Python for CC3D - Quick Reference Guide

Release 4.2.4

Julio Belmonte, Maciej Swat, T.J. Sego, James A. Glazier

Feb 21, 2021

1 List of cell properties

All cell properties are proportional to a unit grid according to lattice dimensionality (*e.g.*, for unit length x in a 3-dimensional regular lattice, `cell.volume` and `cell.surface` have dimensions x^3 and x^2, respectively):

Name	Attribute	Modifiable?	Comments
Cell identity	`id`	No	Unique cell identification number
Cell type	`type`	Yes	Integer indicating the cell type
Cell volume	`volume`	No	Instantaneous cell volume
[1]	`targetVolume`	Yes	Target value of volume constraint
[1]	`lambdaVolume`	Yes	Lambda of the volume constraint
Cell surface[2]	`surface`	No	Instantaneous cell surface
	`targetSurface`	Yes	Target value of surface constraint
	`lambdaSurface`	Yes	Lambda of the surface constraint
Center of mass[3]	`xCOM`	No	Cartesian coordinate x of center of mass
	`yCOM`	No	Cartesian coordinate y of center of mass
	`zCOM`	No	Cartesian coordinate z of center of mass
Eccentricity[4]	`ecc`	No	Eccentricity of cell
Inertia tensor[4]	`iXX`	No	Moment of inertia x-axis about the x-axis
	`iYY`	No	Moment of inertia y-axis about the y-axis
	`iZZ`	No	Moment of inertia z-axis about the z-axis
	`iXY`	No	Moment of inertia x-axis about the y-axis
	`iXZ`	No	Moment of inertia x-axis about the z-axis
	`iYZ`	No	Moment of inertia y-axis about the z-axis
Minor axis vector[4]	`lX`	No	Component x of vector along semiminor axis
	`lY`	No	Component y of vector along semiminor axis
	`lZ`	No	Component z of vector along semiminor axis
Directional forces[5]	`lambdaVecX`	Yes	Force component acting on the x-direction
	`lambdaVecY`	Yes	Force component acting on the y-direction
	`lambdaVecZ`	Yes	Force component acting on the z-direction
Cell internal pressure[1]	`pressure`	No	Instantaneous internal pressure
Cell surface tension[2]	`surfaceTension`	No	Instantaneous surface tension
Cluster surface tension[2]	`clusterSurfaceTension`	No	Surface tension of cluster

2 How to loop over all cells

```
for cell in self.cell_list:
    # commands for all cells go here

for cell in self.cell_list_by_type(self.TYPENAME1, self.TYPENAME2):
    # commands for all cells of cell type "Typename1" or "Typename2" go here
```

Note: Integers specified for all cell types `Type` defined in the *.xml* file are assigned to all steppables as attributes `TYPE` (*e.g.*, for a type with name `Condensing` and integer = 2, `self.CONDENSING = 2` for all steppables).

[1] Only available when `Volume` plugin is loaded and parameters are assigned per cell in a steppable (rather than in CC3DML)
[2] Only available when `Surface` , `SurfaceTracker`, `SurfaceFlex` or `SurfaceLocalFlex` plugins are loaded.
[3] Only available when `CenterOfMass` plugin is loaded.
[4] Only available when `MomentOfInertia` plugin is loaded.
[5] Only available when `ExternalPotential` plugin is loaded.

3 How to loop over a cell's neighbors

```
for cell in self.cell_list:
    for neighbor, common_surface_area in self.get_cell_neighbor_data_list(cell):
        # commands for all neighbors go here
        if neighbor:
            # commands for all neighbors, excluding medium, go here
```

Note: Make sure to load `NeighborTracker` plugin from either the *.xml* or the *.py* file. Neighbor cells have the same properties as those listed before for cells. To access them, substitute `cell` with `neighbor`.

4 How to loop over all lattice sites

```
for x, y, z in self.every_pixel():
    # commands for each lattice site go here
```

Note: Make sure to load the `PixelTracker` plugin from either the *.xml* or the *.py* file.

5 How to do a loop over cell and medium sites

```
# Loop over all pixels of cell with id = 1
cell_1 = self.fetch_cell_by_id(1)
for ptd in self.get_cell_pixel_list(cell_1)
    this_pixel = ptd.pixel

# Loop over all current medium pixels
for ptd in self.pixel_tracker_plugin.getMediumPixelSet():
    this_pixel = ptd.pixel
```

Note: Make sure to load the `PixelTracker` plugin from either the *.xml* or the *.py* file. For tracking medium sites, make sure to first enable the medium tracking option for `PixelTracker`.

