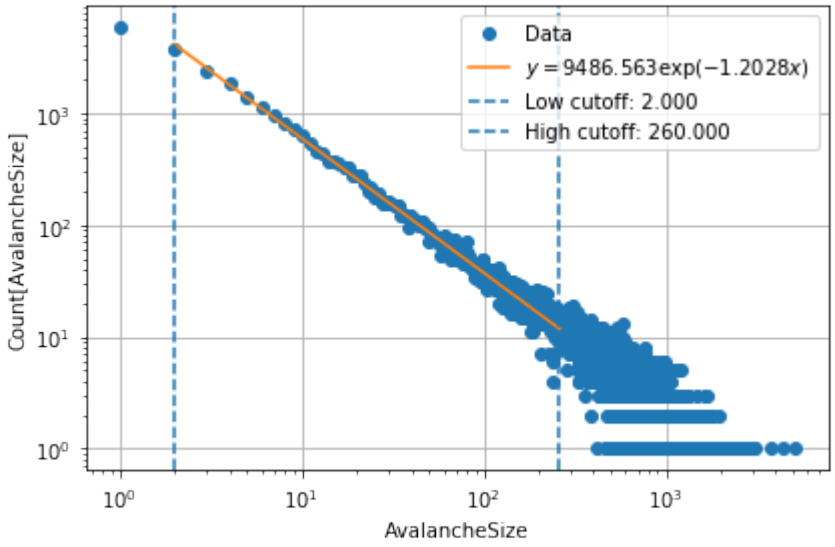

```
y = 5898.736 exp(-1.0567 x)
Waiting for wait_for_n_iters=30000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=90000.0), HTML(value='')))
```




```
y = 5673.058 exp(-1.0858 x)
Waiting for wait_for_n_iters=50000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=150000.0), HTML(value='')))
```




```
y = 9486.563 exp(-1.2028 x)
```

CHAPTER 5

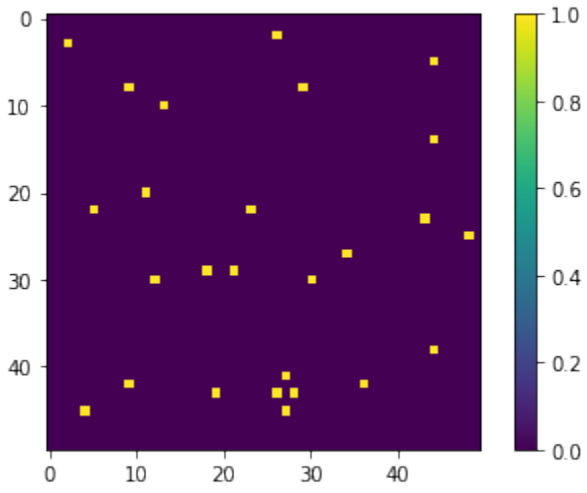
Forest fire model

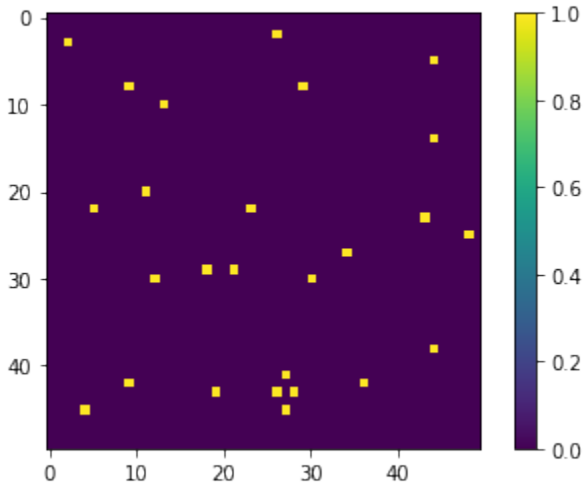
[1]:
```
from SOC.models import Forest
```

```
/progs/miniconda3/lib/python3.7/site-packages/statsmodels/tools/_testing.py:19:␣
↪FutureWarning: pandas.util.testing is deprecated. Use the functions in the public␣
↪API at pandas.testing instead.
  import pandas.util.testing as tm
```

[2]:
```
T chance_of_thunder = 0.001
model = Forest(p=0.04, f=chance_of_thunder, L=50, save_every = 1)

model.plot_state(False)
```

[2]:





[3]:
```
model.run(1000, wait_for_n_iters=1000)
```

```
Waiting for wait_for_n_iters=1000 iterations before collecting data. This should let␣
↪the system thermalize.
```

```
HBox(children=(FloatProgress(value=0.0, max=2000.0), HTML(value='')))
```

- 0 - ash
- 1 - tree
- 2 - burning

[4]:
```
model.animate_states(notebook=True, interval=100)
```

```
<IPython.core.display.HTML object>
```

