cellsium

Release 1.0.0

Christian C. Sachs

Jun 02, 2023

CONTENTS:

CHAPTER

ONE

CELLSIUM README

CellSium - *Cell Si*mulator for *micro*fluidic *m*icrocolonies

Fig. 1: CellSium example simulation result

1.1 Front Matter

CellSium is a cell simulator developed for the primary application of generating realistically looking images of bacterial microcolonies, which may serve as ground truth for machine learning training processes.

1.1.1 Publication

If you use CellSium within scientific research, we ask you to cite our publication:

> Sachs CC, Ruzaeva K, Seiffarth J, Wiechert W, Berkels B, Nöh K (2022) CellSium: versatile cell simulator for microcolony ground truth generation Bioinformatics Advances, Volume 2, Issue 1, 2022, vbac053, doi: 10.1093/bioadv/vbac053

It is available on the *Bioinformatics Advances* homepage at *DOI: 10.1093/bioadv/vbac053 <https://doi.org/10.1093/bioadv/vbac053>*.

1.1.2 Documentation

The documentation to CellSium can be built using Sphinx, or be found readily built at Read the Docs.

1.1.3 License

CellSium is available under the BSD license (see LICENSE.rst / license section).

1.2 Installation

1.2.1 Installation using pip

CellSium can be installed via pip, ideally create and activate an environment beforehand to install CellSium in.

```
> python -m pip install cellsium
```

1.2.2 Installation using conda

CellSium is available in the modsim Anaconda channel as well, using packages from the conda-forge channel. It can be installed with the following commands:

```
> conda install -c modsim -c conda-forge -y cellsium
```

1.3 Usage

Once installed, run CellSium via `python -m cellsium`, specifying the desired entrypoint and options, such as outputs. CellSium is built modular, various output modules can be activated simultaneously. To get an overview of the available options, use the `--help` switch. Furthermore, the main mode of setting tunable parameters are so called *tunables*, which can be set from the command line using the `-t` switches. A list of tunables can be shown using the `--tunables-show` argument.

```
> python -m cellsium --help
usage: __main__.py [-v] [-q] [-c CELL] [-p] [-w] [-o OUTPUT] [-h] [-m MODULE]
               [--Output {COCOOutput,CsvOutput,FluorescenceRenderer,GenericMaskOutput,
↪JsonPickleSerializer,MeshOutput,NoisyUnevenIlluminationPhaseContrast,
↪PhaseContrastRenderer,PlainRenderer,PlotRenderer,QuickAndDirtyTableDumper,SvgRenderer,
↪TiffOutput,TrackMateXML,UnevenIlluminationPhaseContrast,YOLOOutput}]
               [--PlacementSimulation {Box2D,Chipmunk,NoPlacement}] [-t TUNABLE] [--
↪tunables-show] [--tunables-load TUNABLES_LOAD] [--tunables-save TUNABLES_SAVE]

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output-file OUTPUT
  -w, --overwrite
  -p, --prefix
  -c CELL, --cell CELL
  -q, --quiet
```

(continues on next page)

(continued from previous page)

```
  -v, --verbose
  -m MODULE, --module MODULE
  --Output {COCOOutput,CsvOutput,FluorescenceRenderer,GenericMaskOutput,
↪JsonPickleSerializer,MeshOutput,NoisyUnevenIlluminationPhaseContrast,
↪PhaseContrastRenderer,PlainRenderer,PlotRenderer,QuickAndDirtyTableDumper,SvgRenderer,
↪TiffOutput,TrackMateXML,UnevenIlluminationPhaseContrast,YOLOOutput}
  --PlacementSimulation {Box2D,Chipmunk,NoPlacement}
  -t TUNABLE, --tunable TUNABLE
  --tunables-show
  --tunables-load TUNABLES_LOAD
  --tunables-save TUNABLES_SAVE
```

You can for example run a default simulation by just starting CellSium, the results will be shown interactively using matplotlib:

```
> python -m cellsium
```

For more in-depth usage examples, please see the examples section of the documentation.

1.4 Docker

An alternative to installing CellSium locally is running it via Docker. To run CellSium without interactive (GUI) elements, the following Docker command can be used, with parameters to CellSium being appended.

```
> docker run --tty --interactive --rm --volume `pwd`:/data --user `id -u` ghcr.io/modsim/
↪cellsium
```

To use interactive (GUI) elements such as the `PlotRenderer`, an X server must be reachable; under Linux the following command can be used:

```
> docker run --tty --interactive --rm --volume `pwd`:/data --user `id -u` --env DISPLAY=
↪$DISPLAY --volume /tmp/.X11-unix:/tmp/.X11-unix ghcr.io/modsim/cellsium
```

CHAPTER

TWO

LICENSE

CHAPTER

THREE

EXAMPLES

The following sections contain some usage examples of CellSium.

3.1 Creating random training data

The core mode of operation is the creation of ground truth data as training data for machine learning/deep learning applications. To this end, CellSium contains two output modes specifically tailored to produce outputs for common deep learning based object detectors/instance segmentation toolkits: The COCO and YOLO format. CellSium can as well just output binary masks along the images for use with other learning tools.

For example, the following command will random cell images, and output three datasets:

```
> python -m cellsium training \
    -t TrainingDataCount=64 \
    -t TrainingCellCount=512 \
    -t TrainingImageWidth=512 \
    -t TrainingImageHeight=512 \
    -t Calibration=0.0905158 \
    -t ChipmunkPlacementRadius=0.01 \
    -o training \
    --Output COCOOutput \
    --Output YOLOOutput \
    --Output GenericMaskOutput \
    -p
```

Note how the main mode of configuration of CellSium are *tunables*, these tunable parameters are set using the `-t` argument, followed by `Name=value`. The tunables are explained in the documentation, and can be listed via `--tunables-show` as well.

In this example, the output of 64 images of 512x512 size are requested, setting the pixel calibration to 0.0905158 µm per pixel. `ChipmunkPlacementRadius` configures the physical placement and yields denser colonies. The name of the output files/directories is specified using `-o`.

The outputs of CellSium are modular. In this example, the `COCOOutput`, `YOLOOutput`, and `GenericMaskOutput` are enabled. As to prevent name clashes and allow easy coexistence, the `-p:code:` switch enables prefixing of the output names with the name of the respective output module.

Once the exmaple has run, the directories `COCOOutput-training`, `GenericMaskOutput-training`, and `YOLOOutput-training` have been created, with examples of the `GenericMaskOutput` shown.

