cellsystem Documentation

Release 0.1.1

Ad115

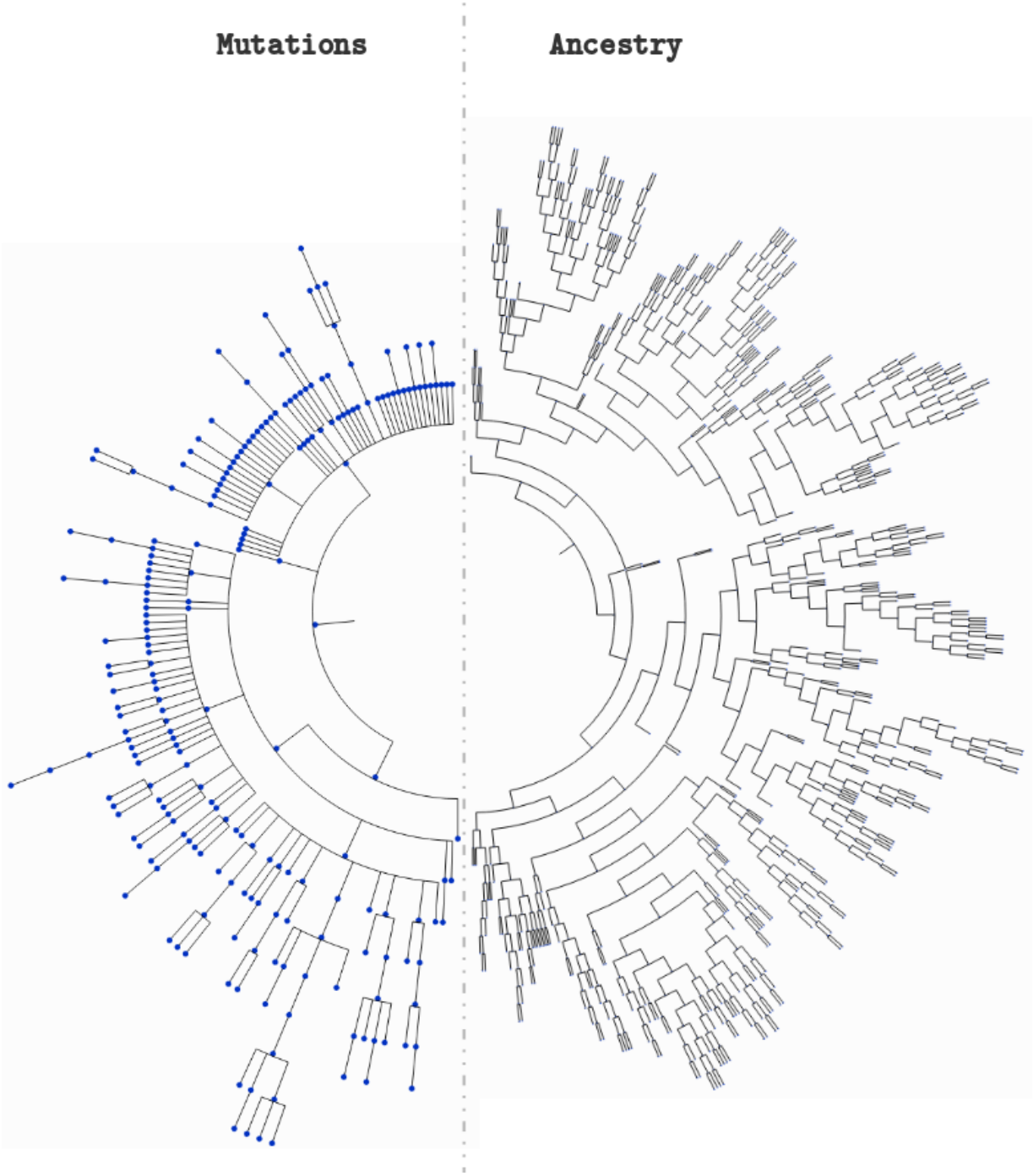
Jun 14, 2018

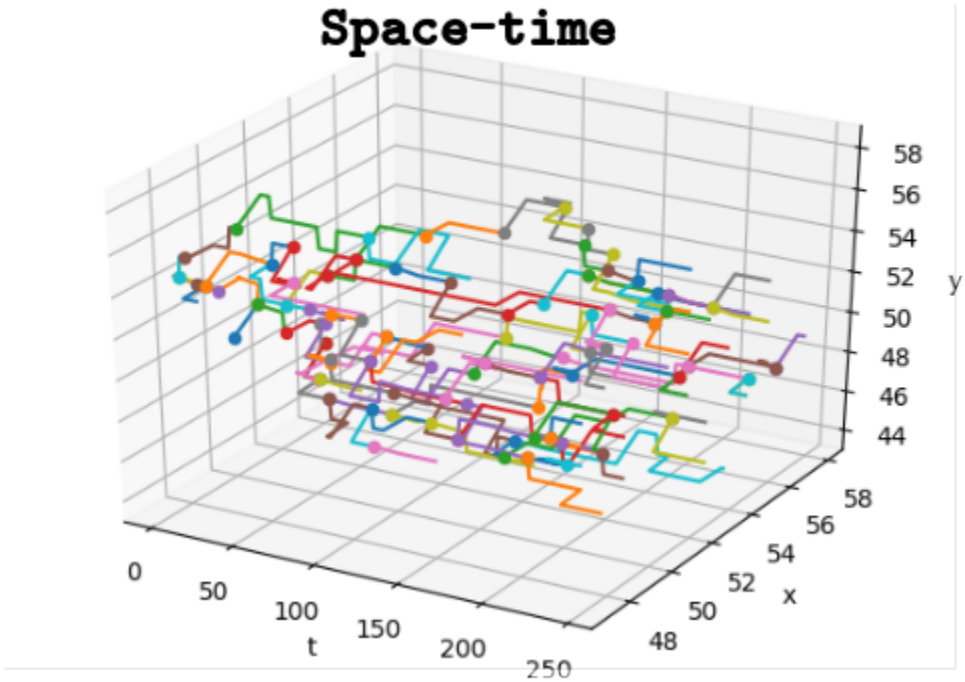
Contents

CHAPTER 1

CellSystem

This was created to simulate cancer growth, taking into account nutrients and cell migration while allowing to track mutations, cell division and cell position history to study tumour phylogeny reconstruction algorithms.


Mutations
Ancestry



1.1 Installation

You can install it from PyPI:

```
$ pip install cellsystem
```

1.2 Example

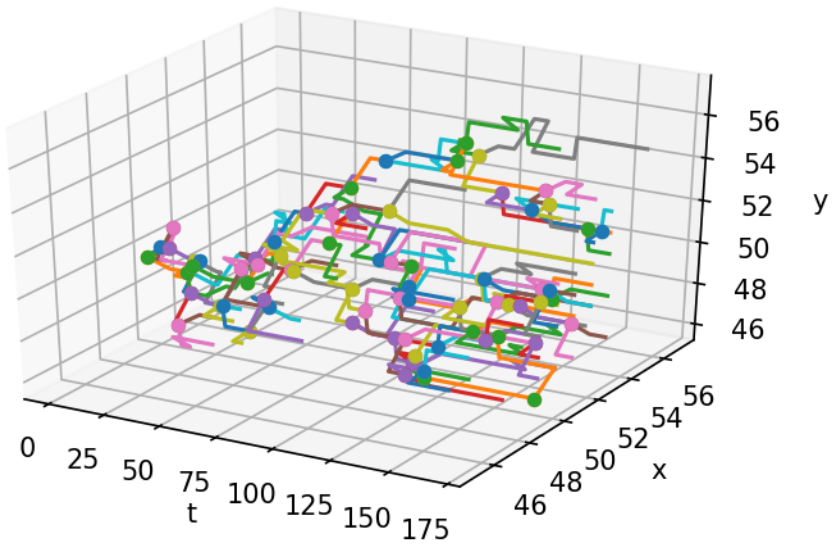
A use case integrated in the repository:

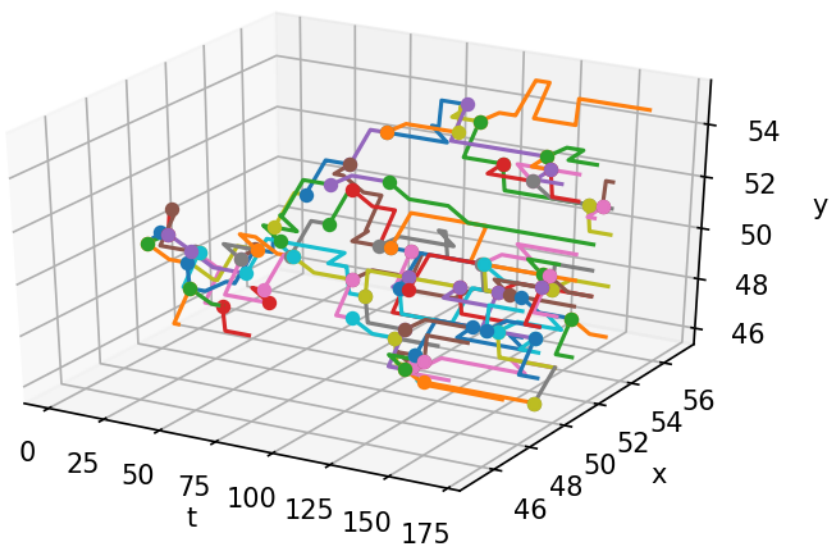
```
>>> from cellsystem import *

# The cell system will simulate cell growth
# while tracking the steps in that process.
>>> system = CellSystem(grid_shape=(100, 100))

# Initialize the first cell
# in the middle of the grid
>>> system.seed()


New cell 0 added @ (50, 50)


# Take 35 steps forward in time
>>> system.run(steps=30)
```

(continues on next page)

(continued from previous page)

```
Cell no. 0 mutating @ site (50, 50) (father None)
         Initial mutations: []
                     Initial genome: AAAAAAAAAA
         Final mutations: [(4, 'G')]
                     Final genome: AAAAGAAAAA
Cell no. 0 dividing @ (50, 50)
    New cells: 1 @ (49, 50) and 2 @ (50, 51)
Cell no. 2 dividing @ (50, 51)
    New cells: 3 @ (51, 52) and 4 @ (51, 52)
Cell no. 4 mutating @ site (51, 52) (father 2)
         Initial mutations: [(4, 'G')]
                     Initial genome: AAAAGAAAAA
         Final mutations: [(4, 'G'), (7, 'A')]
                     Final genome: AAAAGAAAAA
Cell no. 1 death @ site (49, 50) (father None)
Cell no. 3 death @ site (51, 52) (father 2)
Cell no. 4 mutating @ site (51, 52) (father 2)
         Initial mutations: [(4, 'G'), (7, 'A')]
                     Initial genome: AAAAGAAAAA
         Final mutations: [(4, 'G'), (7, 'A'), (2, 'T')]
                     Final genome: AATAGAAAAA
Cell no. 4 migrating from site (51, 52) (father 2)
     New site: (50, 52)
...
...
...


# Prepare to explore the simulation logs
>>> history = system['log']


# First, let's see the cells' evolution in time and space!
>>> history.worldlines().show()

# Remove the cells that died somewhere along the way
>>> history.worldlines(prune_death=True).show()
```



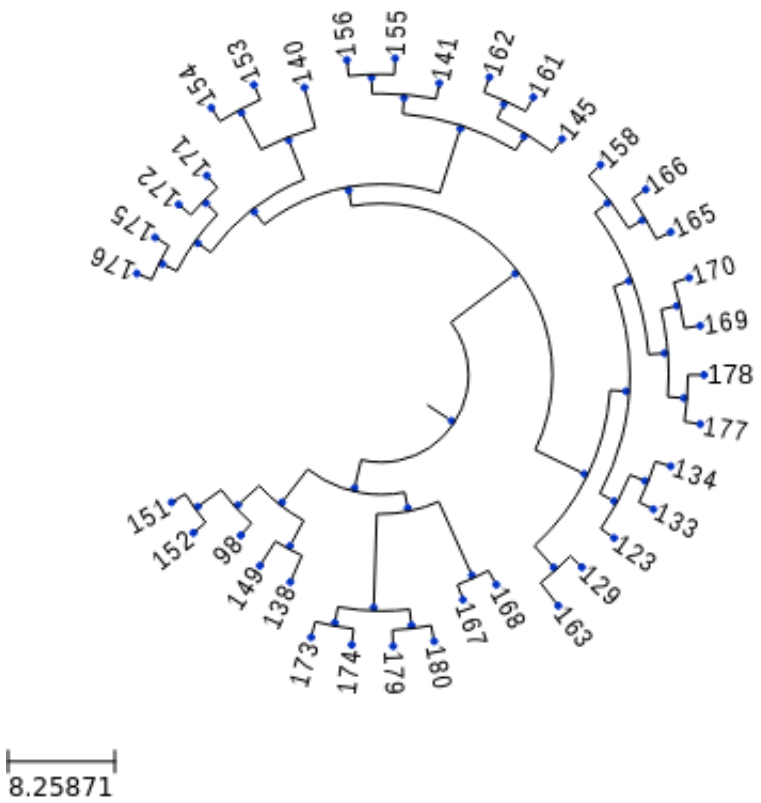



```
>>> tree_style = {'show_leaf_name' : True,
...               'mode' : 'c',         # Circular
...               'arc_start' : -135,   # Degrees
...               'arc_span' : 270 }    # Degrees also


# Lookup the tree formed by cellular division
```

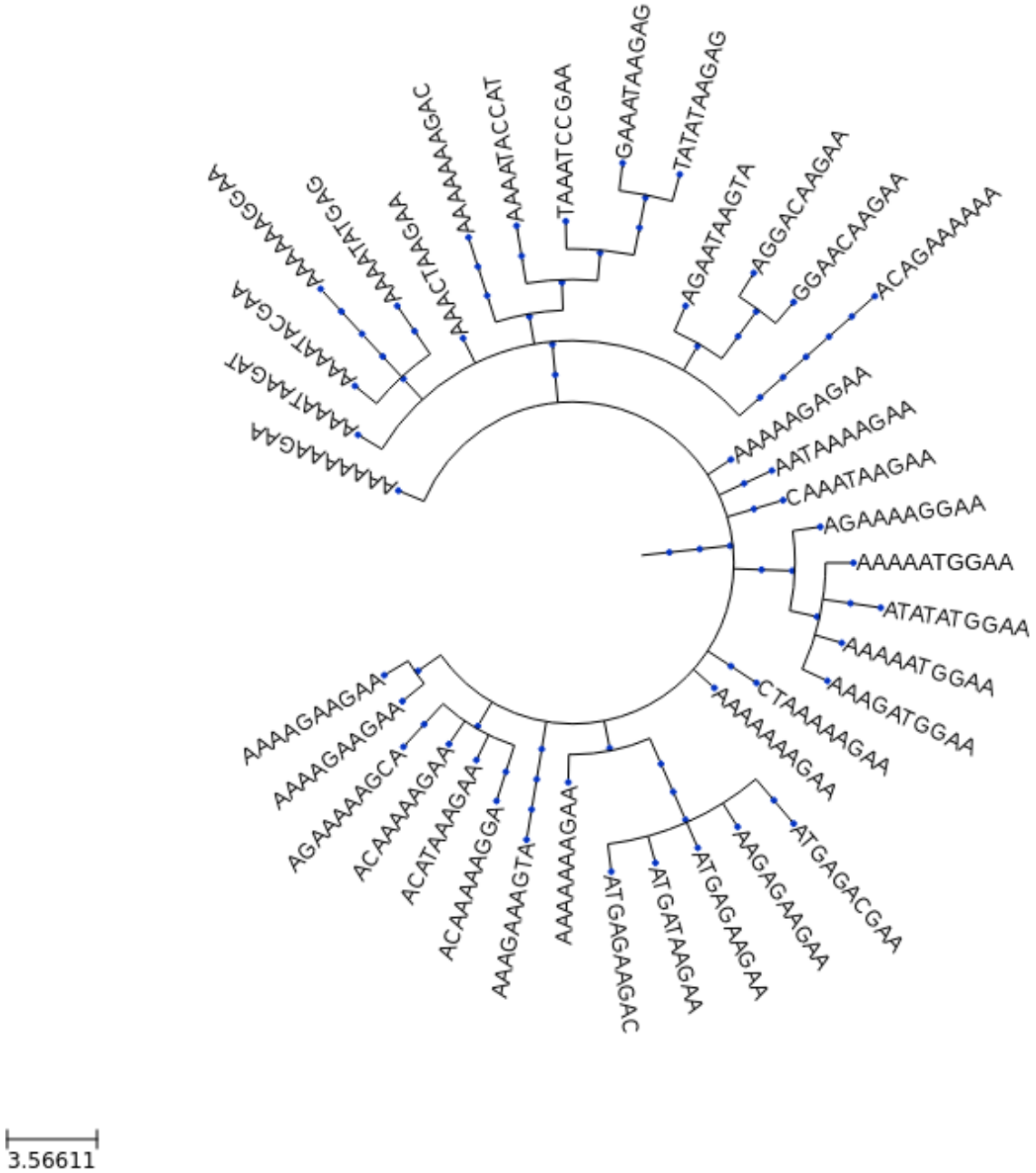
(continues on next page)

