SimPhoNy-Mayavi Documentation

Release 0.1.0

SimPhoNy FP7 Collaboration

August 25, 2015

Contents

A plugin-library for the Simphony framework (http://www.simphony-project.eu/) to provide visualization support (using http://www.paraview.org/) of the CUDS highlevel components.

CHAPTER 1

Repository

Simphony-paraview is hosted on github: https://github.com/simphony/simphony-paraview

CHAPTER 2

Requirements

- paraview >= 3.14.1 (official Ubuntu 12.04 package)
- simphony >= 0.2.0

2.1 Optional requirements

To support the documentation built you need the following packages:

- sphinx >= 1.2.3
- sectiondoc https://github.com/enthought/sectiondoc
- mock

Alternative running **`pip install -r doc_requirements.txt`** should install the minimum necessary components for the documentation built.

CHAPTER 3

Installation

The package requires python 2.7.x, installation is based on setuptools:

```
# build and install
python setup.py install
```

or:

```
# build for in-place development
python setup.py develop
```

CHAPTER 4

Testing

To run the full test-suite run:

```
python -m unittest discover
```

CHAPTER 5

Documentation

To build the documentation in the doc/build directory run:

```
python setup.py build_sphinx
```

Note:

- One can use the –help option with a setup.py command to see all available options.
- The documentation will be saved in the `./build` directory.

CHAPTER 6

Usage

After installation the user should be able to import the `paraview` visualization plugin module by:

```
from simphony.visualization import paraview_tools
paraview_tools.show(cuds)
```

CHAPTER 7

Known Issues

- Intermittent segfault when running the test-suite (#22)
- Pressing *a* while interacting with a view causes a segfault (#23)
- An Empty window appears when using the snapshot function (#24)

CHAPTER 8

Directory structure

- simphony-paraview – Main package code. - core – Utilities and basic conversion tools.
- examples – Holds examples of visualizing simphony objects with simphony-paraview.
- doc – Documentation related files:
  - source – Sphinx rst source files
  - build – Documentation build directory, if documentation has been generated using the `make` script in the `doc` directory.

CHAPTER 9

User Manual

9.1 SimPhoNy

Paraview tools are available in the simphony library through the visualisation plug-in named `paraview_tools`.

e.g:

```
from simphony.visualisation import paraview_tools
```

9.1.1 Visualizing CUDS

The *show()* function is available to visualise any top level CUDS container. The function will open a window containing a 3D view of the dataset. Interaction is supported using the mouse and keyboard:

keyboard

- **j / t**: toggle between joystick (position sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.
- **3**: toggle the render window into and out of stereo mode. By default, red-blue stereo pairs are created.
- **e**: exit the application.
- **f**: fly to the picked point
- **p**: perform a pick operation.
- **r**: reset the camera view along the current view direction. Centers the actors and moves the camera so that all actors are visible.
- **s**: modify the representation of all actors so that they are surfaces.
- **w**: modify the representation of all actors so that they are wireframe.