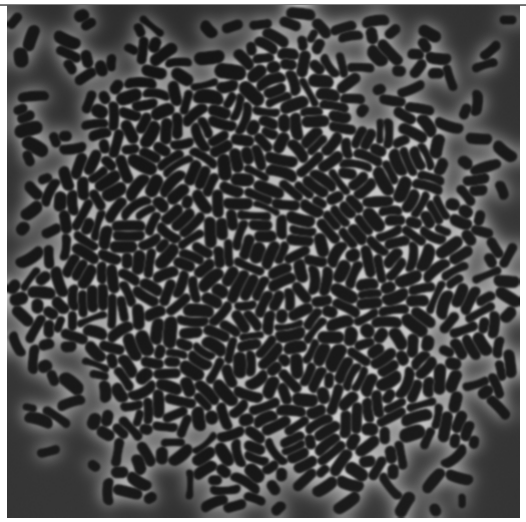

Fig. 1: `GenericMaskOutput-training/images/000000000000.png`

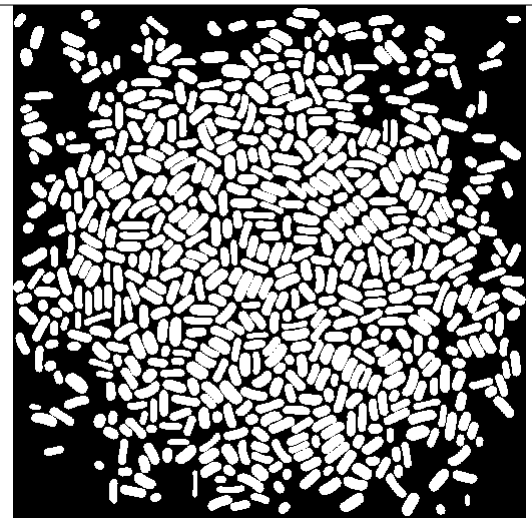

Fig. 2: `GenericMaskOutput-training/masks/000000000000.png`

3.2 Creating a timelapse simulation

CellSium was originally developed to create time lapse simulations to create realistic microcolonies, which can serve as input data e.g., for simulations based on the geometry or the training and validation of tracking algorithms. To run a simulation, ste up the desired outputs and tunables as explained, and use the `simulate` subcommand.

```
> python -m cellsium simulate \
    -o simulate \
    --Output GenericMaskOutput \
    --Output TiffOutput \
    -p
```

In this case, a microcolony will be simulated, a time lapse TIFF stack as well as mask output generated. Shown are three example images:

Fig. 3: GenericMaskOutput-training/images/000000000000.png

Fig. 4: GenericMaskOutput-training/images/000000000016.png

3.3 Adding a custom cell model

In the previous examples, the standard (sizer) cell model was used. However, the modular nature of CellSium makes it easy to integrate a custom cell model. In this example, the sizer model will be defined externally, so it can be more easily changed, and to showcase its difference, the easter egg square geometry will be applied:

Listing 1: square.py

```
from cellsium.model import assemble_cell, SimulatedCell, h_to_s, Square


class SquareCellModel(SimulatedCell):
    @staticmethod
    def random_sequences(sequence):
        return dict(elongation_rate=sequence.normal(1.5, 0.25))  # µm·h¹

    def birth(self, parent=None, ts=None) -> None:
        self.elongation_rate = next(self.random.elongation_rate)
        self.division_time = h_to_s(1.0)

    def grow(self, ts) -> None:
        self.length += self.elongation_rate * ts.hours

        if ts.time > (self.birth_time + self.division_time):
            offspring_a, offspring_b = self.divide(ts)
            offspring_a.length = offspring_b.length = self.length / 2


Cell = assemble_cell(SquareCellModel, Square)
```

The custom model can be specified using the `-c` switch, specifying either an importable Python module, or the path of a Python file. If no class name is specified after the `:` colon, CellSium will attempt to import a class named `Cell` from the file/module.

```
> python -m cellsium simulate \
    -o square \
    --Output GenericMaskOutput \
    -c square.py:Cell \
    -p
```

CellSium cell objects are Python objects. They are built lending from OOP principles, using mixins in a very flexible way to join various properties. To gain deeper insights how to implement and alter cellular behavior or rendering, it is best to study the source code of CellSium.

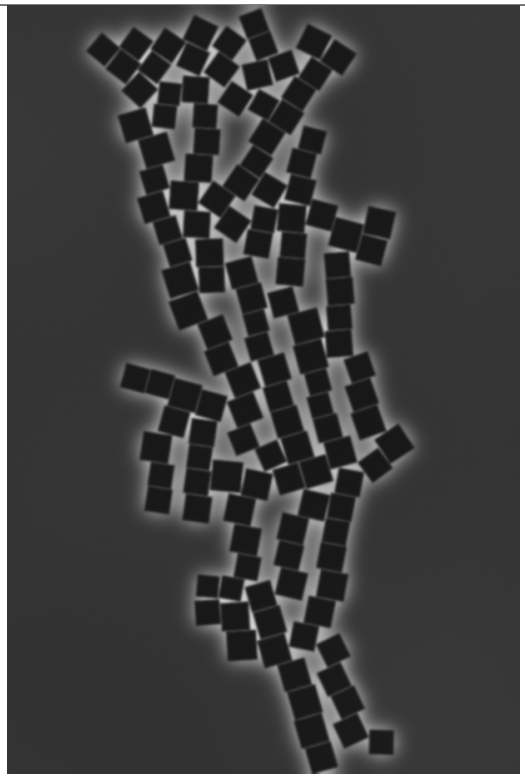

Fig. 6: GenericMaskOutput-square/images/000000000033.png

3.4 Jupyter Notebook Embedding Example

In this example, CellSium is embedded in a Jupyter notebook to interactively run small simulations.

First, the necessary modules are imported:

```
# plotting
from matplotlib_inline.backend_inline import set_matplotlib_formats
set_matplotlib_formats('svg')
from matplotlib import pyplot

# general
from functools import partial

# RRF is the central random helper, used for seeding
from cellsium.random import RRF
# the model parts the new model is built upon
from cellsium.model import PlacedCell, SimulatedCell, assemble_cell
# the functions to actually perform the simulation
from cellsium.cli.simulate import perform_simulation, initialize_cells, h_to_s, s_to_h
# the PlotRenderer as it embeds nicely in Jupyter
from cellsium.output.plot import PlotRenderer
```

For the example, a model is defined directly within a Jupyter cell:

```
class SizerCell(SimulatedCell):
    @staticmethod
    def random_sequences(sequence):
        return dict(elongation_rate=sequence.normal(1.5, 0.25))  # µm·h¹

    def birth(
        self, parent=None, ts=None
    ) -> None:
        self.elongation_rate = next(self.random.elongation_rate)
        self.division_time = h_to_s(0.5)  # fast division rate

    def grow(self, ts):
        self.length += self.elongation_rate * ts.hours

        if ts.time > (self.birth_time + self.division_time):
            offspring_a, offspring_b = self.divide(ts)
            offspring_a.length = offspring_b.length = self.length / 2
```

```
# seed the random number generator
RRF.seed(1)

# perform_simulation returns an iterator which will indefinitely yield timesteps

simulation_iterator = perform_simulation(
    setup=partial(
        initialize_cells,
        count=1,
        cell_type=assemble_cell(SizerCell, placed_cell=PlacedCell)),
    time_step=30.0 * 60.0
)

# we step thru the first 5 of them ...
for _, ts in zip(range(5), simulation_iterator):
    # ... and plot them
    PlotRenderer().output(world=ts.world)
    pyplot.title("Simulation output at time=%.2fh" % (s_to_h(ts.time)))
    pyplot.show()
    # we have access to the Cell objects as well
    print(repr(ts.world.cells))
```

```
[Cell(angle=2.3294692309443428, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=0.0, division_time=1800.
↪0, elongation_rate=1.5981931788501658, id_=1, length=2.872894940610261, lineage_
↪history=[0], parent_id=0, position=[17.854225629902448, 28.67645476278087], width=0.
↪8960666826747447)]
```

```
[Cell(angle=2.3294692309443428, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=3600.0, division_
↪time=1800.0, elongation_rate=1.4017119040732795, id_=2, length=1.835995765017672,␣
```

(continues on next page)