6 How to loop over a cell's boundary sites

```
pixel_list = self.get_cell_boundary_pixel_list(cell)
for boundary_pixel_tracker_data in pixel_list:
    pt = boundary_pixel_tracker_data.pixel
    # commands for each cell boundary pixel (pt) go here
```

Note: Make sure to load the `BoundaryPixelTracker` plugin from either the *.xml* or the *.py* file.

7 How to attach/access/modify a dictionary to a cell

Each cell, by default, has a Python dictionary `dict` attached to it as a cell attribute.

```
for cell in self.cell_list:
    # get custom cell attributes 'custom_cell_val1' and 'custom_cell_val2'
    # with keys 'custom_cell_keyA' and 'custom_cell_keyB'
    custom_cell_val1 = cell.dict['custom_cell_keyA']
    custom_cell_val2 = cell.dict['custom_cell_keyB']
    # do calculations for custom cell attributes here
    # <calculations -> custom_cell_val1, custom_cell_val2>
    # store custom cell attributes
    cell.dict['custom_cell_keyA'] = custom_cell_val1
    cell.dict['custom_cell_keyB'] = custom_cell_val2
```

8 How to simulate mitosis

A special steppable class `MitosisSteppableBase` implements convenient mitosis-related methods. Mitosis events are triggered in `step` and handled in `update_attributes`:

```
class MitosisSteppable(MitosisSteppableBase):
    def __init__(self, frequency=1):
        MitosisSteppableBase.__init__(self, frequency)
        # Select relative position of parent and child after mitosis
        #   0  - parent and child positions will be randomized between mitosis event
        #   -1 - parent appears on the 'left' of the child
        #   1  - parent appears on the 'right' of the child
        self.set_parent_child_position_flag(-1)

    def step(self, mcs):
        # Make a Python list of cells to divide
        cells_to_divide = []
        for cell in self.cell_list:
            if <mitosis_condition_here>:
                cells_to_divide.append(cell)
        # Implement oriented mitosis by applying an available cell division method
        for cell in cells_to_divide:
            self.divide_cell_random_orientation(cell)
            # self.divide_cell_orientation_vector_based(cell, 1, 1, 0)
            # self.divide_cell_along_major_axis(cell)
            # self.divide_cell_along_minor_axis(cell)

    def update_attributes(self):
        # Updates to parent cell attributes before cloning them go here
        self.parent_cell.targetVolume /= 2.0 # Example: reduce parent target volume
        # Clone all parent attributes to child
        self.clone_parent_2_child()
        # Changes to child cell attributes after clone go here
        self.child_cell.type = self.parent_cell.ANOTHERTYPE # Example: change type
```

Note: `update_attributes` is called for every call to a cell division method (*e.g.*, `divide_cell_random_orientation`), where `self.parent_cell` is the cell object passed to the cell division method, and `self.child_cell` is the cell object added to simulation by mitosis.

9 How to share data between steppables

All steppables, by default, can share data with each other by accessing a global Python dictionary `shared_steppable_vars` as a steppable attribute:

```
class InitiatorSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        self.shared_steppable_vars['x_shared'] = 0

class PrinterSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def step(self, mcs):
        print('x_shared=', self.shared_steppable_vars['x_shared'])
```

10 How to access/modify the cell lattice

```
pt = CompuCell.Point3D() # define a lattice vector
pt.x = 3; pt.y = 2; pt.z = 0 # specify its coordinates
cell = self.cell_field[pt.x, pt.y, pt.z] # access the cell or Medium at (3, 2, 0)
# create an extension of that cell in another location (3, 1, 0)
pt.x = 3; pt.y = 1; pt.z = 0
self.cell_field[pt.x, pt.y, pt.z] = cell
# place a brand new cell over a subdomain with type "Condensing" defined in .xml file
self.cell_field[10:14, 10:14, 0] = self.new_cell(self.CONDENSING)
```

11 How to access/modify PDE field values

```
# Get PDE field defined in .xml with name "MyFieldName"
my_field = CompuCell.getConcentrationField(self.simulator, "MyFieldName")
# Blend at (0, 0, 0) with a neighbor
my_field[0, 0, 0] = (my_field[0, 0, 0] + my_field[1, 0, 0]) / 2.0
minVal = my_field.min() # Get current field minimum
maxVal = my_field.max() # Get current field maximum
```