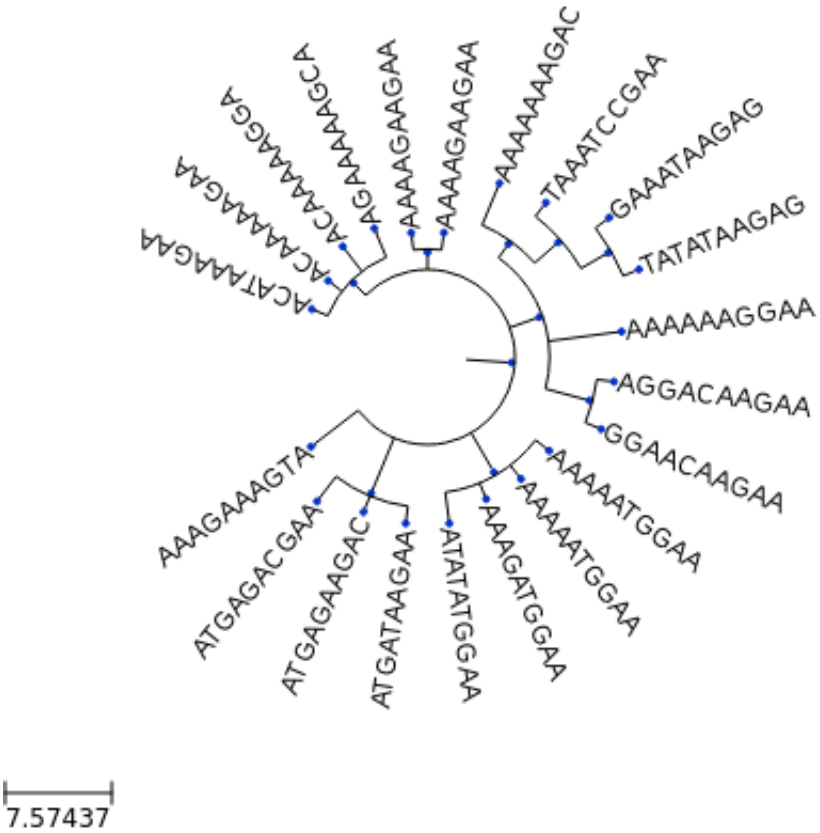
(continued from previous page)

```
>>> history.ancestry().show(styling=tree_style)

# Now, remove cells that are no longer alive
>>> history.ancestry(prune_death=True).show(styling=tree_style)
```




```
# Now, check out the tree formed by the mutations
>>> history.mutations().show(styling=tree_style)

# Remove genomes with no living representatives.
>>> history.mutations(prune_death=True).show(styling=tree_style)
```


AAAAAAAGAA
AAAATAAGAT
AAAATACGAA
AAAAAAGGAA
AAAATATGAG
AAAACTAGAA
AAAAAAAGAC
AAAATACCAT
TAAATCCGAA
GAAATAAGAG
TATATAAGAG
AGAATAAGTA
AGGACAAGAA
GGAACAAGAA
ACAGAAAAAA
AAAAAGAGAA
AATAAAAGAA
CAAATAAGAA
AGAAAAGGAA
AAAAATGGAA
ATATATGGAA
AAAAATGGAA
AAAGATGGAA
CTAAAAAGAA
AAAAAAAGAA
ATGAGACGAA
AAGAGAAGAA
ATGAGAAGAA
ATGATAAGAA
ATGAGAAGAC
AAAAAAAGAA
AAAGAAAGTA
ACAAAAAGGA
ACATAAAGAA
ACAAAAAGAA
AGAAAAAGCA
AAAAGAAGAA
AAAAGAAGAA
3.56611



For more examples and usage, please refer to the [Wiki](wikigoeshere.com).

1.3 Meta

Author: Ad115 - Github – a.garcia230395@gmail.com

Distributed under the MIT license. See LICENSE for more information.

Warning: The project is still in alpha stage, so the API is just stabilizing and may change in the near future. This also means that if you want to contribute, now is the right moment to make important change suggestions ;D

1.4 Contributing

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug.
2. Fork the repository on GitHub to start making your changes to a feature branch, derived from the **master** branch.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published.

CHAPTER 2

cellsystem

2.1 cellsystem package

2.1.1 Subpackages

cellsystem.logging package

Subpackages

cellsystem.logging.core package

Submodules

cellsystem.logging.core.log module

class `cellsystem.logging.core.log.`**`Log`**(**args*, ***kwargs*)

Bases: `object`

A logger class that registers certain actions.

`activate`()

Activate the log if deactivated.

`log`(*actionname*, **args*, ***kwargs*)

Log the action.

`preparefor`(*actionname*, **args*, ***kwargs*)

Save previous state before the entity takes the given action.

`silence`()

Silence/deactivate log temporarily.

cellsystem.logging.core.multi module

class cellsystem.logging.core.multi.**MultiLog**

Bases: *cellsystem.logging.core.log.Log*

An aggregate of logs.

log(*actionname*, **args*, ***kwargs*)

Log the action.

preparefor(*actionname*, **args*, ***kwargs*)

Save previous state before the entity takes the given action.

register(*log*, *name*)

Register a named log entity.

cellsystem.logging.core.weak module

class cellsystem.logging.core.weak.**WeakLog**(**args*, ***kwargs*)

Bases: *cellsystem.logging.core.log.Log*

Ignore silently calls to not implemented log methods.

log(*action*, **args*, ***kwargs*)

Log the action.

preparefor(*action*, **args*, ***kwargs*)

Save previous state before the entity takes the given action.

Module contents

Core logging classes

The fundamental interfaces for logging.

class cellsystem.logging.core.**Log**(**args*, ***kwargs*)

Bases: `object`

A logger class that registers certain actions.

activate()

Activate the log if deactivated.

log(*actionname*, **args*, ***kwargs*)

Log the action.

preparefor(*actionname*, **args*, ***kwargs*)

Save previous state before the entity takes the given action.

silence()

Silence/deactivate log temporarily.

class cellsystem.logging.core.**WeakLog**(**args*, ***kwargs*)

Bases: *cellsystem.logging.core.log.Log*

Ignore silently calls to not implemented log methods.

log(*action*, **args*, ***kwargs*)

Log the action.

preparefor(*action*, **args*, ***kwargs*)
Save previous state before the entity takes the given action.

class cellsystem.logging.core.**MultiLog**
Bases: *cellsystem.logging.core.log.Log*

An aggregate of logs.

log(*actionname*, **args*, ***kwargs*)
Log the action.

preparefor(*actionname*, **args*, ***kwargs*)
Save previous state before the entity takes the given action.

register(*log*, *name*)
Register a named log entity.

Submodules

cellsystem.logging.full module

class cellsystem.logging.full.**FullLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.core.multi.MultiLog*

A log that records geometric information, mutations, ancestry and prints the actions to the screen.

Each part can be accesed with:

```
log[{{logname}}]
```

where {{logname}} can be one of: 'geometry', 'mutations', 'ancestry' or 'printer'.

also, each log can be (de)activated with:

```
# Deactivate log
log[{{logname}}].silence()

# Rectivate log
log[{{logname}}].activate()
```

ancestry(*prune_death=False*)
Fetch information of the cells' "family tree".

mutations(*prune_death=False*)
Fetch information of the cells' mutational history.

worldlines(*prune_death=False*)
Fetch information of the cells' evolution in space and time.

cellsystem.logging.geometric module

Geometric Logging

This module defines functionality for following the geometric evolution of the cell blob through time.