[5]:
```
model.data_df
```

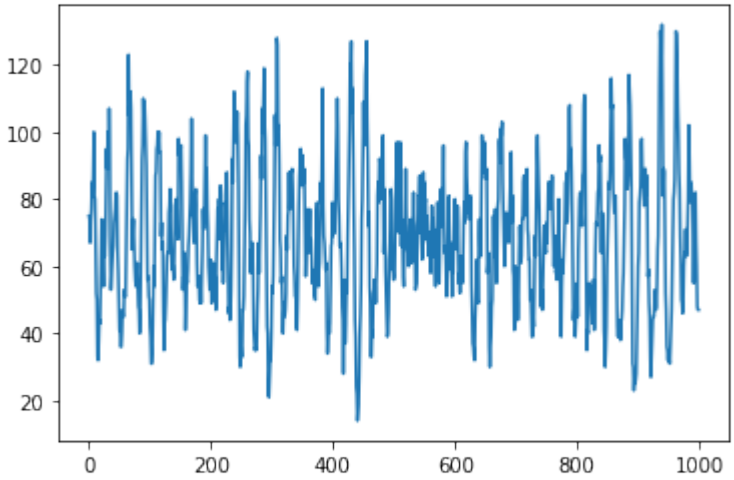
[5]:

	AvalancheSize	NumberOfReleases	number_of_iterations
0	0	0	75
1	0	0	74
2	0	0	67
3	0	0	74
4	0	0	77
..	...	...	...
995	0	0	68
996	0	0	60
997	0	0	48
998	0	0	47
999	0	0	47

[1000 rows x 3 columns]

[14]:
```
model.data_df.number_of_iterations.plot()
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9ea2102f90>
```


120
100
80
60
40
20
0
200
400
600
800
1000

CHAPTER 6

Input/Output

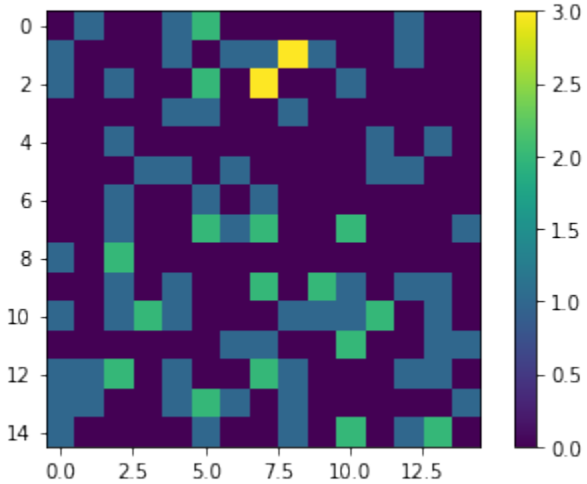
6.1 Saving state of simulation

implementaion is based on zarr library

[1]:
```
from SOC.models import BTW

a = BTW(15)
a.run(100)
a.plot_state()
root = a.save()
root.tree()
```

```
100%|| 100/100 [00:03<00:00, 26.37it/s]
```

[1]:
```
/
 └── values (17, 17) int32
```



[10]:
```
import zarr
read = zarr.open_group('state/sim.zarr', mode = 'r')
read.tree()
```

[10]:
```
/
 └── values (17, 17) int32
```

[11]:
```
read.attrs.keys()
```

[11]:
```
dict_keys(['L', 'save_every'])
```

[12]:
```
print(read.attrs['L'], " -  ", read.attrs['save_every'])
```

```
15 -100
```

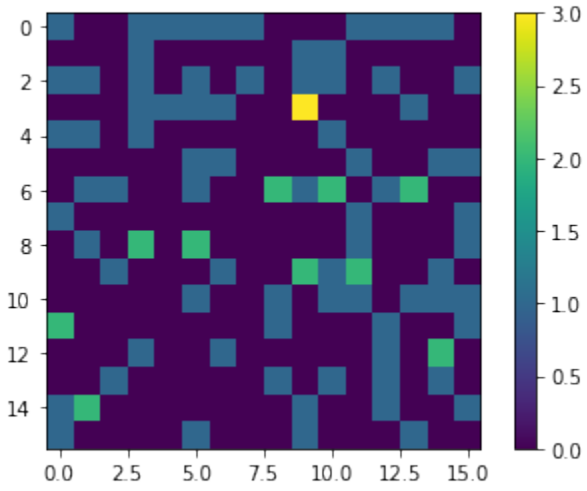
6.2 Values is empty array?