12 How to create extra fields

```
class ExtraFieldsSteppable(SteppableBasePy):
    def __init__(self, frequency=10):
        SteppableBasePy.__init__(self, frequency)
        # Create extra fields
        self.create_scalar_field_py("sFieldP") # pixel-based scalar field
        self.create_vector_field_py("vFieldP") # pixel-based vector field
        self.create_scalar_field_cell_level_py("sFieldC") # cell-based scalar field
```

(continues on next page)

(continued from previous page)

```
        self.create_vector_field_cell_level_py("vFieldC") # cell-based vector field
        # Create extra fields that use automatic tracking of cell attributes
        self.track_cell_level_scalar_attribute(field_name='ID2_FIELD',
                                               attribute_name='id2')
        self.track_cell_level_vector_attribute(field_name='COM_VECTOR_FIELD',
                                               attribute_name='com_vector')

    def start(self):
        # Initialize attributes in cell dictionary for automatic tracking fields
        for cell in self.cell_list:
            cell.dict['id2'] = cell.id ** 2
            cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]

    def step(self, mcs):
        # access extra fields by names passed to instantiation methods in __init__
        scalar_field_pixel = self.field.sFieldP
        vector_field_pixel = self.field.vFieldP
        scalar_field_cell = self.field.sFieldC
        vector_field_cell = self.field.vFieldC
        # modify some pixel-based values; sites are accessed just like the cell field
        scalar_field_pixel[0, 1, 2] = 3.0
        vector_field_pixel[1, 2, 3, 0] = 1.0 # vector components are in dim. 4
        # modify some cell-based values
        cell = self.cell_field[0, 1, 2]
        scalar_field_cell[cell] = cell.id * 2
        vector_field_cell[cell] = [0.0, 1.0, 2.0]
        # Update attributes in cell dictionary for automatic tracking fields
        for cell in self.cell_list:
            cell.dict['id2'] = cell.id ** 2
            cell.dict['com_vector'] = [cell.xCOM, cell.yCOM, 0.0]
```

Note: Extra fields do not necessarily have to be created inside `__init__`, though full functionality associated with fields requires it.

Note: Extra fields do not directly participate in any core calculations. Rather, they can be used to store data associated with core calculations at each lattice site that can, like other data, be passed to the CC3D computational core or visualized in Player.

13 How to write output files to the simulation output directory

```
def step(self, mcs):
    output_dir = self.output_dir
    if output_dir is not None:
        # Write output file with unique name by appending MCS to template
        output_path = Path(output_dir).joinpath('step_' + str(mcs).zfill(3) + '.dat')
        with open(output_path, 'w') as f_out:
            f_out.write('{} {} {}\n'.format(1, 2, 3))
```

Note: `self.output_dir` is a special variable in each steppable that stores the directory where the output of the current simulation will be written. Other files can be specified by substituting `output_dir` and `output_path` with a different directory and file name, respectively.

14 How to add custom plots

```
class VisualizationSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        # Create plot window for Cell 1 volume and surface
        self.plot_win1 = self.add_new_plot_window(
            title='Volume and surface area of Cell 1',
            x_axis_title='Monte Carlo Step (MCS)',
            y_axis_title='Variables',
            x_scale_type='linear',
            y_scale_type='log',
            grid=True)
        self.plot_win1.add_plot("Volu1", style='Dots', color='red', size=5)
        self.plot_win1.add_plot('Surf1', style='Steps', color='black', size=5)
        # Create plot window for histogram of cell volumes ans surfaces
        self.plot_win2 = self.add_new_plot_window(
            title='Cell volume/surface histogram',
            x_axis_title='Number of cells',
            y_axis_title='Volume/surface (pixels^n)')
        # alpha is transparency: = 0 -> transparent, = 255 -> opaque
        self.plot_win2.add_histogram_plot(plot_name='voluH', color='green', alpha=100)
        self.plot_win2.add_histogram_plot(plot_name='surfH', color='red', alpha=100)

    def step(self, mcs):
        # Collect cell data
        cell1_volu = 0
        cell1_surf = 0
        volu_list = []
        surf_list = []
        for cell in self.cell_list:
            volu_list.append(cell.volume)
            surf_list.append(cell.surface)
            if cell.id == 1:
                cell1_volu = cell.volume
                cell1_surf = cell.surface
        # Update plots
        self.plot_win1.add_data_point('Volu1', mcs, cell1_volu)
        self.plot_win1.add_data_point('Surf1', mcs, cell1_surf)
        self.plot_win2.add_histogram(plot_name='VoluH', value_array=volu_list,
                                     number_of_bins=10)
        self.plot_win2.add_histogram(plot_name='SurfH', value_array=surf_list,
                                     number_of_bins=10)
        if self.output_dir is not None: # Export histogram plots
            # Export data in CSV format (needs "from pathlib import Path")
            txt_path = Path(self.output_dir).joinpath("Hist_" + str(mcs) + ".txt")
            self.plot_win.save_plot_as_data(txt_path, CSV_FORMAT)
            # export image with size 1000 x 1000 (default is 400 x 400)
            png_path = Path(self.output_dir).joinpath("Hist_" + str(mcs) + ".png")
            self.plot_win.save_plot_as_png(png_path, 1000, 1000)
```