The classes GeometricLog and WorldLines are defined here.

class cellsystem.logging.geometric.**GeometricLog**(**args*, ***kwargs*)

Bases: *cellsystem.logging.core.weak.WeakLog*

Registers the geometric positions of the cells.

iter_changes()

Iterate through every action

iter_states()

Generate the intermediate states.

log_death(*cell*)

Register the disappeareance of the cell.

log_division(*daughters*)

log_migration(*cell*)

log_newcell(*cell*)

preparefor_division(*cell*)

worldlines(*prune_death=False*)

Get the geometric evolution of individual cells in space and time.

class cellsystem.logging.geometric.**WorldLines**(*initial_state=None*, *time=0*)

Bases: object

A class that represents the worldlines of a set of cells.

A worldline is the place in time and space that a cell occupies throughout it's existence.

add_event(*cell*, *event*)

Add an event (a spacetime coordinate) for the cell.

classmethod from_log(*geometric_log*, *prune_death=False*)

Initialize from a geometric log.

last_state_of(*cell*)

Return the last recorded event of the given cell.

remove(*cell*)

Remove the given cell's timeline.

show(*div_marker='o'*, *end_marker=''*, *savefig=None*)

Render the 3D worldlines as a plot.

update(*state*, *transition*, *time*, *prune_death=False*)

Add the event described by the transition, time and final state.

cellsystem.logging.simple module

cellsystem.logging.treelogs module

class cellsystem.logging.treelogs.**AncestryLog**(**args*, ***kwargs*)

Bases: *cellsystem.logging.treelogs.TreeLog*

A tree log that maintains a "family tree".

Each leaf represents a cell. When that cell divides, the leaf branches into leaves representing the daughters.

add_child(**args*, ***kwargs*)

log_division(*daughters*)
Add 2 new branches to the father of the cells.

log_newcell(*cell*)

class cellsystem.logging.treelogs.**MutationsLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.treelogs.TreeLog*

A tree log that maintains a record of genome branching events.

Each leaf represents a genome that may be present in one or more cells. When one of those cells mutates, the new genome is added as a child of that leaf.

log_division(*daughters*)
Remove the father from the alive cells.

log_mutation(*cell*)
Add a new child to the parent genome.

log_newcell(*cell*)

class cellsystem.logging.treelogs.**TreeLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.core.weak.WeakLog*

Base class for logs that grow trees.

add_child(*parent=None*, *name=None*)

alive_nodes

fetch_tree(*prune_death=False*)
Fetch a copy of the tree.

If *prune_death* is True, remove the leaves that correspond to death cells.

log_death(*cell*)

preparefor_division(*cell*)

Module contents

Logging classes.

Classes related to the recording of the simulation progress, analysis and history.

class cellsystem.logging.**FullLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.core.multi.MultiLog*

A log that records geometric information, mutations, ancestry and prints the actions to the screen.

Each part can be accesed with:

```
log[{{logname}}]
```

where {{logname}} can be one of: 'geometry', 'mutations', 'ancestry' or 'printer'.

also, each log can be (de)activated with:

```
# Deactivate log
log[{{logname}}].silence()

# Rectivate log
log[{{logname}}].activate()
```

ancestry(*prune_death=False*)
Fetch information of the cells' "family tree".

mutations(*prune_death=False*)
Fetch information of the cells' mutational history.

worldlines(*prune_death=False*)
Fetch information of the cells' evolution in space and time.

class cellsystem.logging.**PrinterLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.core.log.Log*

Simple logger that limits to print the action.

log_death(*cell*)

log_division(*daughters*)

log_migration(*cell*)

log_mutation(*cell*)

log_newcell(*cell*)

preparefor_death(*cell*)

preparefor_division(*cell*)

preparefor_migration(*cell*)

preparefor_mutation(*cell*)

class cellsystem.logging.**MutationsLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.treelogs.TreeLog*

A tree log that maintains a record of genome branching events.

Each leaf represents a genome that may be present in one or more cells. When one of those cells mutates, the new genome is added as a child of that leaf.

log_division(*daughters*)
Remove the father from the alive cells.

log_mutation(*cell*)
Add a new child to the parent genome.

log_newcell(*cell*)

class cellsystem.logging.**AncestryLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.treelogs.TreeLog*

A tree log that maintains a "family tree".

Each leaf represents a cell. When that cell divides, the leaf branches into leaves representing the daughters.

add_child(**args*, ***kwargs*)

log_division(*daughters*)
Add 2 new branches to the father of the cells.

log_newcell(*cell*)

class cellsystem.logging.**GeometricLog**(**args*, ***kwargs*)
Bases: *cellsystem.logging.core.weak.WeakLog*

Registers the geometric positions of the cells.

iter_changes()
Iterate through every action

iter_states()
Generate the intermediate states.

log_death(*cell*)
Register the disappeareance of the cell.

log_division(*daughters*)

log_migration(*cell*)

log_newcell(*cell*)

preparefor_division(*cell*)

worldlines(*prune_death=False*)
Get the geometric evolution of individual cells in space and time.

`cellsystem.logging.`**logged**(*action_name*, *prepare=True*)
Decorate an action with logging.

Allows to specify if a prelogging action is called.

cellsystem.simulation package

Submodules

cellsystem.simulation.cells module

Structures representing the biological entities.

class `cellsystem.simulation.cells.`**Cell**(*lineage*, *index*)
Bases: `object`

A single cell.

It acts according to it's state, and the states of nearby cells and sites.

+ Index
A label that identifies it among others in the same lineage.

+ Father
The index (lineage label) of it's father.

+ CellLine
The lineage this cell belongs to.

+ Site
The place in the grid this cell inhabits in.

+ Mutations
The mutations in this cell relative to the cell lineage's reference

add_mutation(*position*, *mutated*)
Add a mutation to the cell in the given position of the genome.

Note: The genome may not represent a nucleotide sequence, so these mutations may not represent SNPs.

add_to(*site*)
Add the cell to the site.

ancestral_genome
Ancestral genome of the cell lineage.

coordinates
Coordinates of the site that the cell inhabits.

genome
Genome of the cell.

It is assembled from the cell's ancestral genome and it's mutations.

genome_alphabet
Genome alphabet of the cell line.

The set of characters a genome is composed of. (The genome characters may really represent genes, aminoacids, etc...)

initialize(*site=None*, *father=None*, *mutations=None*)
Initialize grid site, mutations and father.

mutations
Record of the mutations the cell has had.

The mutations are relative to the ancestral genome.

new_daughter()
Initialize a new daughter cell.

Initialize with father and mutation attributes.

process(**args*, ***kwargs*)
Select an action and perform it.

class cellsystem.simulation.cells.**CellLine**(**args*, *genome=None*, *genome_alphabet=None*, *recycle_dead=True*, ***kwargs*)

Bases: object

Handles the specimens of a specific cell lineage.

A cell lineage is a group of cells that have a common ancestor, we represent the common ancestor by it's ancestral genome. This structure is in charge of holding this ancestral code and managing the cells creating new cells when needed and cleaning up the dead ones.

Atributes:

- Ancestral genome: A string-like object.
- Cells: The cells inherited from this cell line.
- Alive/Dead cells.
- **Current cell index: Each cell has a unique index. This is the** index to place in the next cell to be born.

add_behaviors(*behaviors*, *weights=None*)
Add the behaviors defining the cells from this cell line.

Params:

behaviors (list of callables): The list of actions that the cells in this lineage will be able to perform.

weights (optional list of numeric values): The list of relative weights for selecting each action. The bigger the weight of an action relative to the weights of the others, the more likely is

that that action will be selected by the cell at each step. default is all actions have the same weights.

Raises `ValueError` – If the weights are not of the same length as the behaviors.

cell_to_recycle()
Return a cell from the dead ones.

fetch_behaviors()
The behaviors that the cells in this lineage perform.

handle_death(*dying*)
Process a dying cell.