[13]:
```
read['values'][:]
```

[13]:
```
array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

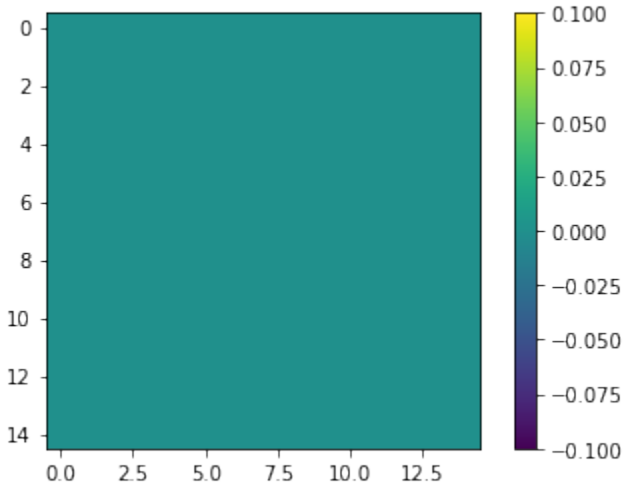
[14]:
```
c = BTW(16)
c.run(100)
c.plot_state();
```

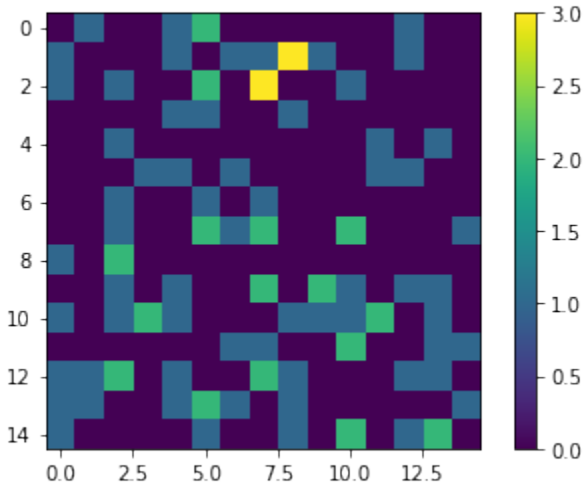
```
100%|| 100/100 [00:00<00:00, 6334.66it/s]
```



[15]:
```
c.open('sim')
```

[16]:
```
c.plot_state();
a.plot_state();
```





[9]:
```
def save(self, file_name = 'sim'):
        """ serialization of object and saving it to file"""

        root = zarr.open_group('state/' + file_name + '.zarr', mode = 'w')
        values = root.create_dataset('values', shape = (self.L_with_boundary, self.L_
↪with_boundary), chunks = (10, 10), dtype = 'i4')
        values = zarr.array(self.values)
        #data_acquisition = root.create_dataset('data_acquisition', shape = (len(self.
↪data_acquisition)), chunks = (1000), dtype = 'i4')
        #data_acquisition = zarr.array(self.data_acquisition)
        root.attrs['L'] = self.L
        root.attrs['save_every'] = self.save_every

        return root

def open(self, file_name = 'sim'):
        root = zarr.open_group('state/' + file_name + '.zarr', mode = 'r')
        self.values = np.array(root['values'][:])
        self.data_acquisition = root['data_acquisition'][:]
        self.L = root.attrs['L']
        self.save_every = root.attrs['save_every']
```

[ ]:

CHAPTER 7

RSOC

7.1 Mesa Tutorial

[1]:
```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
import numpy as np
from mesa.datacollection import DataCollector
from mesa.batchrunner import BatchRunner

# For a jupyter notebook add the following line:
%matplotlib inline

# The below is needed for both notebooks and scripts
import matplotlib.pyplot as plt
```

[2]:
```
class MoneyAgent(Agent):
    """Agent with fixed intial wealth"""

    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.wealth = 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos,
            moore = True,
            include_center = False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def give_money(self):
```

(continues on next page)

(continued from previous page)

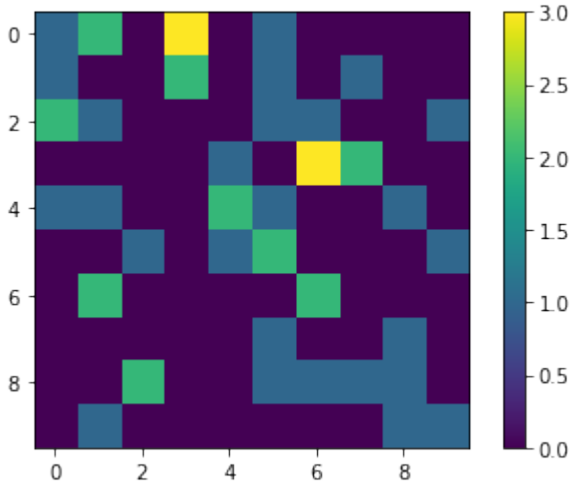
```
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        if len(cellmates) > 1:
            self.wealth -= 1
            other = self.random.choice(cellmates)
            other.wealth += 1

    def step(self):
        self.move()
        if self.wealth > 0:
            self.give_money()
```

[5]:
```
def compute_gini(model):
    agent_wealths = [agent.wealth for agent in model.schedule.agents]
    x = sorted(agent_wealths)
    N = model.num_of_agents
    B = sum( xi * (N-i) for i, xi in enumerate(x) ) / (N*sum(x))
    return (1 + (1/N) - 2*B)

class MoneyModel(Model):
    """A model with some number of agents"""

    def __init__(self, N, width, height):
        super().__init__()
        self.num_of_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)
        self.running = True

        for i in range(self.num_of_agents):
            a = MoneyAgent(i, self)
            self.schedule.add(a)