15 How to load and run a subcellular SBML model

```
class SBMLSolverSteppable(SteppableBasePy):
    def __init__(self, frequency=1):
        SteppableBasePy.__init__(self, frequency)

    def start(self):
        # Add options that setup SBML solver integrator
        # These are optional but useful when encountering integration instabilities
        options = {'relative': 1e-10, 'absolute': 1e-12}
        self.set_sbml_global_options(options)
        # Specify location of SBML model file for a model of a species "S"
        model_file = 'Simulation/test_1.xml'
        # Specify initial conditions
        initial_conditions = {}
        initial_conditions['S'] = 0.00020
        # Add SBML model with name "dp" to some cells by id
        self.add_sbml_to_cell_ids(model_file=model_file, model_name='dp',
                                  cell_ids=list(range(1, 11)), step_size=0.5,
                                  initial_conditions=initial_conditions)
        # Add free-floating SBML model with name "Medium_dp" to the medium
        self.add_free_floating_sbml(model_file=model_file, model_name='Medium_dp',
                                    step_size=0.5,
                                    initial_conditions=initial_conditions)
        # Add SBML model to cell with id 20
        cell_20 = self.fetch_cell_by_id(20)
        self.add_sbml_to_cell(model_file=model_file, model_name='dp', cell=cell_20)

    def step(self, mcs):
        self.timestep_sbml() # Perform integration for this step in SBML solver
        # Get SBML model current values by model name for cell with id 20
        cell_20 = self.fetch_cell_by_id(20)
        vals_20 = cell_20.sbml.dp.values()
        # Get free-floating SBML model current values by model name for the medium
        vals_ff = self.sbml.Medium_dp.values()
        # Set value for species S1 in free-floating SBML model
        Medium_dp = self.sbml.Medium_dp
        Medium_dp['S'] = 10
        # Delete SBML model from some cells by id
        self.delete_sbml_from_cell_ids(model_name='dp', cell_ids=list(range(1, 11)))
        # Copy SBML solver from cell 20 to cell 25
        cell_25 = self.fetch_cell_by_id(25)
        self.copy_sbml_simulators(from_cell=cell_20, to_cell=cell_25)
```

16 How to write/load/run a subcellular Antimony model all in Python

```
class AntimonySolverSteppable(SteppableBasePy):

    def __init__(self,frequency=1):
        SteppableBasePy.__init__(self,frequency)

    def start(self):
        # Define Antimony model with a Python multi-line string
        antimony_model_string = """model rkModel()
```

(continues on next page)

(continued from previous page)

```
        <Antimony model specification goes here>
        end"""

        options = {'relative': 1e-10, 'absolute': 1e-12}
        self.set_sbml_global_options(options)
        step_size = 1e-2

        for cell in self.cell_list:
            self.add_antimony_to_cell(model_string=antimony_model_string,
                                      model_name='dp',
                                      cell=cell,
                                      step_size=step_size)

    def step(self):
        self.timestep_sbml()
```

Note: All SBML model instantiations methods have corresponding Antimony methods. Antimony models are translated into SBML models, which are then passed to the corresponding SBML instantiation methods. As such, after instantiation they can be accessed and manipulated in the same ways as models specified using SBML model specification.

Note: Antimony models can also be specified in separate text files and loaded by instead passing the location of the file to an instantiation methods using the keyword argument `model_file` (as in SBML methods).

17 How to load a CC3D steppable class

In the main Python file `<mainFile.py>`, register custom steppables by following the procedure shown between the first and last lines:

```
from cc3d import CompuCellSetup # First line: import CompuCellSetup

from <SteppablesFile> import <NameOfClass> # Import steppable from steppables file
CompuCellSetup.register_steppable(steppable=<NameOfClass>(frequency=1)) # Register
# <Additional steppables registered here...>

CompuCellSetup.run() # Last line: run CC3D
```

Note: `<NameOfClass>` refers to the name of the steppable class defined in the Python script `<SteppablesFile>` (*e.g.*, `class MySteppable(SteppableBasePy)` in `MySteppables.py`).