(continued from previous page)

```
↪lineage_history=[0, 1], parent_id=1, position=[18.485770454254308, 28.010218132802386],
↪ width=0.8960666826747447), Cell(angle=2.3294692309443428, bend_lower=-0.
↪016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.09614075306364522,
↪ birth_time=3600.0, division_time=1800.0, elongation_rate=1.7743185975635118, id_=3,␣
↪length=1.835995765017672, lineage_history=[0, 1], parent_id=1, position=[17.
↪22268080555059, 29.342691392759356], width=0.8960666826747447)]
```

```
[Cell(angle=2.3360626329715437, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=3600.0, division_
↪time=1800.0, elongation_rate=1.4017119040732795, id_=2, length=2.5368517170543115,␣
↪lineage_history=[0, 1], parent_id=1, position=[18.756872759489948, 27.72423023484169],␣
↪width=0.8960666826747447), Cell(angle=2.3228640880321927, bend_lower=-0.
↪016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.09614075306364522,
↪ birth_time=3600.0, division_time=1800.0, elongation_rate=1.7743185975635118, id_=3,␣
↪length=2.7231550637994277, lineage_history=[0, 1], parent_id=1, position=[16.
↪951578500314948, 29.62867929072005], width=0.8960666826747447)]
```

```
[Cell(angle=2.352798559960036, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=7200.0, division_
↪time=1800.0, elongation_rate=0.8317980611709823, id_=4, length=1.6188538345454755,␣
↪lineage_history=[0, 1, 2], parent_id=2, position=[19.593155424028517, 26.
↪85358455987381], width=0.8960666826747447), Cell(angle=2.3416190585127383, bend_lower=-
↪0.016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.
↪09614075306364522, birth_time=7200.0, division_time=1800.0, elongation_rate=1.
↪2231841085165691, id_=5, length=1.6188538345454755, lineage_history=[0, 1, 2], parent_
↪id=2, position=[18.471310193545275, 28.02153383690875], width=0.8960666826747447),␣
↪Cell(angle=2.314495314084782, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=7200.0, division_
↪time=1800.0, elongation_rate=1.4247110511570817, id_=6, length=1.8051571812905918,␣
↪lineage_history=[0, 1, 3], parent_id=3, position=[17.289158390942983, 29.
↪25243321304336], width=0.8960666826747447), Cell(angle=2.304654468034007, bend_lower=-
↪0.016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.
↪09614075306364522, birth_time=7200.0, division_time=1800.0, elongation_rate=1.
↪8079038604401219, id_=7, length=1.8051571812905918, lineage_history=[0, 1, 3], parent_
↪id=3, position=[16.06327851109302, 30.578267441297573], width=0.8960666826747447)]
```

```
[Cell(angle=2.3820603181599442, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=7200.0, division_
↪time=1800.0, elongation_rate=0.8317980611709823, id_=4, length=2.0347528651309665,␣
↪lineage_history=[0, 1, 2], parent_id=2, position=[20.228643748180197, 26.
↪199313752340714], width=0.8960666826747447), Cell(angle=2.3526475982043165, bend_
↪lower=-0.016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.
↪09614075306364522, birth_time=7200.0, division_time=1800.0, elongation_rate=1.
↪2231841085165691, id_=5, length=2.23044588880376, lineage_history=[0, 1, 2], parent_
↪id=2, position=[18.75284683312141, 27.731672298371393], width=0.8960666826747447),␣
↪Cell(angle=2.2968665429831496, bend_lower=-0.016261360844027475, bend_overall=-0.
↪04126808565523994, bend_upper=-0.09614075306364522, birth_time=7200.0, division_
```

(continues on next page)

(continued from previous page)

```
↪time=1800.0, elongation_rate=1.4247110511570817, id_=6, length=2.5175127068691325,␣
↪lineage_history=[0, 1, 3], parent_id=3, position=[17.088954083136315, 29.
↪417294148769457], width=0.8960666826747447), Cell(angle=2.2660140670444004, bend_
↪lower=-0.016261360844027475, bend_overall=-0.04126808565523994, bend_upper=-0.
↪09614075306364522, birth_time=7200.0, division_time=1800.0, elongation_rate=1.
↪8079038604401219, id_=7, length=2.709109111510653, lineage_history=[0, 1, 3], parent_
↪id=3, position=[15.346457855171874, 31.35753885164192], width=0.8960666826747447)]
```

CHAPTER

FOUR

CELLSIUM PACKAGE

CellSium - _Cell_ _Si_mulator for _micro_fluidic _m_icrocolonies

4.1 cellsium.cli package

CLI package, home to the individual entry points

`cellsium.cli.`**`Cell`**

alias of *SizerCell*

class `cellsium.cli.`**`SizerCell`**(***kwargs*)

Bases: *PlacedCell*, *SizerCell*

Cell.

class `cellsium.cli.`**`TimerCell`**(***kwargs*)

Bases: *PlacedCell*, *TimerCell*

Cell.

`cellsium.cli.`**`add_output_prefix`**(*output_name: str*, *output:* Output) → str

Adds an prefix to an output filename.

Parameters

- **`output_name`** – Output name
- **`output`** – Output object

Returns

Name

`cellsium.cli.`**`initialize_cells`**(*simulator:* Simulator, *count: int = 1*, *cell_type: Optional[*PlacedCell*] = None*, *sequence: Optional[Any] = None*) → *Simulator*

Initialize cells and add them to a simulator.

Parameters

- **`simulator`** – Simulator to add cells to.
- **`count`** – Count of cells to generate
- **`cell_type`** – cell type to use
- **`sequence`** – Random number sequence to use

Returns

Simulator

`cellsium.cli.initialize_simulator()` → *Simulator*

Constructor helper for a simulator.

Returns
Simulator instance

4.1.1 cellsium.cli.render package

Rendering CLI utility, render a simulation state saved using jsonpickle.

`cellsium.cli.render.subcommand_argparser(parser: ArgumentParser)` → None

Handle the argument parser for the 'render' subcommand.

Parameters
parser – Argument parser

Returns
None

`cellsium.cli.render.subcommand_main(args: Namespace)` → None

Entry point for the 'render' subcommand.

Parameters
args – Pre-parsed arguments

Returns
None

4.1.2 cellsium.cli.simulate package

Simulation CLI entrypoint.

class `cellsium.cli.simulate.BoundariesFile(*args, **kwargs)`

Bases: `Tunable`

Boundaries file (in DXF format) to add boundaries/geometrical constraints

default: str = ''

value = ''

class `cellsium.cli.simulate.BoundariesScaleFactor(*args, **kwargs)`

Bases: `Tunable`

Scale factor for the boundaries

default: float = 1.0

value = 1.0

class `cellsium.cli.simulate.SimulationDuration(*args, **kwargs)`

Bases: `Tunable`

Time (simulated) the simulation should run

default: float = 12.0

value = 12.0

class cellsium.cli.simulate.**SimulationOutputFirstState**(**args*, ***kwargs*)

Bases: Tunable

Whether to output the first state

default: bool = False

value = False

class cellsium.cli.simulate.**SimulationOutputInterval**(**args*, ***kwargs*)

Bases: Tunable

Time intervals (simulated) at which an output should be written

default: float = 0.25

value = 0.25

class cellsium.cli.simulate.**SimulationTimestep**(**args*, ***kwargs*)

Bases: Tunable

Time step at which the simulation state should be calculated

default: float = 0.016666666666666666

value = 0.016666666666666666

cellsium.cli.simulate.**add_boundaries_from_dxf**(*file_name: str*, *simulator:* Simulator, *scale_factor: float = 1.0*) → None

Add boundaries from a DXF file. Supported are LWPolyline and Polyline objects.

Parameters

- **file_name** – dxf file name
- **simulator** – Simulator instance to add the boundaries to
- **scale_factor** – Scale factor for the geometry

Returns
None

cellsium.cli.simulate.**add_boundaries_from_tunables**(*simulator:* Simulator) → *Simulator*

Add boundaries to a simulator if the appropriate tunables are set.

Parameters
simulator – Simulator

Returns
Simulator

cellsium.cli.simulate.**compose**(**args*)

Compose callable args.

Parameters
args – Args

Returns
Callable

`cellsium.cli.simulate.initialize_output_times_from_tunables()`

Initialize the duration, output_interval and last_output variables, from tunables.