mouse

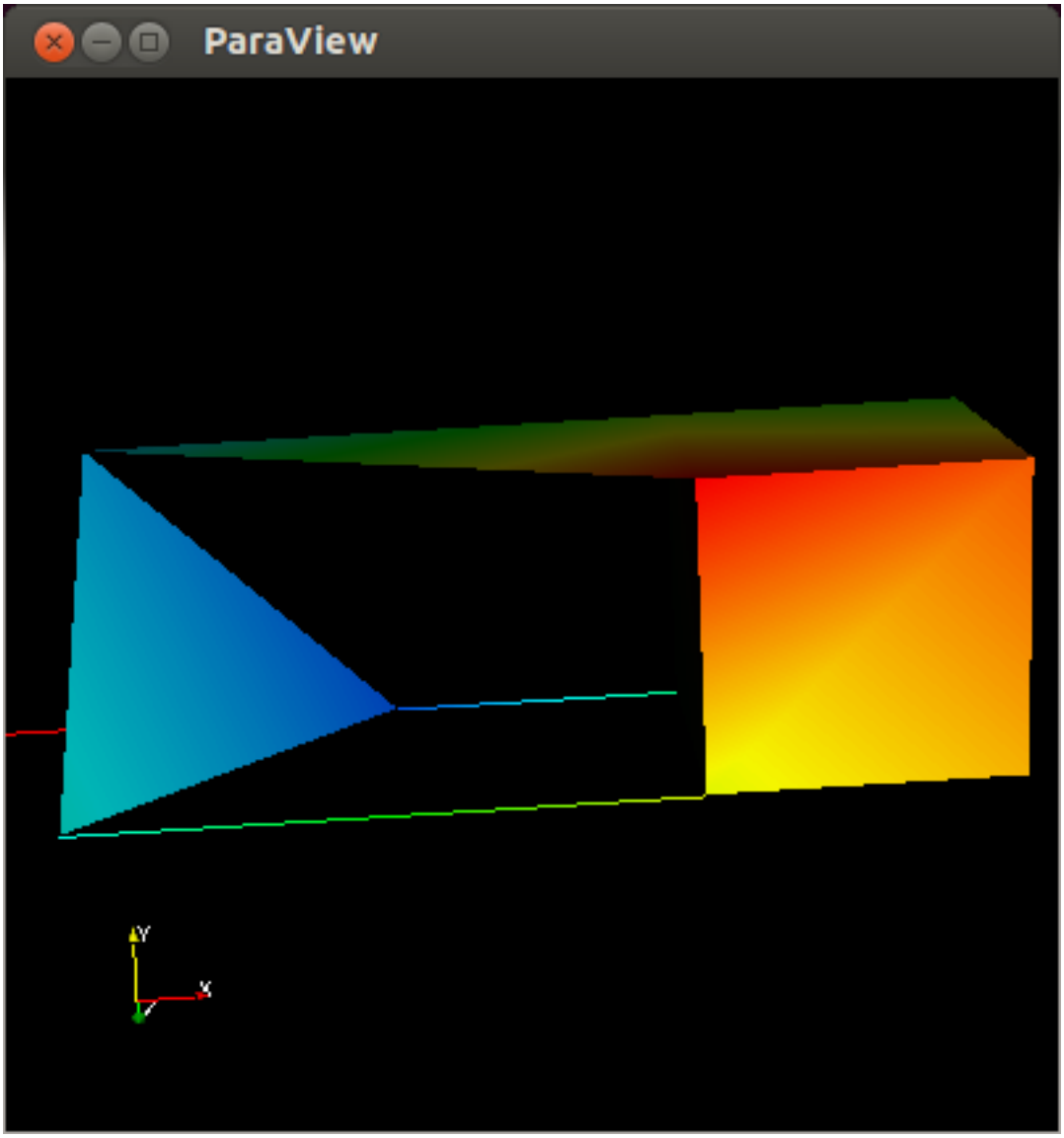
- **Button 1**: rotate the camera around its focal point (if camera mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.
- **Button 2**: pan the camera (if camera mode) or translate the actor (if actor mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse

position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with 2-button mice, pan is defined as <Shift>-Button 1.)

- **Button 3**: zoom the camera (if camera mode) or scale the actor (if actor mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.

Mesh example

```
from numpy import array

from simphony.cuds import Mesh, Point, Cell, Edge, Face
from simphony.core.data_container import DataContainer
from simphony.core.cuba import CUBA


points = array([
    [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1],
    [2, 0, 0], [3, 0, 0], [3, 1, 0], [2, 1, 0],
    [2, 0, 1], [3, 0, 1], [3, 1, 1], [2, 1, 1]],
    'f')

cells = [
    [0, 1, 2, 3],  # tetra
    [4, 5, 6, 7, 8, 9, 10, 11]]  # hex

faces = [[2, 7, 11]]
edges = [[1, 4], [3, 8]]

mesh = Mesh('example')

# add points
uids = mesh.add_points(
    Point(coordinates=point, data=DataContainer(TEMPERATURE=index))
    for index, point in enumerate(points))

# add edges
edge_uids = mesh.add_edges(
    Edge(points=[uids[index] for index in element])
    for index, element in enumerate(edges))

# add faces
face_uids = mesh.add_faces(
    Face(points=[uids[index] for index in element])
    for index, element in enumerate(faces))

# add cells
cell_uids = mesh.add_cells(
    Cell(points=[uids[index] for index in element])
    for index, element in enumerate(cells))

if __name__ == '__main__':
    from simphony.visualisation import paraview_tools

    # Visualise the Mesh object
    paraview_tools.show(mesh, select=(CUBA.TEMPERATURE, 'points'))
```