            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        self.datacollector = DataCollector(
            model_reporters = {"Gini": compute_gini},
            agent_reporters = {"Wealth": "wealth"})

    def plot_state(self):
        agent_counts = np.zeros((self.grid.width, self.grid.height))
        for cell in self.grid.coord_iter():
            cell_content, x, y = cell
            agent_count = len(cell_content)
            agent_counts[x][y] = agent_count
        plt.imshow(agent_counts, interpolation='nearest')
        plt.colorbar()

    def plot_histogram(self):
        agent_wealth = [a.wealth for a in self.schedule.agents]
        plt.hist(agent_wealth)

    def step(self):
        """advance the model by one step"""
        self.datacollector.collect(self)
        self.schedule.step()
```

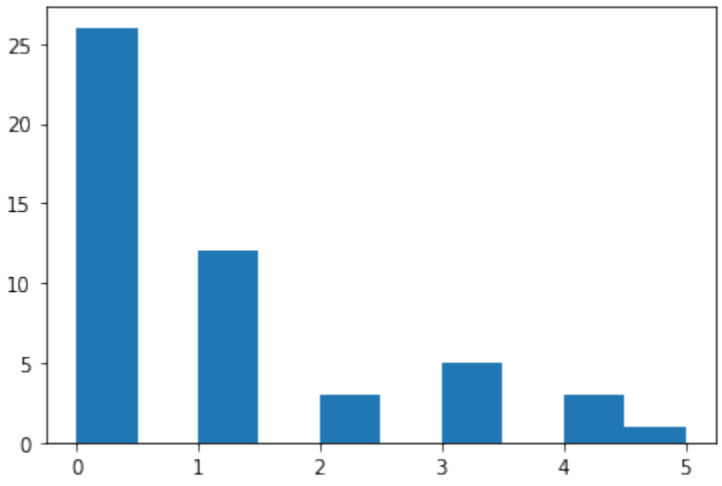
[23]:
```
mdl = MoneyModel(50, 10, 10)
for i in range(1000):
    if i%500 == 0:
        print(i)
    mdl.step()
```

```
0
500
```

[24]:
```
mdl.plot_state()
```

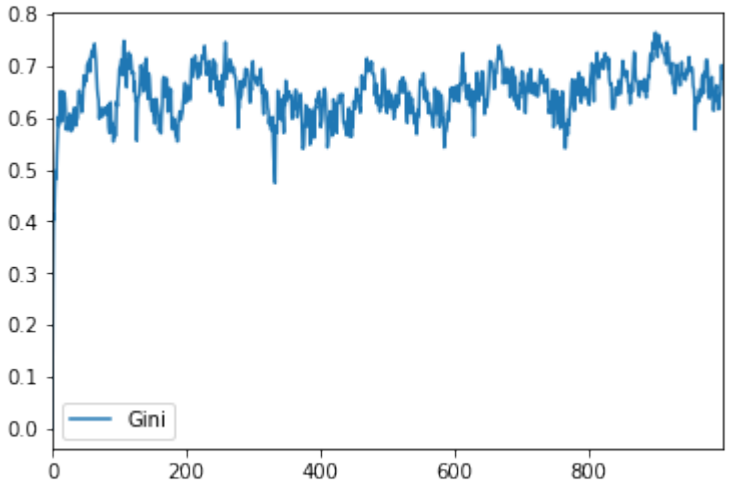


[25]:
```
mdl.plot_histogram()
```



[26]:
```
gini = mdl.datacollector.get_model_vars_dataframe()
gini.plot()
```

[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f3f3b32438>
```

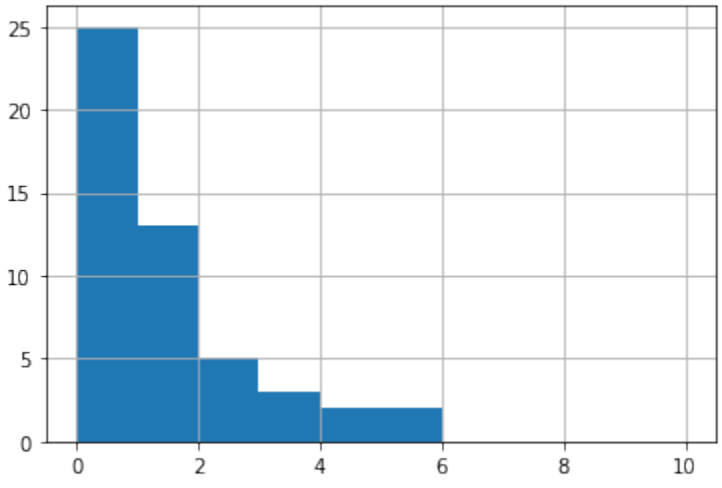


[27]:
```
agent_wealth = mdl.datacollector.get_agent_vars_dataframe()
agent_wealth.head()
```

[27]:
```
              Wealth
Step AgentID
0    0             1
     1             1
     2             1
     3             1
     4             1
```