Returns
duration, output_interval, last_output

`cellsium.cli.simulate.measure_duration(iterator: Iterator[T])` → Iterator[Tuple[float, T]]

Measure the (wall clock) time it takes to step forward the iterator, yielding the duration and iterator value.

Parameters
iterator – Iterator

Returns
Iterator of duration and iterator value

`cellsium.cli.simulate.perform_outputs(world: World, simulation_time: float, outputs: Iterable[Output], output_name: Optional[str] = None, overwrite: bool = False, prefix: bool = False, output_count: int = 0)` → None

Performs the output operations configured.

Parameters

- **world** – World to output
- **simulation_time** – Simulation timepoint
- **outputs** – Outputs
- **output_name** – Name to output to
- **overwrite** – Whether to overwrite
- **prefix** – Whether to prefix the outputs with the name of the Output type
- **output_count** – The count of already outputted timesteps

Returns
None

`cellsium.cli.simulate.perform_simulation(simulator: Optional[Simulator] = None, setup: Optional[Callable[[Simulator], Simulator]] = None, time_step: float = 1.0)` → Iterator[*Timestep*]

Simple entrypoint to perform simulations. Will either accept a simulator instance or create one, call the setup callback on it if set, and then yield timesteps time_step apart indefinitely.

Parameters

- **simulator** – Simulator
- **setup** – Setup callable
- **time_step** – Time step

Returns
Iterator of Timestep instances

`cellsium.cli.simulate.prepare_output_name(output_name: str, output: Output, prefix: str)` → str

Prepare an output name.

Parameters

- **output_name** – Output name
- **output** – Output object

- **prefix** – Prefix

Returns
Output name

`cellsium.cli.simulate.`**`subcommand_main`**(*args: Namespace*) → None

Entry point for the 'simulate' subcommand.

Parameters
args – pre-parsed arguments

Returns
None

4.1.3 cellsium.cli.training package

Training Generation CLI entrypoint.

class `cellsium.cli.training.`**`TrainingCellCount`**(**args*, ***kwargs*)

Bases: `Tunable`

Cells to add to training samples

default: int = 32

value = 32

class `cellsium.cli.training.`**`TrainingDataCount`**(**args*, ***kwargs*)

Bases: `Tunable`

Training samples to generate

default: int = 16

value = 16

class `cellsium.cli.training.`**`TrainingImageHeight`**(**args*, ***kwargs*)

Bases: `Tunable`

Image height in pixels of training images

default: int = 128

value = 128

class `cellsium.cli.training.`**`TrainingImageWidth`**(**args*, ***kwargs*)

Bases: `Tunable`

Image width in pixels of training images

default: int = 128

value = 128

`cellsium.cli.training.`**`subcommand_main`**(*args: Namespace*) → None

Entry point for the 'training' subcommand.

Parameters
args – pre-parsed arguments

Returns
None

4.1.4 cellsium.cli.cli module

CLI entrypoint.

`cellsium.cli.cli.load_class_from_module(module_class: str, default_class_name: str) → type`

`cellsium.cli.cli.main(args: Optional[Iterable[str]] = None) → Optional[int]`

Main entrypoint of the script, will redirect to various sub-scripts.

Parameters
args – arguments, if not specified will be taken from sys.argv

Returns
The return code of the individual subcommand

`cellsium.cli.cli.parse_arguments_and_init(args: Iterable[str], parser_callback: Optional[Callable[[ArgumentParser], None]] = None) → Namespace`

Basic setup (i.e. logging) and argument parsing.

Parameters

- **args** – Arguments
- **parser_callback** – Additional callback to configure the argument parser

Returns
Parsed arguments

4.2 cellsium.geometry package

Various geometry handling functions.

`cellsium.geometry.add_empty_third_dimension(array: ndarray) → ndarray`

Adds an empty third dimension.

Parameters
array – 2D Array

Returns
3D Array

`cellsium.geometry.circle_segment(radius: float, start: ndarray, stop: ndarray, interval: float = 0.1, minimum_times: int = 10, times: Optional[int] = None) → ndarray`

Rasters a circle segment from start to stop with a radius radius.

Parameters

- **radius** – Radius
- **start** – Start point
- **stop** – Stop point
- **interval** – Interval
- **minimum_times** – Minimal count of points to put between start and stop
- **times** – Alternatively: count of points to place

Returns
Coordinates

`cellsium.geometry.line(start: ndarray, stop: ndarray, interval: float = 0.1, minimum_times: int = 10, times: Optional[int] = None) → ndarray`

Rasters a line from start to stop with points at interval interval.

Parameters

- **start** – Start point
- **stop** – Stop point
- **interval** – Interval
- **minimum_times** – Minimal count of points to put between start and stop
- **times** – Alternatively: count of points to place

Returns
Coordinates

`cellsium.geometry.parabolic_deformation(array: ndarray, factor: float) → ndarray`

Deform array by a parabola.

Parameters

- **array** – Coordinates
- **factor** – Factor

Returns
Deformed coordinates

`cellsium.geometry.rotate(data: ndarray, angle: float) → ndarray`

Rotates data by angle.

Parameters

- **data** – Coordinates
- **angle** – Angle

Returns
Rotated coordinates

`cellsium.geometry.rotate3d(data: ndarray, angle: float, axis_vector: ndarray) → ndarray`

Rotates data within 3D space around axis_vector.

Parameters

- **data** – Coordinates
- **angle** – Angle
- **axis_vector** – Axis vector of the rotation

Returns
Rotated points

`cellsium.geometry.rotate_and_mesh(points: ndarray, steps: int = 16, clean: bool = True, close_ends: bool = True) → Tuple[ndarray, ndarray]`

Produces a solid of revolution.

Parameters

- **points** – Coordinates
- **steps** – Steps of the revolution
- **clean** – Whether to clean the data beforehand
- **close_ends** – Whether to close the ends

Returns
Tuple(Array of Points, Triangle Indices)

`cellsium.geometry.shift(data: ndarray, vector: ndarray)` → ndarray

Shifts coordinates.

Parameters

- **data** – Coordinates
- **vector** – Shift vector

Returns
Shifted coordinates

4.3 cellsium.model package

Cell model package.

class `cellsium.model.AutoMesh3D`

Bases: *Shape3D*

Mixin adding automatic solid-of-revolution generation.

points3d_on_canvas(*steps: int = 16, simplify: bool = False*) → Tuple[ndarray, ndarray]

raw_points3d(*steps: int = 16, simplify: bool = False*) → Tuple[ndarray, ndarray]

class `cellsium.model.BentRod`

Bases: *RodShaped*

Bent rod shaped cell geometry.

bend(*points: ndarray*) → ndarray

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

raw_points3d(*steps: int = 16, simplify: bool = False*) → Tuple[ndarray, ndarray]

class `cellsium.model.CellGeometry`

Bases: *WithAngle*, *WithPosition*, *AutoMesh3D*

Cell geometry base by combining multiple mixins.

points_on_canvas() → ndarray

class cellsium.model.**Coccoid**

Bases: *Shape*

Coccoid (spherical) cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.**Copyable**

Bases: object

Mixin for copyable objects.

copy() → *Copyable*

class cellsium.model.**Ellipsoid**

Bases: *Coccoid*

Ellipsoid cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.**IdCounter**

Bases: object

Id provider singleton class.

id_counter: int = 0

classmethod next_cell_id() → int

classmethod reset() → None

class cellsium.model.**InitializeWithParameters**(***kwargs*)

Bases: object

Mixin for objects with defaults.

class cellsium.model.**PlacedCell**(***kwargs*)

Bases: *WithLineageHistory*, *WithLineage*, *WithTemporalLineage*, *WithProperDivisionBehavior*, *InitializeWithParameters*, *Copyable*, *Representable*, *WithRandomSequences*, *RandomWidthLength*, *RandomBentRod*, *RandomPosition*, *RandomAngle*, *CellGeometry*, *BentRod*

class cellsium.model.**Rectangle**

Bases: *Shape*

Rectangular cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.**Representable**

Bases: object

Mixins for adding a repr implementation.

class cellsium.model.**RodShaped**

Bases: *Shape*

Rod shaped cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

rod_raw_points(*simplify: bool = False*) → Tuple[ndarray, ndarray, ndarray, ndarray]

class cellsium.model.**Shape**

Bases: object

Base class for implementing cell shapes.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.**Shape3D**

Bases: *Shape*

Base class for implementing 3D cell shapes.

raw_points3d(*steps: int = 32, simplify: bool = False*) → ndarray

class cellsium.model.**SimulatedCell**

Bases: object

Base class for simulated cells, allowing for division behavior.

birth(*parent: Optional[SimulatedCell] = None, ts: Optional[Timestep] = None*) → None

Called when a cell is "born".