This means removing from the alive cells and adding to the dead ones, maybe to recycle it when another is born.

new_cell()
Get a new blank cell in this lineage and system.

process(**args*, ***kwargs*)
Move a step forward in time.

recycle_cell(*cell*)
Clear previous information from a cell.

register_log(*log*)

sample(*all=False*, *n=1*)
Take a sample of alive cells.

Parameters

- **all** – If True, return all alive cells, else, return a sample of size n.
- **n** – The size of the sample. If 1, return the cell without a container.

total_cells

`cellsystem.simulation.cells.`**behavior**(*actionname*, *actionfn=None*, *probability=None*, *prepare=True*)
Assemble a cell behavior.

Adds logging and asociates a name and a probability function to the raw action function. Allows to specify if the logging of the action requires to prepare the log.

Can be used as a function decorator or as a normal function.

cellsystem.simulation.logging module

cellsystem.simulation.system module

System-related classes.

This module defines a general system that can be used as the base of a computation graph.

A system is composed of entities and interactions btw them, it coordinates all processes.

class `cellsystem.simulation.system.`**Entity**
Bases: `object`

An entity is something that resides in the system.

It processes information according to it's internal state and the information flowing through the links it shares with other entities.

An entity registers the following methods:

- process(time)

process(*time*)

class cellsystem.simulation.system.**Interaction**(*entities*, *effect*)
Bases: object

A structure representing flow of information btw entities.

append(*effect*)
Effects btw the same entities can be appended and executed in order

process()
Executes the interaction.

class cellsystem.simulation.system.**Process**(*entities*, *effects*)
Bases: tuple

effects
Alias for field number 1

entities
Alias for field number 0

class cellsystem.simulation.system.**System**(**args*, ***kwargs*)
Bases: object

The global system and event dispatcher.

Aware of the passage of time (steps). A system is composed of entities and interactions between them, at each time step, the system triggers the proceses associated with them.

add_entity(*entity*, *name*, *procesable=True*, *inithook=None*)
Add an entity to the graph.

If procesable, the entity.process(time) method is called on each time step.

Inithooks are callables called at initialization.

add_interaction_to(*container*, *effect*, *entitynames*)
Add an interaction btw named entities to a dict-like container.

link(*effect*, *entitynames*)
Add an interation btw named entities.

process_interactions_in(*interactions*)
From the dict-like container of interactions, process items.

run(*steps*, *init=None*, *after_step=None*, *before_step=None*)
Start running the simulation.

start()
Initialize things before starting simulation.

stateof(*entityname*)
Ask the entity for it's state

step()
Take a single step forward in time.

update_hooks (*hooktype*, *newhooks*)
Add the given hooks to the system.

cellsystem.simulation.world module

Classes associated with physical space where entities live and interact.

class `cellsystem.simulation.world.`**Site** (*world*, *coordinates*)
Bases: `object`

A unit of space.

Cells inhabitate in these spaces and interact with their neighborhood.

Is aware of:

- World: The world it forms a part of.
- Coordinates <i,j>: Coordinates in the matrix.
- Guests: <List>: The guests currently inhabiting this site.

add_guest (*guest*)
Add the given cell as a new guest to this site.

coordinates
Getter for the site's coordinates.

guest_count ()
Return the number of guests residing in this site.

random_neighbor ()
Return a random site in the neighborhood of the current site.

remove_guest (*guest*)
Remove the given cell as guest for this site.

If the cell is not currently in this site, an error is throwed.

class `cellsystem.simulation.world.`**World** (*shape=(10, 10)*, *wrap=<function toroidal_wrap>*)
Bases: `object`

The space in which cells inhabit.

It represents physical space and enforces rules and properties like distance and closeness.

A world is aware of:

- Grid: The sites the action develops in.
- **Neighborhood: How many and which sites may directly influence or** be influenced by another.

at (*coordinates*)
Get the site at the specified coordinates.

middle
Get the site at the middle of the world.

random_neighbor_of (*site*)
Return the relative coordinates of the available neighbors.

Returns a container holding pairs of numbers. the prescence of an item (a,b) means that a site at i,j has a neighbor at (i+a, j+b).

`cellsystem.simulation.world.`**`toroidal_wrap`**(*grid*, *coord*)
Return the coordinates wrapped on the grid dimensions.

`cellsystem.simulation.world.`**`wrap`**(*n*, *maxValue*)
Auxiliary function to wrap an integer on maxValue.

Examples

```
>>> # For positives: wrap(n, maxValue) = n % maxValue
>>> [ wrap(i,3) for i in range(9) ]
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