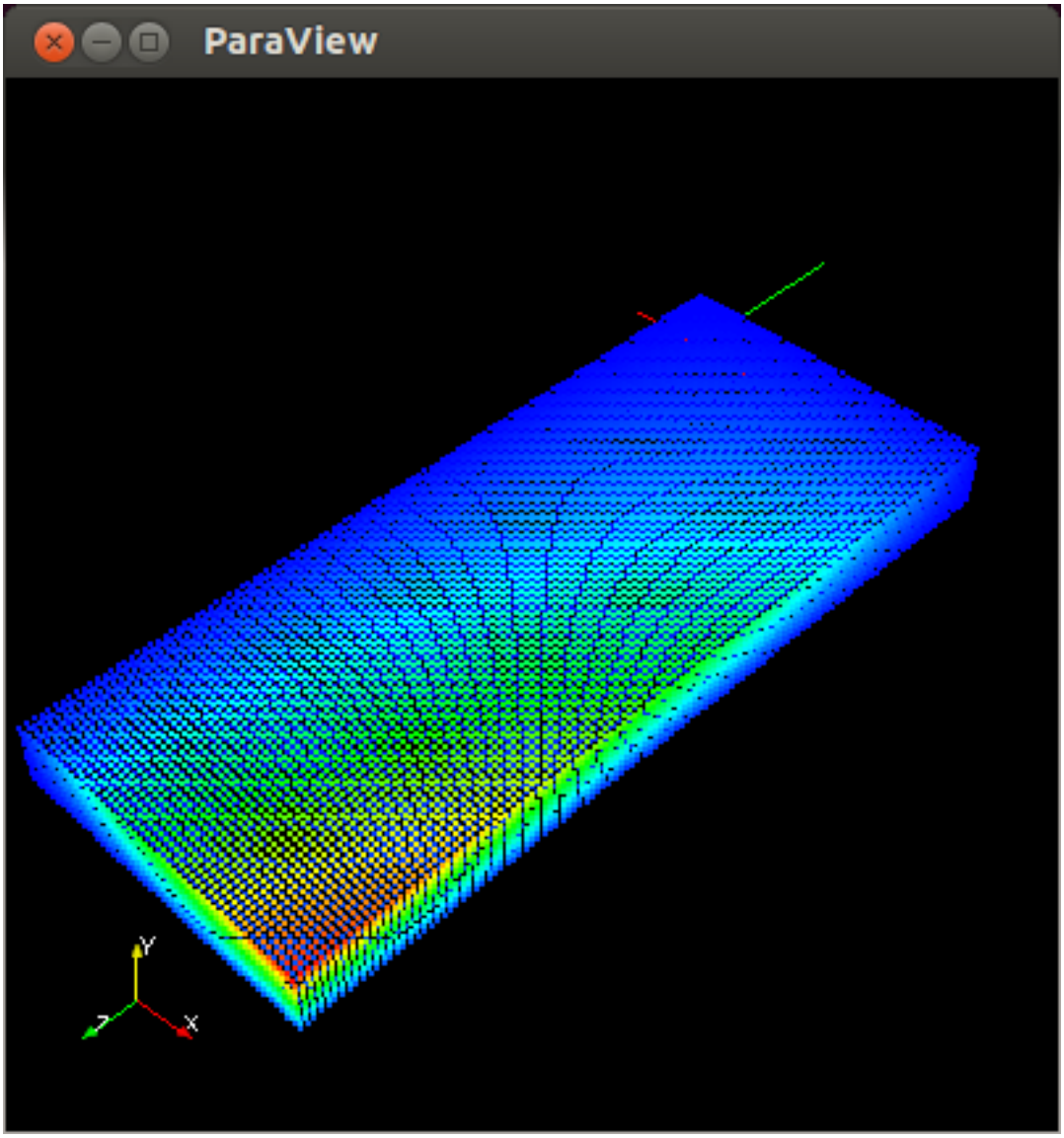

ParaView
Y
X

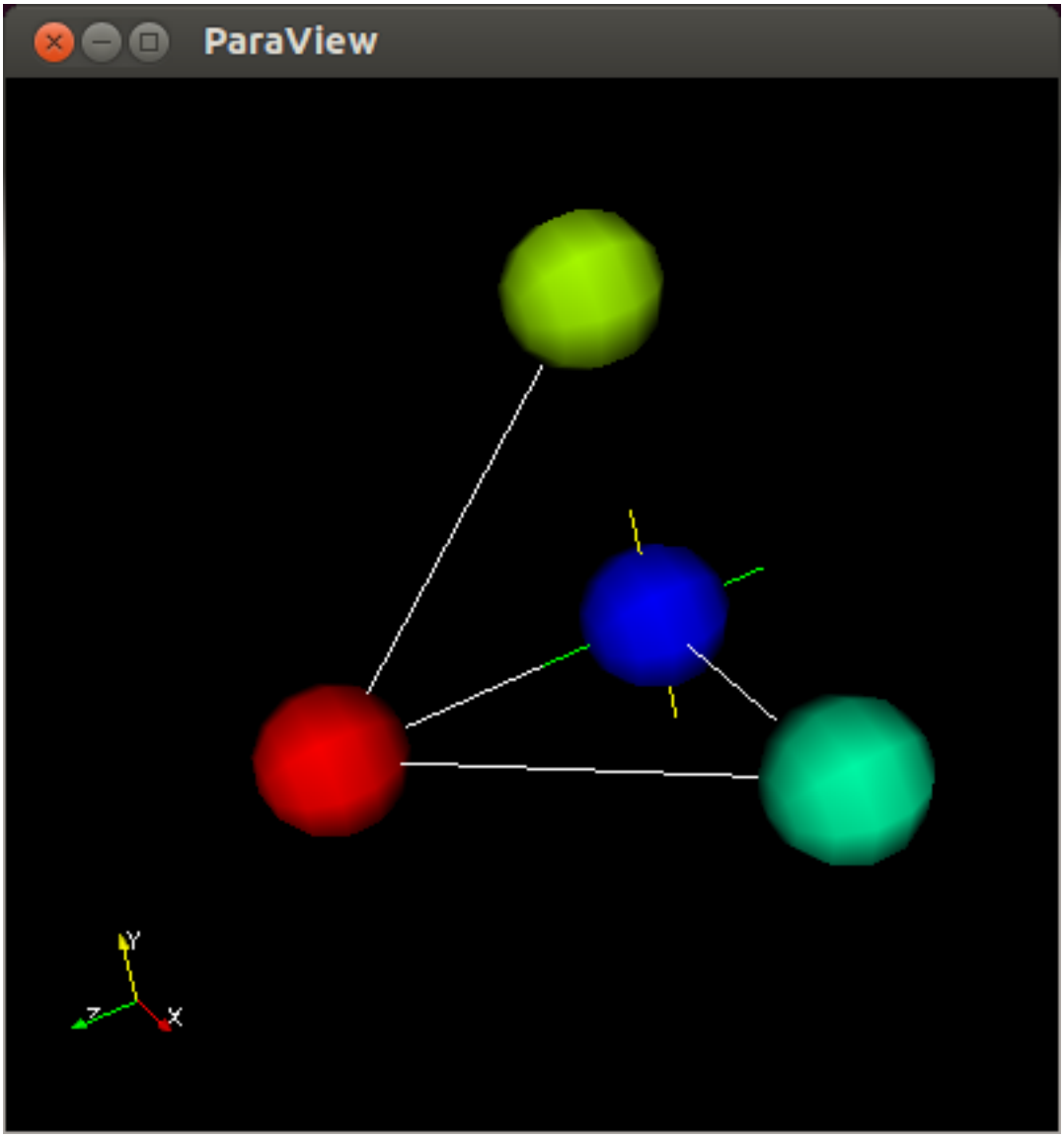
Lattice example

```
import numpy

from simphony.cuds.lattice import make_cubic_lattice
from simphony.core.cuba import CUBA

lattice = make_cubic_lattice('test', 0.1, (50, 10, 120))


def set_temperature(nodes):
    for node in nodes:
        index = numpy.array(node.index) + 1.0
        node.data[CUBA.TEMPERATURE] = numpy.prod(index)
        yield node

lattice.update_nodes(set_temperature(lattice.iter_nodes()))


if __name__ == '__main__':
    from simphony.visualisation import paraview_tools

    # Visualise the Lattice object
    paraview_tools.show(lattice, select=(CUBA.TEMPERATURE, 'nodes'))
```

Particles example

```
from numpy import array

from simphony.cuds import Particles, Particle, Bond
from simphony.core.data_container import DataContainer
from simphony.core.cuba import CUBA


points = array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]], 'f')
bonds = array([[0, 1], [0, 3], [1, 3, 2]])
temperature = array([10., 20., 30., 40.])

particles = Particles('test')
uids = particles.add_particles(
    Particle(
        coordinates=point,
        data=DataContainer(TEMPERATURE=temperature[index]))
    for index, point in enumerate(points))

particles.add_bonds(
    Bond(particles=[uids[index] for index in indices])
    for indices in bonds)


if __name__ == '__main__':
    from simphony.visualisation import paraview_tools

    # Visualise the Particles object
    paraview_tools.show(particles, select=(CUBA.TEMPERATURE, 'particles'))
```


ParaView
Y
Z
X


ParaView
Y
Z
X

CHAPTER 10

API Reference

10.1 Plugin module

This module `simphony_paraview.plugin` provides a set of tools to visualize CUDS objects. The tools are also available as a visualisation plug-in to the simphony library.