Parameters

- **parent** – Parent cell
- **ts** – Timestep

Returns
None

divide(*ts: Timestep*) → Iterable[*SimulatedCell*]

Called when a cell should divide, creates the daughter cells.

Parameters
ts – Timestep

Returns
None

grow(*ts: Timestep*) → None

Called each timestep to grow cell.

Parameters
ts – Timestep

Returns
None

step(*ts: Timestep*) → None

Timestep function of the cell object, called by the simulator.

Parameters
ts – Timestep

Returns
None

class cellsium.model.**SizerCell**

Bases: *SimulatedCell*

Example cell implementing a simple sizer growth mechanism.

birth(*parent: Optional[SizerCell] = None, ts: Optional[Timestep] = None*) → None

Called when a cell is "born".

Parameters

- **parent** – Parent cell
- **ts** – Timestep

Returns
None

grow(*ts: Timestep*) → None

Called each timestep to grow cell.

Parameters
ts – Timestep

Returns
None

static random_sequences(*sequence: Any*) → Mapping[str, Any]

class cellsium.model.**Square**

Bases: *Rectangle*

Square cell geometry.

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.**TimerCell**

Bases: *SimulatedCell*

Example cell implementing a simple timer growth mechanism.

birth(*parent: Optional[TimerCell] = None, ts: Optional[Timestep] = None*) → None

Called when a cell is "born".

Parameters

- **parent** – Parent cell

- **ts** – Timestep

Returns
None

grow(*ts:* Timestep) → None

Called each timestep to grow cell.

Parameters
ts – Timestep

Returns
None

static random_sequences(*sequence: Any*) → Mapping[str, Any]

class cellsium.model.**Timestep**(*timestep: float*, *simulation:* Simulation, *simulator:* Simulator)

Bases: object

Timestep is an auxiliary class combining a certain timepoint, simulation and simulator.

property hours: float

The hours passed within this timestep.

Returns
Hours

simulation

simulator

property time: float

Total simulation time passed in seconds.

Returns
Seconds

property time_hours: float

Total simulation time passed in hours.

Returns
Hours

timestep

property world: *World*

class cellsium.model.**WithAngle**

Bases: object

Mixin adding a cell angle.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.**WithFluorescence**

Bases: object

Mixin adding a fluorescence value.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.**WithLineage**

Bases: object

Mixin providing lineage tracking.

copy() → *WithLineage*

static defaults() → Dict[str, Union[Callable[[], Any], float]]

next_cell_id() → None

class cellsium.model.**WithLineageHistory**

Bases: object

Mixin providing lineage history.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.**WithPosition**

Bases: object

Mixin adding a cell position.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.**WithProperDivisionBehavior**

Bases: object

Mixin adding division angle calculation.

get_division_positions(*count: int = 2*) → List[List[float]]

class cellsium.model.**WithRandomSequences**

Bases: object

Mixin for objects with random sequences.

all_random_sequences_generated_for = {}

classmethod get_random_sequences(*sequence: Optional[Any] = None*) → Any

property random: Any

class cellsium.model.**WithTemporalLineage**

Bases: object

Mixing providing temporal lineage history.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

cellsium.model.**assemble_cell**(*simulated_model: type*, **additional_classes*, *placed_cell: type = <class 'cellsium.model.PlacedCell'>*, *name: str = 'Cell'*)

Assembles a cell class from parent classes. Necessary as the cell class needs the right level of inheritance.

Parameters

- **simulated_model** – Model class
- **additional_classes** – Classes to create a cell type, or
- **placed_cell** – A prepared cell
- **name** – Optional name

Returns
Cell class

`cellsium.model.generate_cell(*additional_classes: type, name: str = 'PlacedCell')`

Generates a cell class using the standard classes, and possible additional classes.

Parameters

- **additional_classes** – Additional classes to inherit the cell from.
- **name** – Name of the class

Returns
Class

`cellsium.model.h_to_s(hours: Union[float, ndarray]) → Union[float, ndarray]`

Convert hours to seconds.

Parameters
hours – Hours

Returns
Seconds

`cellsium.model.s_to_h(seconds: Union[float, ndarray]) → Union[float, ndarray]`

Convert seconds to hours.

Parameters
seconds – Seconds

Returns
Hours

4.3.1 cellsium.model.initialization module

Cell parameter random initializations.

class `cellsium.model.initialization.RandomAngle`

Bases: `object`

Random initializations for cell angles.

static `random_sequences(sequence: RRF) → Dict[str, Any]`

class `cellsium.model.initialization.RandomBentRod`

Bases: `object`

Random initializations for cell bent radii.

static `random_sequences(sequence: RRF) → Dict[str, Any]`

class `cellsium.model.initialization.RandomFluorescence`

Bases: `object`

Random initializations for fluorescences.

static `random_sequences(sequence: RRF) → Dict[str, Any]`

class cellsium.model.initialization.**RandomPosition**

Bases: object

Random initializations for cell positions.

static random_sequences(*sequence:* RRF) → Dict[str, Any]

class cellsium.model.initialization.**RandomWidthLength**

Bases: object

Random initializations for cell width/lengths.

static random_sequences(*sequence:* RRF) → Dict[str, Any]

4.3.2 cellsium.model.agent module

Cell model classes and routines, general.

class cellsium.model.agent.**Copyable**

Bases: object

Mixin for copyable objects.

copy() → *Copyable*

class cellsium.model.agent.**IdCounter**

Bases: object

Id provider singleton class.

id_counter: int = 0

classmethod next_cell_id() → int

classmethod reset() → None

class cellsium.model.agent.**InitializeWithParameters**(***kwargs*)

Bases: object

Mixin for objects with defaults.

class cellsium.model.agent.**Representable**

Bases: object

Mixins for adding a repr implementation.

class cellsium.model.agent.**WithLineage**

Bases: object

Mixin providing lineage tracking.

copy() → *WithLineage*

static defaults() → Dict[str, Union[Callable[[], Any], float]]

next_cell_id() → None

class cellsium.model.agent.**WithLineageHistory**

Bases: object

Mixin providing lineage history.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.agent.**WithRandomSequences**

Bases: object

Mixin for objects with random sequences.

all_random_sequences_generated_for = {}

classmethod get_random_sequences(*sequence: Optional[Any] = None*) → Any

property random: Any

class cellsium.model.agent.**WithTemporalLineage**

Bases: object

Mixing providing temporal lineage history.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

4.3.3 cellsium.model.geometry module

Cell geometry model classes and routines.

class cellsium.model.geometry.**AutoMesh3D**

Bases: *Shape3D*

Mixin adding automatic solid-of-revolution generation.