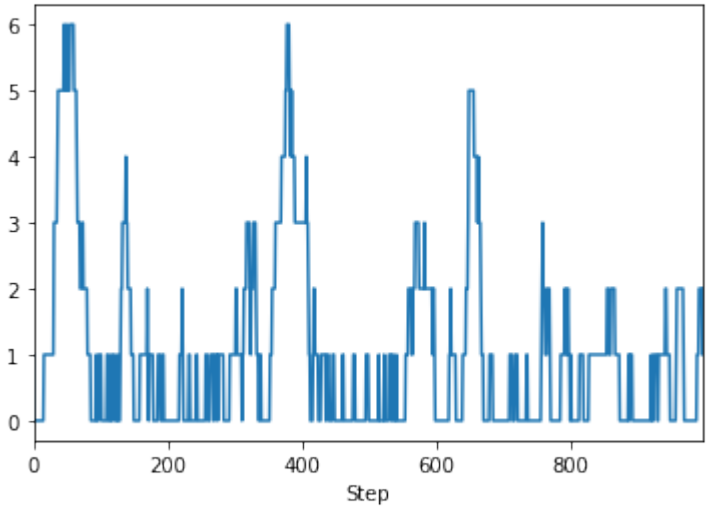
[29]:
```
end_wealth = agent_wealth.xs(99, level="Step")["Wealth"]
end_wealth.hist(bins=range(agent_wealth.Wealth.max() + 1))
```

[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f3f0cb9eb8>
```



[30]:
```
one_agent_wealth = agent_wealth.xs(5, level="AgentID")
one_agent_wealth.Wealth.plot()
```

[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f3f1fedd68>
```



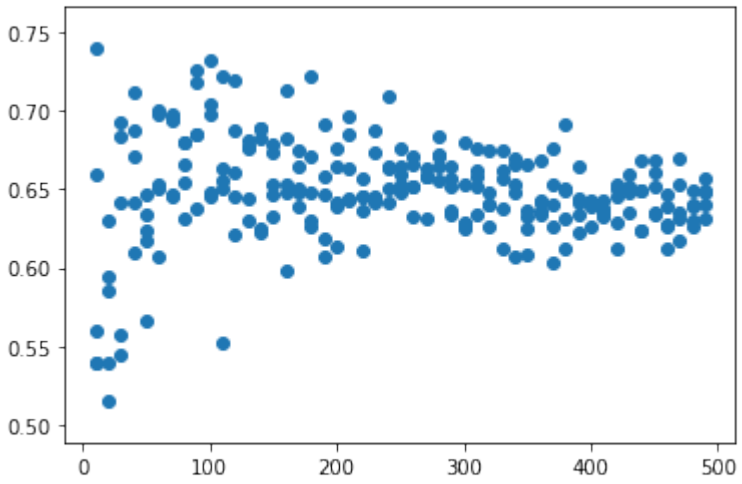
7.2 Batch Runner

[13]:
```
T fixed_params = {
    "width": 10,
    "height": 10
}

variable_params = {"N": range(10, 500, 10)}

# The variables parameters will be invoke along with the fixed parameters allowing␣
↪for either or both to be honored.
batch_run = BatchRunner(
    MoneyModel,
    variable_params,
    fixed_params,
    iterations = 5,
    max_steps = 100,
    model_reporters = {"Gini": compute_gini}
)

batch_run.run_all()
```

```
245it [04:17,  1.05s/it]
```

[14]:
```
run_data = batch_run.get_model_vars_dataframe()
run_data.head()
plt.scatter(run_data.N, run_data.Gini)
```

[14]:
```
<matplotlib.collections.PathCollection at 0x1f3f08ae898>
```



7.3 Visualization

[1]:
```
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer
from mesa.visualization.modules import ChartModule
```

[54]:
```
def agent_portrayal(agent):
    portrayal = {"Shape": "circle",
                 "Filled": "true",
                 "r": 0.5}

    if agent.wealth > 0:
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    else:
        portrayal["Color"] = "grey"
        portrayal["Layer"] = 1
        portrayal["r"] = 0.2
    return portrayal
```

[55]:
```
grid = CanvasGrid(agent_portrayal, 100, 100, 500, 500)
chart = ChartModule([{"Label": "Gini",
                      "Color": "Black"}],
                    data_collector_name='datacollector')
```

[50]:
```
server = ModularServer(MoneyModel,
                       [grid, chart],
                       "Money Model",
                       {"N": 30, "width": 100, "height": 100})
server.port = 8529 # The default
server.launch()
```

```
Interface starting at http://127.0.0.1:8528
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-50-cbeb805557c5> in <module>
      4                          {"N": 30, "width": 100, "height": 100})
      5 server.port = 8528 # The default
----> 6 server.launch()

~\AppData\Local\Programs\Python\Python37\lib\site-
↪packages\mesa\visualization\ModularVisualization.py in launch(self, port)
    322         webbrowser.open(url)
    323         tornado.autoreload.start()
--> 324         tornado.ioloop.IOLoop.current().start()

~\AppData\Local\Programs\Python\Python37\lib\site-packages\tornado\platform\asyncio.
↪py in start(self)
    146             self._setup_logging()
    147             asyncio.set_event_loop(self.asyncio_loop)
--> 148             self.asyncio_loop.run_forever()
    149         finally:
    150             asyncio.set_event_loop(old_loop)

~\AppData\Local\Programs\Python\Python37\lib\asyncio\base_events.py in run_
↪forever(self)
    524         self._check_closed()
    525         if self.is_running():
--> 526             raise RuntimeError('This event loop is already running')
    527         if events._get_running_loop() is not None:
    528             raise RuntimeError(

RuntimeError: This event loop is already running
```