```
>>> # For negatives, the pattern is continued in a natural way
>>> for i in range(-5, 5+1): print(f'{i} : {wrap(i, 3)}')
...
-3 : 0
-2 : 1
-1 : 2
0 : 0
1 : 1
2 : 2
```

Module contents

class `cellsystem.simulation.`**`System`**(**args*, ***kwargs*)
Bases: `object`

The global system and event dispatcher.

Aware of the passage of time (steps). A system is composed of entities and interactions between them, at each time step, the system triggers the proceses associated with them.

`add_entity`(*entity*, *name*, *procesable=True*, *inithook=None*)
Add an entity to the graph.

If procesable, the entity.process(time) method is called on each time step.

Inithooks are callables called at initialization.

`add_interaction_to`(*container*, *effect*, *entitynames*)
Add an interaction btw named entities to a dict-like container.

`link`(*effect*, *entitynames*)
Add an interation btw named entities.

`process_interactions_in`(*interactions*)
From the dict-like container of interactions, process items.

`run`(*steps*, *init=None*, *after_step=None*, *before_step=None*)
Start running the simulation.

`start`()
Initialize things before starting simulation.

`stateof`(*entityname*)
Ask the entity for it's state

step()
Take a single step forward in time.

update_hooks(*hooktype*, *newhooks*)
Add the given hooks to the system.

class cellsystem.simulation.**CellLine**(**args*, *genome=None*, *genome_alphabet=None*, *recycle_dead=True*, ***kwargs*)

Bases: object

Handles the specimens of a specific cell lineage.

A cell lineage is a group of cells that have a common ancestor, we represent the common ancestor by it's ancestral genome. This structure is in charge of holding this ancestral code and managing the cells creating new cells when needed and cleaning up the dead ones.

Atributes:

- Ancestral genome: A string-like object.
- Cells: The cells inherited from this cell line.
- Alive/Dead cells.
- **Current cell index: Each cell has a unique index. This is the** index to place in the next cell to be born.

add_behaviors(*behaviors*, *weights=None*)
Add the behaviors defining the cells from this cell line.

Params:

behaviors (list of callables): The list of actions that the cells in this lineage will be able to perform.

weights (optional list of numeric values): The list of relative weights for selecting each action. The bigger the weight of an action relative to the weights of the others, the more likely is that that action will be selected by the cell at each step. default is all actions have the same weights.

Raises ValueError – If the weights are not of the same length as the behaviors.

cell_to_recycle()
Return a cell from the dead ones.

fetch_behaviors()
The behaviors that the cells in this lineage perform.

handle_death(*dying*)
Process a dying cell.

This means removing from the alive cells and adding to the dead ones, maybe to recycle it when another is born.

new_cell()
Get a new blank cell in this lineage and system.

process(**args*, ***kwargs*)
Move a step forward in time.

recycle_cell(*cell*)
Clear previous information from a cell.

register_log(*log*)

sample(*all=False*, *n=1*)
Take a sample of alive cells.

Parameters

- **all** – If True, return all alive cells, else, return a sample of size n.
- **n** – The size of the sample. If 1, return the cell without a container.

total_cells

class cellsystem.simulation.**Action**(*action*, *probability=None*, *name=None*)
Bases: object

Objects of this class represent actions with an associated probability.

try_action(**args*, *probability=None*, ***kwargs*)
Perform the action according to it's probability.

class cellsystem.simulation.**World**(*shape=(10, 10)*, *wrap=<function toroidal_wrap>*)
Bases: object

The space in which cells inhabit.

It represents physical space and enforces rules and properties like distance and closeness.

A world is aware of:

- Grid: The sites the action develops in.
- **Neighborhood: How many and which sites may directly influence or** be influenced by another.

at(*coordinates*)
Get the site at the specified coordinates.

middle
Get the site at the middle of the world.

random_neighbor_of(*site*)
Return the relative coordinates of the available neighbors.

Returns a container holding pairs of numbers. the prescence of an item (a,b) means that a site at i,j has a neighbor at (i+a, j+b).

cellsystem.simulation.**behavior**(*actionname*, *actionfn=None*, *probability=None*, *prepare=True*)
Assemble a cell behavior.

Adds logging and asociates a name and a probability function to the raw action function. Allows to specify if the logging of the action requires to prepare the log.

Can be used as a function decorator or as a normal function.

cellsystem.utils package

Submodules

cellsystem.utils.tree module

class cellsystem.utils.tree.**Tree**(*tree=None*)
Bases: object

Wrapper for the ETE Tree class with some conveniences.

copy()
Make a copy of the tree.

prune_leaves(*to_stay*)
Prune tree branches to leave only the leaves in *to_stay*.

show(**args*, *inline=False*, *styling=None*, ***kwargs*)
Display the tree.

Module contents

2.1.2 Submodules

2.1.3 cellsystem.cellsystem module

The cell simulation with logging.

class cellsystem.cellsystem.**CellSystem**(**args*, *grid_shape=(100, 100)*, *init_genome=None*, ***kwargs*)
Bases: *cellsystem.simulation.system.System*

A system simulating cell growth.

A cell system is a system subclass, with the automatic initialization of three main entities:

1. Cells represented by a cell line.
2. A 'world' representing the space that the cells inhabit, and;
3. A 'log' that follows and makes a record of the cells' actions.

Each part can be accessed by `system['cells']`, `system['world']` and `system['log']` respectively.

seed()
Place a single cell in the middle of the world.

class cellsystem.cellsystem.**SimpleCells**(**args*, *genome_alphabet=None*, ***kwargs*)
Bases: *cellsystem.simulation.cells.CellLine*

A cell line representing simple cells with default behaviors.

A cell from this line performs:

- Cell division,
- Cell death,
- Cell migration,
- Cell genome mutation.

add_cell_to(*site*)
Add a new, initialized cell to the given site.

Return the added cell to the caller.

static death(*cell*)
Cellular death.

static death_probability(*cell*)
Cellular death probability.

static division(*cell*, *preserve_father=False*)
Cell division.

Get a new daughter of this cell and place it in a nearby neighboring site.

static division_probability(*cell*)
Probability that this cell will divide if selected for division.

static migration(*cell*)
Migrate to a neighboring cell.

static migration_probability(*cell*)
Migration probability for this cell.

static mutation(*cell*)
Do a single site mutation.

static mutation_probability(*cell*)
Probability to mutate if selected for it.

2.1.4 Module contents

class cellsystem.**CellSystem**(**args*, *grid_shape=(100, 100)*, *init_genome=None*, ***kwargs*)
Bases: *cellsystem.simulation.system.System*

A system simulating cell growth.

A cell system is a system subclass, with the automatic initialization of three main entities:

1. Cells represented by a cell line.
2. A 'world' representing the space that the cells inhabit, and;
3. A 'log' that follows and makes a record of the cells' actions.

Each part can be accessed by `system['cells']`, `system['world']` and `system['log']` respectively.

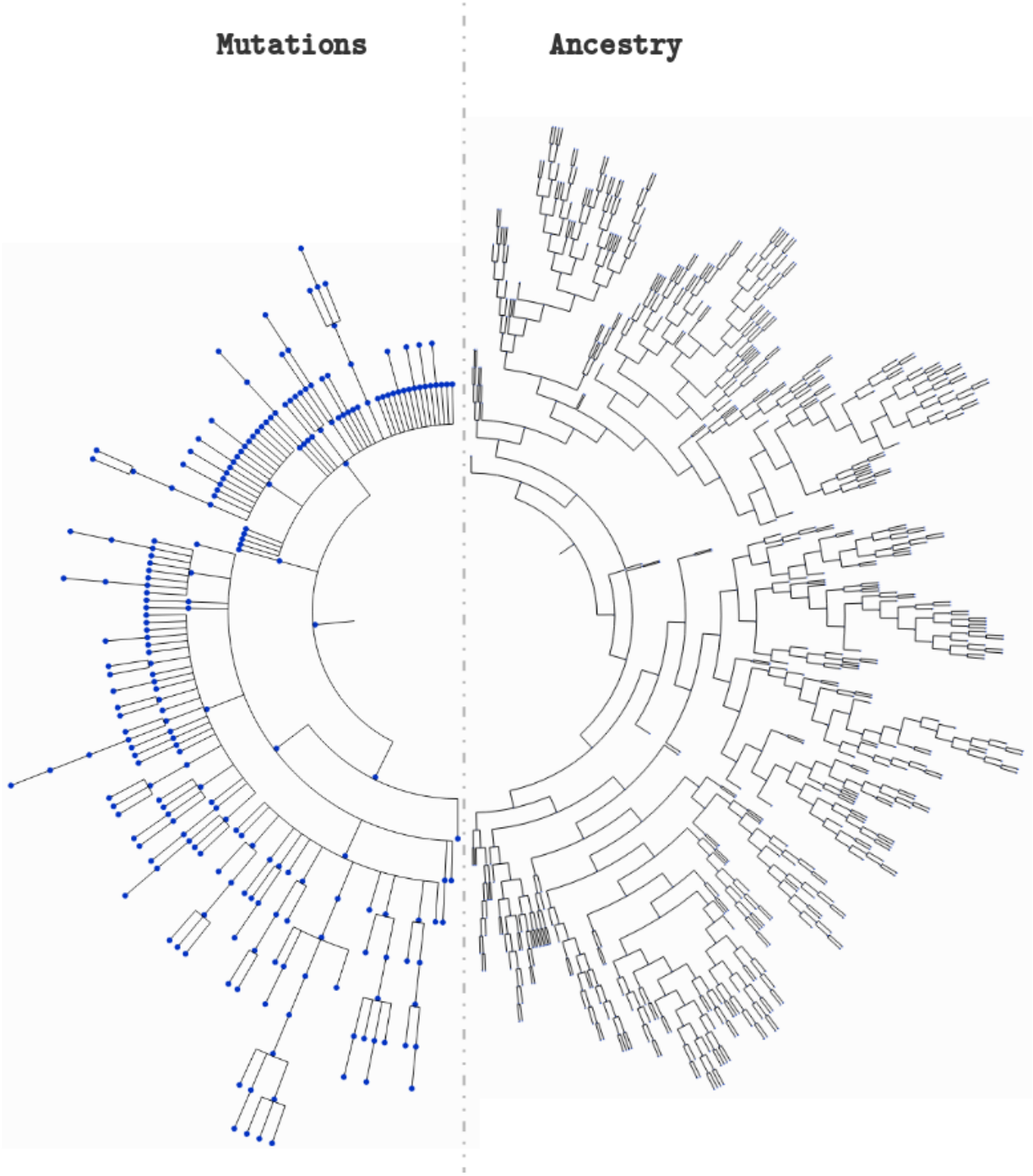
seed()
Place a single cell in the middle of the world.

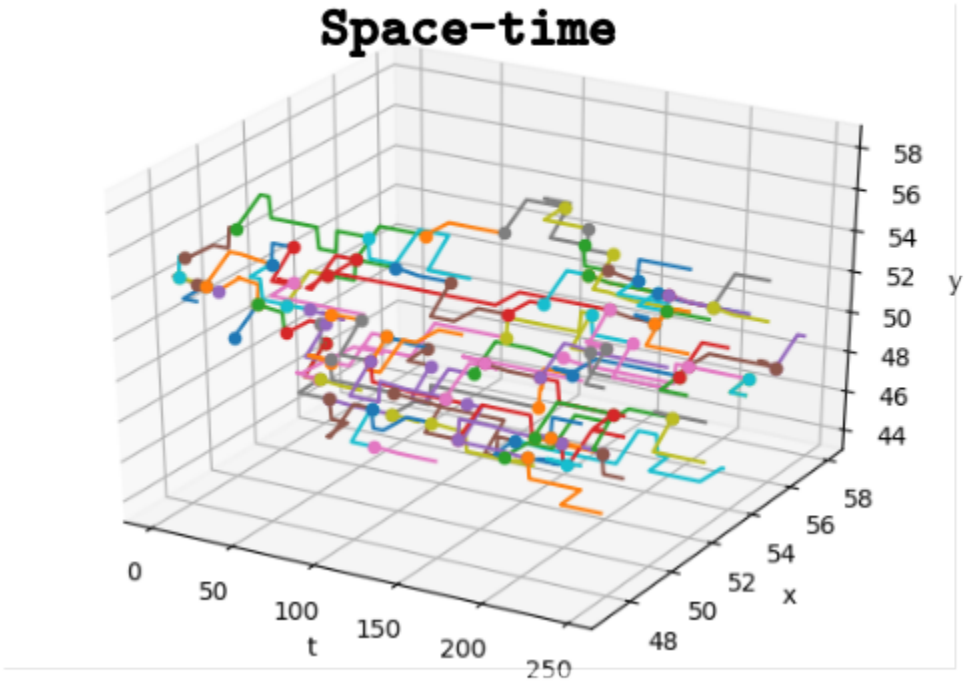
- modindex

CHAPTER 3

CellSystem

This was created to simulate cancer growth, taking into account nutrients and cell migration while allowing to track mutations, cell division and cell position history to study tumour phylogeny reconstruction algorithms.


Mutations
Ancestry



3.1 Installation

You can install it from PyPI:

```
$ pip install cellsystem
```

3.2 Example

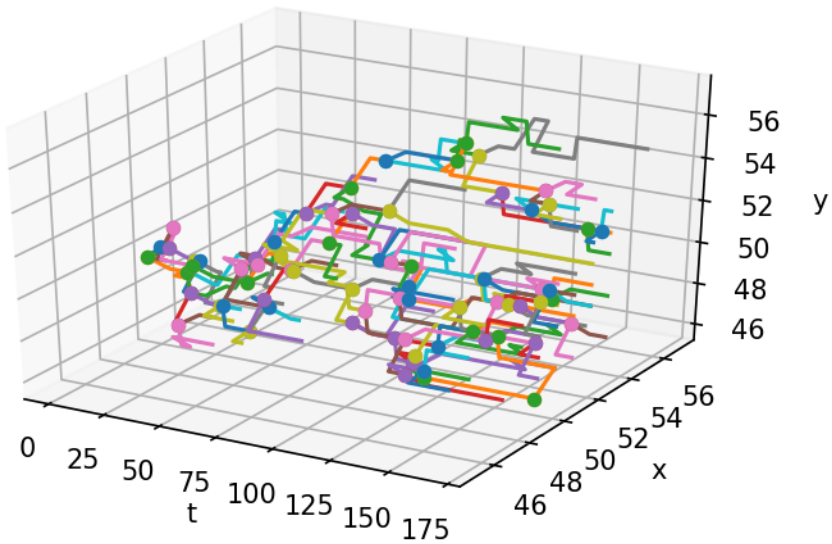
A use case integrated in the repository:

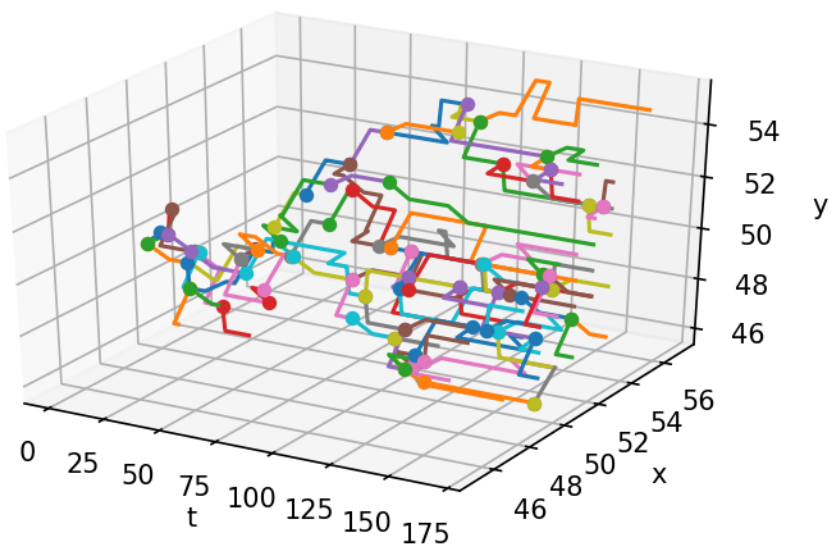
```
>>> from cellsystem import *

# The cell system will simulate cell growth
# while tracking the steps in that process.
>>> system = CellSystem(grid_shape=(100, 100))

# Initialize the first cell
# in the middle of the grid
>>> system.seed()


New cell 0 added @ (50, 50)


# Take 35 steps forward in time
>>> system.run(steps=30)
```

(continues on next page)

(continued from previous page)

```
Cell no. 0 mutating @ site (50, 50) (father None)
         Initial mutations: []
                     Initial genome: AAAAAAAAAA
         Final mutations: [(4, 'G')]
                     Final genome: AAAAGAAAAA
Cell no. 0 dividing @ (50, 50)
    New cells: 1 @ (49, 50) and 2 @ (50, 51)
Cell no. 2 dividing @ (50, 51)
    New cells: 3 @ (51, 52) and 4 @ (51, 52)
Cell no. 4 mutating @ site (51, 52) (father 2)
         Initial mutations: [(4, 'G')]
                     Initial genome: AAAAGAAAAA
         Final mutations: [(4, 'G'), (7, 'A')]
                     Final genome: AAAAGAAAAA
Cell no. 1 death @ site (49, 50) (father None)
Cell no. 3 death @ site (51, 52) (father 2)
Cell no. 4 mutating @ site (51, 52) (father 2)
         Initial mutations: [(4, 'G'), (7, 'A')]
                     Initial genome: AAAAGAAAAA
         Final mutations: [(4, 'G'), (7, 'A'), (2, 'T')]
                     Final genome: AATAGAAAAA
Cell no. 4 migrating from site (51, 52) (father 2)
     New site: (50, 52)
...
...
...


# Prepare to explore the simulation logs
>>> history = system['log']


# First, let's see the cells' evolution in time and space!
>>> history.worldlines().show()

# Remove the cells that died somewhere along the way
>>> history.worldlines(prune_death=True).show()
```



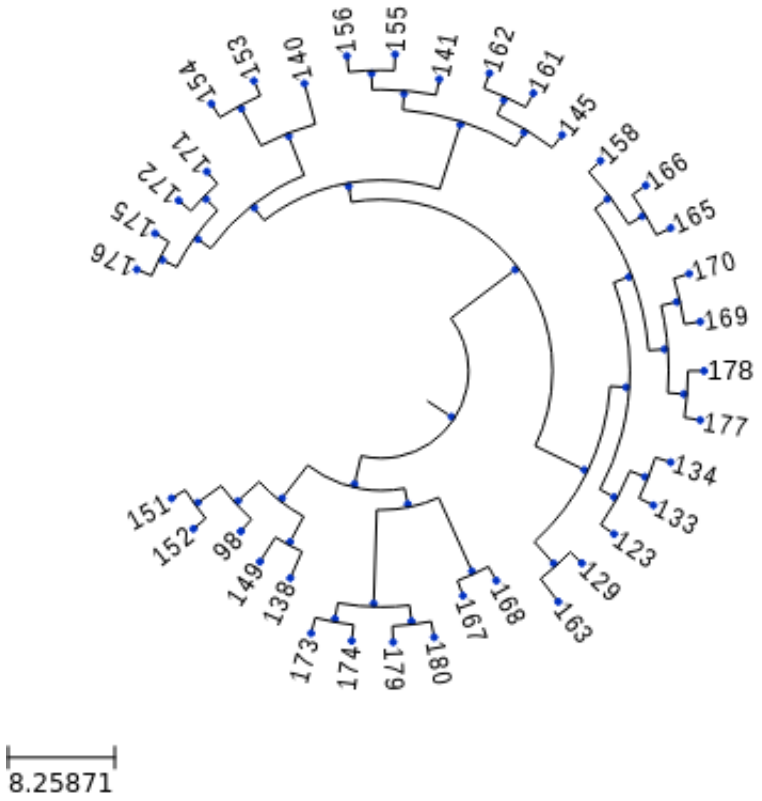



```
>>> tree_style = {'show_leaf_name' : True,
...               'mode' : 'c',         # Circular
...               'arc_start' : -135,   # Degrees
...               'arc_span' : 270 }    # Degrees also