points3d_on_canvas(*steps: int = 16*, *simplify: bool = False*) → Tuple[ndarray, ndarray]

raw_points3d(*steps: int = 16*, *simplify: bool = False*) → Tuple[ndarray, ndarray]

class cellsium.model.geometry.**BentRod**

Bases: *RodShaped*

Bent rod shaped cell geometry.

bend(*points: ndarray*) → ndarray

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

raw_points3d(*steps: int = 16*, *simplify: bool = False*) → Tuple[ndarray, ndarray]

class cellsium.model.geometry.**CellGeometry**

Bases: *WithAngle*, *WithPosition*, *AutoMesh3D*

Cell geometry base by combining multiple mixins.

points_on_canvas() → ndarray

class cellsium.model.geometry.**Coccoid**

Bases: *Shape*

Coccoid (spherical) cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.geometry.**Ellipsoid**

Bases: *Coccoid*

Ellipsoid cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.geometry.**Rectangle**

Bases: *Shape*

Rectangular cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.geometry.**RodShaped**

Bases: *Shape*

Rod shaped cell geometry.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

rod_raw_points(*simplify: bool = False*) → Tuple[ndarray, ndarray, ndarray, ndarray]

class cellsium.model.geometry.**Shape**

Bases: object

Base class for implementing cell shapes.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

get_approximation_circles() → Iterator[Tuple[float, Tuple[float, float]]]

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.geometry.**Shape3D**

Bases: *Shape*

Base class for implementing 3D cell shapes.

raw_points3d(*steps: int = 32, simplify: bool = False*) → ndarray

class cellsium.model.geometry.**Square**

Bases: *Rectangle*

Square cell geometry.

raw_points(*simplify: bool = False*) → ndarray

class cellsium.model.geometry.**WithAngle**

Bases: object

Mixin adding a cell angle.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.geometry.**WithFluorescence**

Bases: object

Mixin adding a fluorescence value.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.geometry.**WithPosition**

Bases: object

Mixin adding a cell position.

static defaults() → Dict[str, Union[Callable[[], Any], float]]

class cellsium.model.geometry.**WithProperDivisionBehavior**

Bases: object

Mixin adding division angle calculation.

get_division_positions(*count: int = 2*) → List[List[float]]

4.4 cellsium.output package

The output package contains the various output modules.

class cellsium.output.**Output**(**args*, ***kwargs*)

Bases: Selectable, Multiple

Base class of the Output classes.

display(*world:* World, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world:* World, ***kwargs*) → Optional[Any]

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

class cellsium.output.**OutputIndividualFiles**(**args*, ***kwargs*)

Bases: Tunable

Output individual files

default: bool = True

value = True

class cellsium.output.**OutputIndividualFilesWildcard**(**args*, ***kwargs*)

Bases: Tunable

Pattern for individual file names

default: str = '{}'

value = '{}'

class cellsium.output.**OutputIndividualFilesZeros**(**args*, ***kwargs*)

Bases: Tunable

Amount of digits used for outputting the frame number of individual file names

default: int = 3

value = 3

class cellsium.output.**OutputReproducibleFiles**(**args*, ***kwargs*)

Bases: Tunable

Output files in a reproducible manner

default: bool = True

value = True

cellsium.output.**check_overwrite**(*path: str*, *overwrite: bool = False*) → str

Check if a path exists, if so raising a RuntimeError if overwriting is disabled.

Parameters

- **path** – Path
- **overwrite** – Whether to overwrite

Returns

Path

`cellsium.output.`**`ensure_extension`**(*path: str*, *extension: str*) → str

Ensures that the path ends with extension, possibly adding it.

Parameters

- **path** – Path
- **extension** – Extension

Returns
Final path

`cellsium.output.`**`ensure_number`**(*path: str*, *number: int*, *disable_individual: bool = False*) → str

Depending on configuration, add a number to the path for consecutive output files.

Parameters

- **path** – Path
- **number** – Number
- **disable_individual** – Possibility to disable adding of a number

Returns
Path with number

`cellsium.output.`**`ensure_path`**(*path: str*) → str

Ensures that the parent directory to the to path exists.

Parameters
path – Path

Returns
the path

`cellsium.output.`**`ensure_path_and_extension`**(*path: str*, *extension: str*) → str

Ensures that the parent directory to path exists, and it has extension, possibly by adding it.

Parameters

- **path** – Path
- **extension** – Extension

Returns
Final path

`cellsium.output.`**`ensure_path_and_extension_and_number`**(*path: str*, *extension: str*, *number: int*, *disable_individual: bool = False*) → str

Ensures that a path exists, has an extension and a number.

Parameters

- **path** – Path
- **extension** – Extension
- **number** – Number
- **disable_individual** – Whether to disable adding of number

Returns
Final path

4.4.1 cellsium.output.xml module

Output as Trackmate XML lineage files, compatible with the JuNGLE extensions.

class cellsium.output.xml.**TrackMateXML**(**args*, ***kwargs*)

Bases: *Output*

display(*world: World*, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world: World*, *time: float = 0.0*, ***kwargs*) → None

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, *time: float = 0.0*, *overwrite: bool = False*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

4.4.2 cellsium.output.plot module

Output using matplotlib.

class cellsium.output.plot.**MicrometerPerCm**(**args*, ***kwargs*)

Bases: Tunable

default: float = 2.5

value = 2.5

class cellsium.output.plot.**PlotRenderer**(**args*, ***kwargs*)

Bases: *Output*, Default

Output using matplotlib.

display(*world: World*, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world: World, **kwargs*) → Tuple[Figure, Axes]

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World, file_name: str, output_count: int = 0, overwrite: bool = False, **kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

4.4.3 cellsium.output.mesh module

Mesh output in the STL format.

class cellsium.output.mesh.**MeshOutput**(**args, **kwargs*)

Bases: *Output*

Mesh output in the STL format.

display(*world: World, **kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world: World, **kwargs*) → List[Dict[str, ndarray]]

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

4.4.4 cellsium.output.svg module

Output as SVG vector images.

class cellsium.output.svg.**SvgRenderer**(**args*, ***kwargs*)

Bases: *Output*

static create_xml()

display(*world: World*, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world: World*, ***kwargs*) → None

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

static points_to_path(*points: ndarray*) → str

write(*world: World*, *file_name: str*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

4.4.5 cellsium.output.render module

Photorealistic rendered output.

class cellsium.output.render.**FluorescenceRenderer**(**args*, ***kwargs*)

Bases: *PlainRenderer*

channel: int = 0

output(*world:* World, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

class cellsium.output.render.**NoisyUnevenIlluminationPhaseContrast**(**args*, ***kwargs*)

Bases: *UnevenIlluminationPhaseContrast*

output(*world:* World, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

class cellsium.output.render.**PhaseContrastRenderer**(**args*, ***kwargs*)

Bases: *PlainRenderer*

output(*world:* World, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

class cellsium.output.render.**PlainRenderer**(**args*, ***kwargs*)

Bases: *Output*

static convert(*image: ndarray*, *max_value: int = 255*) → ndarray

debug_output(*name: str*, *array: ndarray*) → None

display(*world:* World, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

static imwrite(*name: str*, *img: ndarray*, *overwrite: bool = False*, *output_count: Optional[int] = None*) → bool

static new_canvas()

output(*world: World*, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

static render_cells(*canvas: ndarray*, *array_of_points: ndarray*, *fast: bool = False*) → ndarray

write(*world: World*, *file_name: str*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

write_debug_output = False

class cellsium.output.render.**RenderChannels**(**args*, ***kwargs*)