7.4 Ok, Ants

[3]:
```
from mesa import Agent, Model
from mesa.time import RandomActivation
from mesa.space import MultiGrid
import numpy as np
from mesa.datacollection import DataCollector
from mesa.batchrunner import BatchRunner
# For a jupyter notebook add the following line:
%matplotlib inline

# The below is needed for both notebooks and scripts
import matplotlib.pyplot as plt
```

[15]:
```
class Ant(Agent):
    """Ant with fixed intial hunger"""

    def __init__(self, unique_id, model, initial_hunger_value = 460):
        super().__init__(unique_id, model)
        self.hunger = initial_hunger_value
        self.state = 'wandering'#feeding, satiated, disturbed
        #intially we have 'eat_seeking' if ant finds food, then he starts 'eating'␣
↪until he will be 'satisfied'
```

(continues on next page)

(continued from previous page)

```
    def is_satiated(self):
        return self.hunger == 0

    def is_food_found(self):
        return self.model.food_array[self.pos[0], self.pos[1]] > 0

    def eat(self):
        if self.hunger > 0:
            self.hunger -= 1

    def move(self):
        possible_steps = self.model.grid.get_neighborhood(
            self.pos,
            moore = False,
            include_center = False
        )
        new_position = self.random.choice(possible_steps)
        self.model.grid.move_agent(self, new_position)

    def disturb(self):
        cellmates = self.model.grid.get_cell_list_contents([self.pos])
        for antmate in cellmates:
            if antmate.state == 'feeding':
                antmate.state = 'disturbed'

    def step(self):
        #movement
        if self.state == 'disturbed':
            self.state = 'wandering'
            self.move()
        elif not self.is_satiated() and not self.is_food_found():
            self.state = 'wandering'
            self.move()
        elif not self.is_satiated() and self.is_food_found():
            self.disturb()
            self.state = 'feeding'
            self.eat()
        elif self.is_satiated():
            self.state = 'satiated'
            self.move()
```

[12]:
```
class AntModel(Model):
    """A model with some number of ants"""

    def __init__(self, av_number_of_drive_ants = 1, number_of_food = 20, width = 20,␣
↪height = 20):
        super().__init__()

        self.global_agent_index = 1
        self.width = width
        self.height = height

        self.av_number_of_drive_ants = av_number_of_drive_ants

        self.initialize_food_grid(number_of_food)
        self.grid = MultiGrid(width, height, False)
```

(continues on next page)

(continued from previous page)

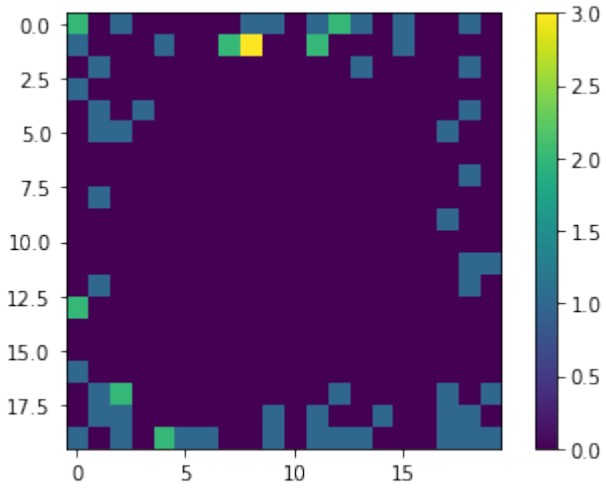
```
        self.schedule = RandomActivation(self)
        self.running = True
        self.datacollector = DataCollector(
            agent_reporters = {"Hunger": "hunger"})

    def initialize_food_grid(self, number_of_food):
        self.number_of_food = number_of_food
        food_array = np.zeros((self.width*self.height), dtype=int)
        assert number_of_food <= self.width*self.height
        food_array[:number_of_food] = 1
        food_array = np.random.permutation(food_array)
        self.food_array = food_array.reshape((self.width, self.height))

    def add_ant(self, pos):
        assert pos[0] >= 0 and pos[0] <= self.width
        assert pos[1] >= 0 and pos[1] <= self.height
        ant = Ant(self.global_agent_index, self)
        self.schedule.add(ant)
        x,y = pos
        self.grid.place_agent(ant, (x, y))
        self.global_agent_index += 1

    def drive(self):
        number_of_ants_to_add = np.random.poisson(self.av_number_of_drive_ants)

        for i in range(number_of_ants_to_add):

            coordHoriz = (
                np.random.choice([0, self.width - 1]),
                np.random.randint(self.height))

            coordVert = (
                np.random.randint(self.width),
                np.random.choice([0, self.height - 1]))

            index = np.random.choice([0, 1])
            coord = np.array([coordHoriz, coordVert])[index]

            self.add_ant(coord)

    def topple(self):
        pass

    def satiated_ants_on_boundary(self):
        ants = []
        for ant in self.schedule.agents:
            if ant.is_satiated():
                pos = ant.pos
                if pos[0] == 0 or pos[0] == self.width - 1 or pos[1] == 0 or pos[1]␣
↪== self.height - 1:
                    ants.append(ant)
        return ants

    def dissipate(self):
        for ant in self.satiated_ants_on_boundary():
            self.schedule.remove(ant)
            self.grid.remove_agent(ant)
```