# Lookup the tree formed by cellular division
```

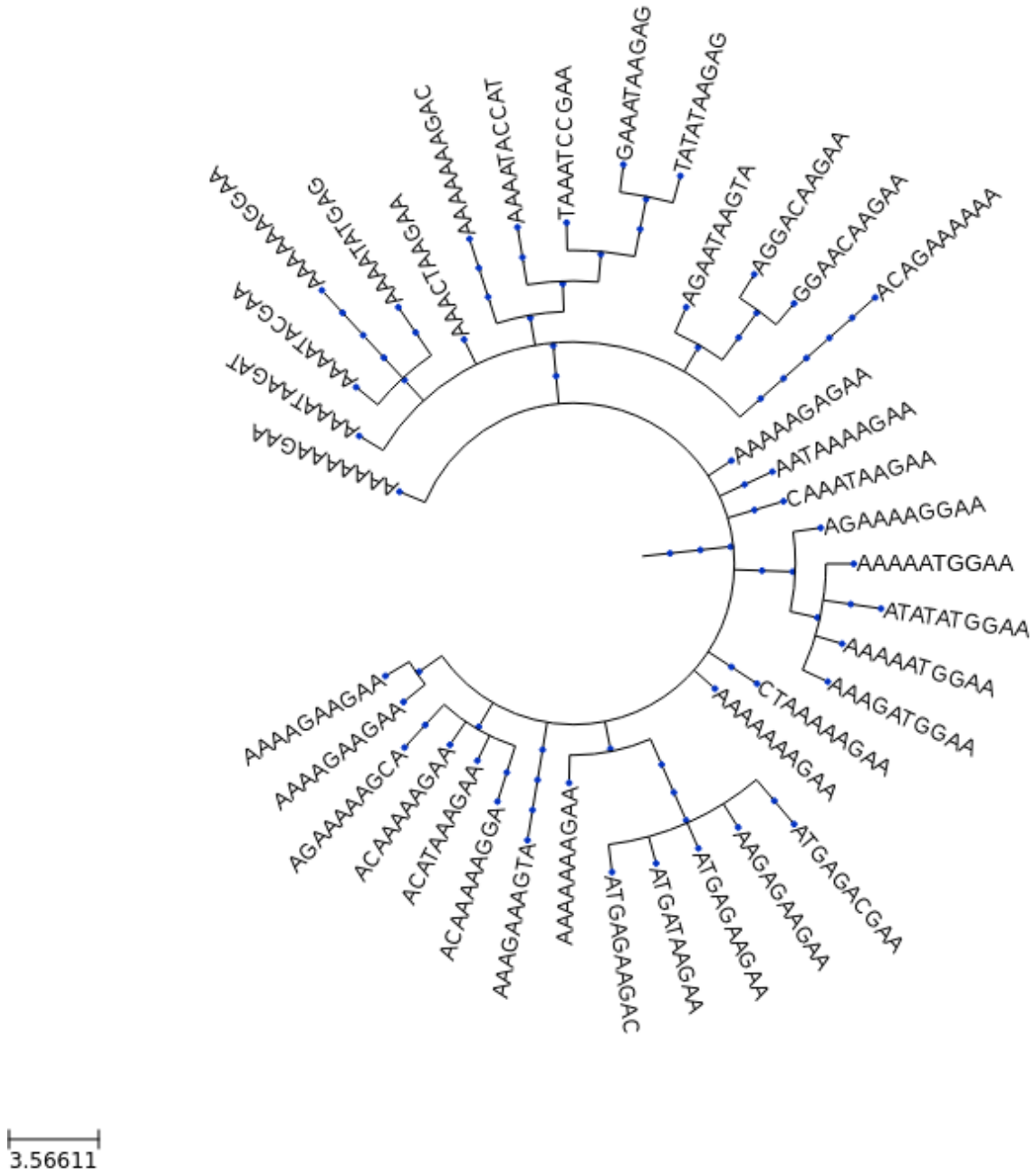
(continues on next page)

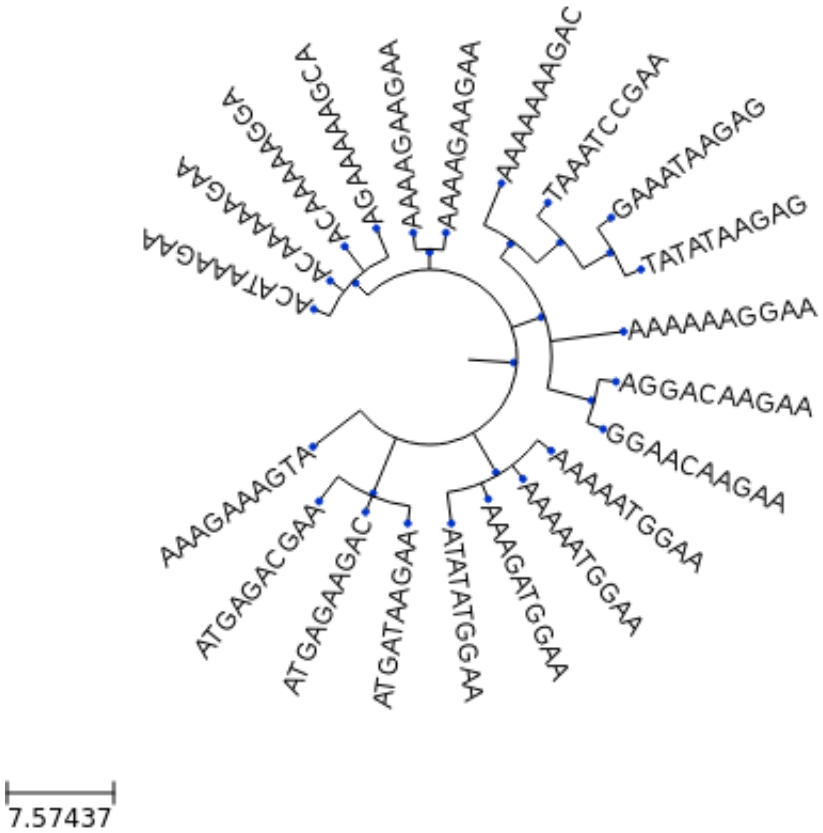
(continued from previous page)

```
>>> history.ancestry().show(styling=tree_style)

# Now, remove cells that are no longer alive
>>> history.ancestry(prune_death=True).show(styling=tree_style)
```




```
# Now, check out the tree formed by the mutations
>>> history.mutations().show(styling=tree_style)

# Remove genomes with no living representatives.
>>> history.mutations(prune_death=True).show(styling=tree_style)
```


AAAAAAAGAA
AAAATAAGAT
AAAATACGAA
AAAAAAGGAA
AAAATATGAG
AAAACTAGAA
AAAAAAAGAC
AAAATACCAT
TAAATCCGAA
GAAATAAGAG
TATATAAGAG
AGAATAAGTA
AGGACAAGAA
GGAACAAGAA
ACAGAAAAAA
AAAAAGAGAA
AATAAAAGAA
CAAATAAGAA
AGAAAAGGAA
AAAAATGGAA
ATATATGGAA
AAAAATGGAA
AAAGATGGAA
CTAAAAAGAA
AAAAAAAGAA
ATGAGACGAA
AAGAGAAGAA
ATGAGAAGAA
ATGATAAGAA
ATGAGAAGAC
AAAAAAAGAA
AAAGAAAGTA
ACAAAAAGGA
ACATAAAGAA
ACAAAAAGAA
AGAAAAAGCA
AAAAGAAGAA
AAAAGAAGAA
3.56611



For more examples and usage, please refer to the [Wiki](wikigoeshere.com).

3.3 Meta

Author: Ad115 - Github – a.garcia230395@gmail.com

Distributed under the MIT license. See LICENSE for more information.

Warning: The project is still in alpha stage, so the API is just stabilizing and may change in the near future. This also means that if you want to contribute, now is the right moment to make important change suggestions ;D

3.4 Contributing

1. Check for open issues or open a fresh issue to start a discussion around a feature idea or a bug.
2. Fork the repository on GitHub to start making your changes to a feature branch, derived from the **master** branch.
3. Write a test which shows that the bug was fixed or that the feature works as expected.
4. Send a pull request and bug the maintainer until it gets merged and published.

Python Module Index

C

Index

M

N

P

R

S

T

U

W