Bases: Tunable

Channels to render (i.e. output classes which produce rendered images)

default = 'NoisyUnevenIlluminationPhaseContrast'

static get_mapping() → Dict[str, type]

classmethod instantiate() → *PlainRenderer*

classmethod test(*value: str*) → bool

value = 'NoisyUnevenIlluminationPhaseContrast'

class cellsium.output.render.**TiffOutput**(**args*, ***kwargs*)

Bases: *Output*

output(*world: World*, ***kwargs*) → List[ndarray]

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output_type

alias of uint8

write(*world: World*, *file_name: str*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

class cellsium.output.render.**UnevenIlluminationPhaseContrast**(**args*, ***kwargs*)

Bases: *PhaseContrastRenderer*

create_uneven_illumination() → ndarray

new_uneven_illumination() → ndarray

output(*world: World*, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

cellsium.output.render.**get_canvas_points_for_cell**(*cell: CellGeometry*, *image_height: Optional[Tuple[int, int]] = None*) → ndarray

cellsium.output.render.**new_canvas**(*dtype=<class 'numpy.float32'>*) → ndarray

4.4.6 cellsium.output.serialization module

Serialization outputs.

class cellsium.output.serialization.**CsvOutput**(**args*, ***kwargs*)

Bases: *Output*

CSV Tabular Output.

output(*world: World*, *time: Optional[float] = None*, ***kwargs*) → List[Dict[str, Any]]

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, *time: Optional[float] = None*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*)

Output and write the World to file_name.

Parameters

- **world** – World

- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

class cellsium.output.serialization.**JsonPickleSerializer**(**args*, ***kwargs*)

Bases: *Output*

Output as jsonpickle serialized files.

display(*world: World*, ***kwargs*) → None

Output and display the World, e.g. via a GUI window.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

output(*world: World*, ***kwargs*) → str

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*) → None

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

class cellsium.output.serialization.**QuickAndDirtyTableDumper**(**args*, ***kwargs*)

Bases: *Output*

Simple tabular output.

output(*world: World*, ***kwargs*) → ndarray

Outputs the World, this function is usually called by either write or display.

Parameters

- **world** – World
- **kwargs** – Additional arguments

Returns

write(*world: World*, *file_name: str*, *time: Optional[float] = None*, *overwrite: bool = False*, *output_count: int = 0*, ***kwargs*)

Output and write the World to file_name.

Parameters

- **world** – World
- **file_name** – Filename to write output to
- **kwargs** – Additional arguments

Returns

4.4.7 cellsium.output.gt module

Ground truth outputs in COCO, YOLO and a generic mask format.

class cellsium.output.gt.**COCOOutput**(**args*, ***kwargs*)

Bases: GroundTruthOutput

Output in the COCO format.

static now() → str

class cellsium.output.gt.**GenericMaskOutput**(**args*, ***kwargs*)

Bases: GroundTruthOutput

Generic mask output (i.e. directories of files).

static generate_cells_mask(*cells: Iterable[*CellGeometry*], cell_value: int = 1, binary: bool = True*) → ndarray

static imwrite(**args*, ***kwargs*) → None

class cellsium.output.gt.**YOLOOutput**(**args*, ***kwargs*)

Bases: GroundTruthOutput

Output in the YOLO format.

4.5 cellsium.simulation package

Simulation package contains the simulation/simulator-related classes.

class cellsium.simulation.**BaseSimulator**

Bases: object

add(*cell: object*) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

clear() → None

Clear the (world of the) simulation.

Returns
None

remove(*cell*) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float*) → None

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

4.5.1 cellsium.simulation.placement package

Placement simulation package, contains the placement simulators.

class cellsium.simulation.placement.**Box2D**(**args*, ***kwargs*)

Bases: *PhysicalPlacement*, *PlacementSimulation*

add(*cell:* PlacedCell) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

remove(*cell:* PlacedCell) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float*) → None

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

verbose: bool = False

class cellsium.simulation.placement.**Chipmunk**(**args*, ***kwargs*)

Bases: *PhysicalPlacement*, *PlacementSimulation*, Default

add(*cell:* PlacedCell) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

clear() → None

Clear the (world of the) simulation.

Returns
None

convergence_check_interval: int = 15

inner_step(*time_step: float = 0.1*, *iterations: int = 9999*, *converge: bool = True*, *epsilon: float = 0.1*) → float

look_back_threshold: int = 5

remove(*cell:* PlacedCell) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float*) → None

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

verbose: bool = False

class cellsium.simulation.placement.**PlacementSimulation**(**args*, ***kwargs*)

Bases: *BaseSimulator*, Selectable

cellsium.simulation.placement.pybox2d module

Placement simulation using Box2D physics engine.

class cellsium.simulation.placement.pybox2d.**Box2D**(**args*, ***kwargs*)

Bases: *PhysicalPlacement*, *PlacementSimulation*

add(*cell:* PlacedCell) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

remove(*cell:* PlacedCell) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float*) → None

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

verbose: bool = False

cellsium.simulation.placement.base module

class cellsium.simulation.placement.base.**NoPlacement**(**args*, ***kwargs*)

Bases: *PlacementSimulation*

class cellsium.simulation.placement.base.**PhysicalPlacement**(**args*, ***kwargs*)

Bases: *PlacementSimulation*, Virtual

clear() → None

Clear the (world of the) simulation.

Returns
None

class cellsium.simulation.placement.base.**PlacementSimulation**(**args*, ***kwargs*)

Bases: *BaseSimulator*, Selectable

class cellsium.simulation.placement.base.**PlacementSimulationSimplification**(**args*, ***kwargs*)

Bases: Tunable

How much the placement should be simplified, 0: use the normal shapes, 1: use simplified shapes, 2: use many-circle approximations

default: int = 0

value = 0

cellsium.simulation.placement.base.**ensure_python**(*value: Any*) → Any

cellsium.simulation.placement.pymunk module

Placement simulation using Pymunk physics engine.

class cellsium.simulation.placement.pymunk.**Chipmunk**(**args*, ***kwargs*)

Bases: *PhysicalPlacement*, *PlacementSimulation*, Default

add(*cell:* PlacedCell) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

clear() → None

Clear the (world of the) simulation.

Returns
None

convergence_check_interval: int = 15

inner_step(*time_step: float = 0.1*, *iterations: int = 9999*, *converge: bool = True*, *epsilon: float = 0.1*) → float

look_back_threshold: int = 5

remove(*cell:* PlacedCell) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float*) → None

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

verbose: bool = False

class cellsium.simulation.placement.pymunk.**ChipmunkPlacementRadius**(**args*, ***kwargs*)

Bases: Tunable

Chipmunk placement radius, additional radius objects will have around them

default: float = 0.05

value = 0.05

4.5.2 cellsium.simulation.simulator module

Simulator base classes.

class cellsium.simulation.simulator.**Simulation**

Bases: object

Simulation represents the simulation state at a certain timepoint, i.e. a World and a time.

class cellsium.simulation.simulator.**Simulator**

Bases: *BaseSimulator*

Simulator class, a class serving as interface to World and sub-simulators (such as physical placement), as well as the caller of each cells step function.

add(*cell: object*) → None

Add a cell to the simulation.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary

Returns
None

clear() → None

Clear the (world of the) simulation.

Returns
None

remove(*cell: object*) → None

Remove a cell from the simulation.

Parameters
cell – Cell

Returns
None

step(*timestep: float = 0.0*) → *Timestep*

Advance the simulation by a timestep.

Parameters
timestep – Time passed in seconds

Returns
None

class cellsium.simulation.simulator.**Timestep**(*timestep: float*, *simulation:* Simulation, *simulator:* Simulator)

Bases: object

Timestep is an auxiliary class combining a certain timepoint, simulation and simulator.

property hours: float

The hours passed within this timestep.

Returns
Hours

simulation

simulator

property time: float

Total simulation time passed in seconds.