(continues on next page)

(continued from previous page)

```
    def step(self):
        """advance the model by one step"""
        self.drive()
        self.topple()
        self.dissipate()
        self.datacollector.collect(self)
        self.schedule.step()

    def plot_state(self):
        agent_counts = np.zeros((self.grid.width, self.grid.height))
        for cell in self.grid.coord_iter():
            cell_content, x, y = cell
            agent_count = len(cell_content)
            agent_counts[x][y] = agent_count
        plt.imshow(agent_counts, interpolation = 'nearest')
        plt.colorbar()

    def plot_food(self):
        plt.imshow(self.food_array)
```

[16]:
```
mdl = AntModel(av_number_of_drive_ants = 10, number_of_food = 10,  width = 20, height
↪= 20)

mdl.step()

mdl.plot_state()
```



[18]:
```
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer

def agent_portrayal(agent):
    portrayal = {
        "Shape": "circle",
        "Filled": "true",
        "r": 1,
        "Color": "red",
        "Layer": 0,
```

(continues on next page)

(continued from previous page)

```
        "text": agent.hunger,
        "text_color": "black"}

    if agent.state == 'wandering':
        portrayal["Color"] = "red"
        portrayal["Layer"] = 0
    elif agent.state == 'feeding':
        portrayal["Color"] = "yellow"
        portrayal["Layer"] = 1
    elif agent.state == 'satiated':
        portrayal["Color"] = "green"
        portrayal["Layer"] = 2
    elif agent.state == 'disturbed':
        portrayal["Color"] = "black"
        portrayal["Layer"] = 3

    return portrayal

grid = CanvasGrid(agent_portrayal, 30, 30, 700, 700)
server = ModularServer(AntModel,
                       [grid],
                       "Rapid Self-Organized Criticality",
                       {"av_number_of_drive_ants": 0.05, "number_of_food" : 30, "width
↪": 30, "height": 30})
server.port = 8551 # The default
server.launch()
```

```
Interface starting at http://127.0.0.1:8551
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-18-525ae9a27079> in <module>
     33                          {"av_number_of_drive_ants": 0.05, "number_of_food" :␣
↪30, "width": 30, "height": 30})
     34 server.port = 8551 # The default
---> 35 server.launch()

~\AppData\Local\Programs\Python\Python37\lib\site-
↪packages\mesa\visualization\ModularVisualization.py in launch(self, port)
    322         webbrowser.open(url)
    323         tornado.autoreload.start()
--> 324         tornado.ioloop.IOLoop.current().start()

~\AppData\Local\Programs\Python\Python37\lib\site-packages\tornado\platform\asyncio.
↪py in start(self)
    146             self._setup_logging()
    147             asyncio.set_event_loop(self.asyncio_loop)
--> 148             self.asyncio_loop.run_forever()
    149         finally:
    150             asyncio.set_event_loop(old_loop)

~\AppData\Local\Programs\Python\Python37\lib\asyncio\base_events.py in run_
↪forever(self)
    524         self._check_closed()
    525         if self.is_running():
--> 526             raise RuntimeError('This event loop is already running')
    527         if events._get_running_loop() is not None:
    528             raise RuntimeError(
```

(continues on next page)

(continued from previous page)

```
RuntimeError: This event loop is already running
```

CHAPTER 8

API Overview

class SOC.common.simulation.**Simulation**(*L: int*, *save_every: int = 1*, *wait_for_n_iters: int = 10*)

Base class for SOC simulations.

Parameters

- **L** (*int*) – linear size of lattice, without boundary layers
- **save_every** (*int or None*) – number of iterations per snapshot save
- **wait_for_n_iters** (*int*) – How many iterations to skip to skip before saving data?

AvalancheLoop() → dict

Bring the current simulation's state to equilibrium by repeatedly toppling and dissipating.

Returns a dictionary with the total size of the avalanche and the number of iterations the avalanche took.

Return type dict

L_with_boundary

The total width of the simulation grid, with boundaries.

animate_states(*notebook: bool = False*, *with_boundaries: bool = False*, *interval: int = 30*)

Animates the collected states of the simulation.

Parameters

- **notebook** (*bool*) – if True, displays via html5 video in a notebook; otherwise returns MPL animation
- **with_boundaries** (*bool*) – include boundaries in the animation?
- **interval** (*int*) – number of miliseconds to wait between each frame.

classmethod clean_boundary_inplace(*array: numpy.ndarray*) → numpy.ndarray

Convenience wrapper to *common.clean_boundary_inplace* with the simulation's boundary size.

Parameters **array** (*np.ndarray*) – array to clean

Return type np.ndarray

data_df
Displays the gathered data as a Pandas DataFrame.

Returns dataframe with gathered data

Return type pandas.DataFrame

drive()
Drive the simulation by adding particles from the outside.

Must be overriden in subclasses.

classmethod from_file(*filename: str*)
Loads simulation state from a saved one.

Parameters **filename** (*str*) – Filename to be loaded.

Returns simulation object, of the subclass you used

Return type *Simulation*

get_exponent(*column: str = 'AvalancheSize'*, *low: int = 1*, *high: int = 10*, *plot: bool = True*, *plot_filename: Optional[str] = None*) → dict
Plot histogram of gathered data from data_df,

Parameters

- **column** (*str*) – which column of data_df should be visualized?
- **low** (*int*) – lower cutoff for log-log-linear fit
- **high** (*int*) – higher cutoff for log-log-linear fit
- **plot** (*bool*) – if False, skips all plotting and just returns fit parameters
- **plot_filename** (*bool*) – optional filename for saved plot. This skips displaying the plot!

Returns fit parameters

Return type dict

classmethod inside(*array: numpy.ndarray*) → numpy.ndarray
Convenience function to get an array without simulation boundaries

Parameters **array** (*np.ndarray*) – array

Returns array of width smaller by 2BC

Return type np.ndarray

plot_state(*with_boundaries: bool = False*) → matplotlib.figure.Figure
Plots the current state of the simulation.

Parameters **with_boundaries** (*bool*) – should the boundaries be displayed as well?

Returns figure with plot

Return type plt.Figure

run(*N_iterations: int*, *filename: str = None*, *wait_for_n_iters: int = 10*) → str
Simulation loop. Drives the simulation, possibly starts avalanches, gathers data.

Parameters

- **N_iterations** (*int*) – number of iterations (per grid node if *scale* is True)
- **filename** (*str*) – filename for saving snapshots. if None, saves to memory; by default if False, makes something like array_Manna_2019-12-17T19:40:00.546426.zarr

- **wait_for_n_iters** (*int*) – wait this many iterations before collecting data (lets model thermalize)

Return type dict

save(*file_name='sim'*)
serialization of object and saving it to file

size
The total size of the simulation grid, without boundaries

topple_dissipate()
Distribute material from overloaded sites to neighbors.

Must be overriden in subclasses.

class SOC.models.**BTW**(**args*, ***kwargs*)
Implements the BTW model.

Parameters **L** (*int*) – linear size of lattice, without boundary layers

drive(*num_particles: int = 1*)
Drive the simulation by adding particles from the outside.

Parameters **num_particles** (*int*) – How many particles to add per iteration (by default, 1)

topple_dissipate() → int
Distribute material from overloaded sites to neighbors.

Convenience wrapper for the numba.njitted *topple* function defined in *manna.py*.

Return type int

class SOC.models.**Forest**(*p: float = 0.05*, *f: float = 0*, **args*, ***kwargs*)
Forest fire model

Parameters

- **f** – probability of thunder setting a tree on fire; set 0 to disable lighting
- **p** (*float*) – probability of a new tree growh per empty cell

drive()
Does nothing in FF!

topple_dissipate() → int
Forest burning and turning into ash.

class SOC.models.**Manna**(*critical_value: int = 1*, *abelian: bool = True*, **args*, ***kwargs*)
Implements the Manna model.

drive(*num_particles: int = 1*)
Drive the simulation by adding particles from the outside.

Parameters **num_particles** (*int*) – How many particles to add per iteration (by default, 1)

topple_dissipate() → int
Distribute material from overloaded sites to neighbors.

Convenience wrapper for the numba.njitted *topple_dissipate* function defined in *manna.py*.

Returns number of iterations it took to

Return type bool

class SOC.models.**OFC**(*critical_value: float = 1.0*, *conservation_lvl: float = 0.25*, **args*, ***kwargs*)
Implements the OFC model.

Parameters

- **L** (*int*) – linear size of lattice, without boundary layers
- **critical_value** (*float*) – 1.0 by default - above this value, nodes start toppling. At 0.25 -> full force distributed (if 4 neighbours)
- **conservation_lvl** (*float*) – 0.25 by default - fraction of the force from a toppling site going to its neighbour

AvalancheLoop() → dict

Bring the current simulation's state to equilibrium by repeatedly toppling and dissipating.

Returns a dictionary with the total size of the avalanche and the number of iterations the avalanche took.

Return type dict

drive()

Drive the simulation by adding force from the outside.

topple_dissipate() → int

Distribute material from overloaded sites to neighbors.

Convenience wrapper for the numba.njitted *topple* function defined in *ofc.py*.

Return type int

Index