Returns
Seconds

property time_hours: float

Total simulation time passed in hours.

Returns
Hours

timestep

property world: *World*

class cellsium.simulation.simulator.**World**

Bases: object

The World class contains the cells and, if present, the boundaries.

add(*cell: object*) → None

Adds a cell to the World.

Parameters
cell – Cell

Returns
None

add_boundary(*coordinates: ndarray*) → None

Add a boundary to the simulation.

Parameters
coordinates – Coordinates of the boundary.

Returns
None

clear() → None

Resets the World.

Returns
None

commit() → None

Commits a step. Cells to be added, and cells to be removed, will only be applied once commit is called.

Returns
None

copy() → *World*

Creates a copy of thw World.

Returns
Copy of the World

remove(*cell: object*) → None

Removes a cell from the world.

Parameters
cell – Cell

Returns
None

4.6 cellsium.random module

Random number generation infrastructure.

class `cellsium.random.RRF`(*mode: str = 'callable'*)

Bases: `object`

Reproducible random function.

classmethod `chain`(*func:* KwargFunction, ***kwargs*) → Iterator

Calls func with kwargs and yields from it.

Parameters

- **func** – Function to call
- **kwargs** – Kwargs to pass

Returns
Iterator of values

classmethod `compose`(*func:* KwargFunction, ***kwargs*) → Iterator

Calls a function func with an element of the sequences from the kwargs as kwargs.

Parameters

- **func** – Function to be called
- **kwargs** – Kwargs of sequences, of which an element each will be used for each function call

Returns
Iterator of values

generator = <cellsium.random.RRF object>

classmethod `seed`(*seed: Optional[int] = None*) → int

Set the seed for the RRF.

Parameters
seed – Seed

Returns
Seed

seed_sequence = SeedSequence(entropy=1, n_children_spawned=26,)

seed_value: int = 1

sequence = <cellsium.random.RRF object>

classmethod `spawn_generator`() → Generator

Generates a new np.random.Generator from the seed and the configured bitgenerator.

Returns
The Generator instance

classmethod `wrap`(*sequence: Iterable*, *func:* AnyFunction) → Iterator

Wraps the sequence with the function func so that each returned element x becomes func(x).

Parameters

- **sequence** – Input sequence

- **func** – Function to be called

Returns
Iterator of values

class cellsium.random.**RandomNumberGenerator**(**args*, ***kwargs*)

Bases: Tunable

Random number generator to be used

classmethod available_rngs() → Dict[str, Type[BitGenerator]]

default: str = 'PCG64'

classmethod get() → Type[BitGenerator]

classmethod test(*value: str*) → bool

type_
alias of str

value = 'PCG64'

class cellsium.random.**Seed**(**args*, ***kwargs*)

Bases: Tunable

Seed for the random number generator

default: int = 1

type_
alias of int

value = 1

cellsium.random.**enforce_bounds**(*iterator: Iterator*, *minimum: float = -inf*, *maximum: float = inf*) → Iterator[Union[float, ndarray]]

Will iter thru an iterator til a value is within bounds. For arrays, all values will be considered.

Parameters

- **iterator** – Iterator
- **minimum** – Minimum value
- **maximum** – Maximum value

Returns
An iterator of values within bounds

4.7 cellsium.parameters module

Main set of parameters for the simulation.

class cellsium.parameters.**Calibration**(**args*, ***kwargs*)

Bases: Tunable

Calibration for outputs, micrometer per pixel

default: float = 0.065

value = 0.065

class cellsium.parameters.**Height**(**args*, ***kwargs*)

Bases: Tunable

Height of the (outputted) simulation

default: float = 60.0

value = 60.0

class cellsium.parameters.**NewCellBendLowerLower**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor minimum for the lower part of new new cells

default: float = -0.1

value = -0.1

class cellsium.parameters.**NewCellBendLowerUpper**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor maximum for the lower part of new new cells

default: float = 0.1

value = 0.1

class cellsium.parameters.**NewCellBendOverallLower**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor minimum for new new cells

default: float = -0.1

value = -0.1

class cellsium.parameters.**NewCellBendOverallUpper**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor maximum for new new cells

default: float = 0.1

value = 0.1

class cellsium.parameters.**NewCellBendUpperLower**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor minimum for the upper part of new new cells

default: float = -0.1

value = -0.1

class cellsium.parameters.**NewCellBendUpperUpper**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Bend factor maximum for the upper part of new new cells

default: float = 0.1

value = 0.1

class cellsium.parameters.**NewCellCount**(**args*, ***kwargs*)

Bases: Tunable

New cells to add to the simulation

default: int = 1

value = 1

class cellsium.parameters.**NewCellLength1Mean**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Mean cell length, subtype one

default: float = 2.5

value = 2.5

class cellsium.parameters.**NewCellLength1Std**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Standard deviation of the cell length, subtype one

default: float = 0.15

value = 0.15

class cellsium.parameters.**NewCellLength2Mean**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Mean cell length, subtype two

default: float = 1.25

value = 1.25

class cellsium.parameters.**NewCellLength2Std**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Standard deviation of the cell length, subtype one

default: float = 0.15

value = 0.15

class cellsium.parameters.**NewCellLengthAbsoluteMax**(**args*, ***kwargs*)

Bases: Tunable

Absolute maximum length of new cells

default: float = 3.5

value = 3.5

class cellsium.parameters.**NewCellLengthAbsoluteMin**(**args*, ***kwargs*)

Bases: Tunable

Absolute minimum length of new cells

default: float = 0.8

value = 0.8

class cellsium.parameters.**NewCellRadiusFromCenter**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Maximum radius for new cells to be spawned from the origin

default: float = 5.0

value = 5.0

class cellsium.parameters.**NewCellWidthAbsoluteMax**(**args*, ***kwargs*)

Bases: Tunable

Absolute maximum width of new cells

default: float = 1.5

value = 1.5

class cellsium.parameters.**NewCellWidthAbsoluteMin**(**args*, ***kwargs*)

Bases: Tunable

Absolute minimum width of new cells

default: float = 0.75

value = 0.75

class cellsium.parameters.**NewCellWidthMean**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Mean cell width for new cells

default: float = 1.0

value = 1.0

class cellsium.parameters.**NewCellWidthStd**(**args*, ***kwargs*)

Bases: *RandomlyDistributed*

Standard deviation of the width of new cells

default: float = 0.1

value = 0.1

class cellsium.parameters.**RandomlyDistributed**(**args*, ***kwargs*)

Bases: Tunable

Parent class for randomly distributed tunables

class cellsium.parameters.**Width**(**args*, ***kwargs*)

Bases: Tunable

Width of the (outputted) simulation

default: float = 40.0

value = 40.0

`cellsium.parameters.h_to_s(hours: Union[float, ndarray])` → Union[float, ndarray]

Convert hours to seconds.

Parameters
hours – Hours

Returns
Seconds

`cellsium.parameters.pixel_to_um(pix: Union[float, ndarray])` → Union[float, ndarray]

Convert pixel to micrometer.

Parameters
pix – Pixel value

Returns
Micrometer value

`cellsium.parameters.s_to_h(seconds: Union[float, ndarray])` → Union[float, ndarray]

Convert seconds to hours.

Parameters
seconds – Seconds

Returns
Hours

`cellsium.parameters.um_to_pixel(um: Union[float, ndarray])` → Union[float, ndarray]

Convert micrometer to pixel.

Parameters
um – Micrometer value

Returns
Pixel value

4.8 cellsium.typing module

Type hints definitions.

class `cellsium.typing.AnyFunction`

Bases: `Protocol`

class `cellsium.typing.KwargFunction`

Bases: `Protocol`

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y