`simphony_paraview.show.`**`show`**(*cuds*, *select=None*, *testing=None*)

Show the cuds objects using the default visualisation.

Parameters

- **`cuds`** – A top level cuds object (e.g. a mesh). The method will detect the type of object and create the appropriate visualisation.
- **`select`** (*tuple(CUBA, kind)*) – The (CUBA, kind) selection of the CUBA attribute to use. `kind` can be one of the {'point', 'particles', 'nodes', 'elements', 'bonds'}
- **`testing`** (*callable(obj, event)*) – A callable object that accepts an the interactor object and a time event. The callable will be executed after 1000 msec. This is commonly used for testing. Default value is None

`simphony_paraview.snapshot.`**`snapshot`**(*cuds*, *filename*, *select=None*)

Save a snapshot of the cuds object using the default visualisation.

Parameters

- **`cuds`** – A top level cuds object (e.g. a mesh). The method will detect the type of object and create the appropriate visualisation.
- **`filename`** (*string*) – The filename to use for the output file.
- **`select`** (*tuple(CUBA, kind)*) – The (CUBA, kind) selection of the CUBA attribute to use. `kind` can be one of the {'point', 'particles', 'nodes', 'elements', 'bonds'}

10.2 Core module

A module containing core tools and wrappers for paraview data containers used in simphony_paraview.

Classes

Continued on next page

Table 10.1 – continued from previous page

CUBADataAccumulator([keys, container])	Accumulate data information per CUBA key.

Functions

supported_cuba()	Return a set of currently supported CUBA keys.
default_cuba_value(cuba)	Return the default value of the CUBA key as a scalar or numpy array.
cuds2vtk(cuds)	Create a vtk.Dataset from a CUDS container

Mappings

points2edge()	Return a mapping from number of points to line cells.
points2face()	Return a mapping from number of points to face cells.
points2cell()	Return a mapping from number of points to volume cells.
dataset2writer()	Return a mapping from dataset type to writer instances.
cuba_value_types()	Return a mapping from CUBA to VALUETYPE.

10.2.1 Description

class simphony_paraview.core.cuba_data_accumulator.**CUBADataAccumulator**(*keys=()*, *container=None*)

Bases: object

Accumulate data information per CUBA key.

A collector object that stores :class:DataContainer data into a vtkPointData or vtkCellData array containers where each CUBA key is an array.

The Accumulator has two modes of operation fixed and expand. fixed means that data will be stored for a predefined set of keys on every append call. Where expand will extend the internal table of values whenever a new key is introduced. Missing values will be stored using *default_cuba_value()*.

expand operation

```
>>> accumulator = CUBADataAccumulator():
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.keys
set([<CUBA.TEMPERATURE: 55>])
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys
set([<CUBA.VELOCITY: 21>, <CUBA.TEMPERATURE: 55>])
>>> vtk_to_numpy(accumulator[CUBA.TEMPERATURE])
array([ 34.,  nan,  56.])
>>> vtk_to_numpy(accumulator[CUBA.VELOCITY])
array([[ nan,  nan,  nan], [ 0.1,  0.1,  0.1], [ nan,  nan,  nan]])
```

fixed operation

```
>>> accumulator = CUBADataAccumulator([CUBA.TEMPERATURE, CUBA.PRESSURE])
>>> accumulator.keys
set([<CUBA.PRESSURE: 54>, <CUBA.TEMPERATURE: 55>])
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys
set([<CUBA.PRESSURE: 54>, <CUBA.TEMPERATURE: 55>])
>>> vtk_to_numpy(accumulator[CUBA.TEMPERATURE])
array([ 34.,  nan,  56.])
>>> vtk_to_numpy(accumulator[CUBA.PRESSURE])
[nan, nan, nan]
>>> accumulator[CUBA.VELOCITY]
KeyError(...)
```

Constructor

Parameters **keys** (*list*) – The list of keys that the accumulator should care about. Providing this value at initialisation sets up the accumulator to operate in `fixed` mode. If no keys are provided then accumulator operates in `expand` mode.

__getitem__(*key*)

Get the list of accumulated values for the CUBA key.

Parameters **key** (*CUBA*) – A CUBA Enum value

Returns

result (*vtkDataArray*) –

An array of data values collected for `key`. Missing values are assigned the default value as returned by **:function:'~.default_cuba_value'**.

Raises **KeyError** – When values for the requested CUBA key do not exist.

__len__()

The number of values that are stored per key

append(*data*)

Append data from a `DataContainer`.

If the accumulator operates in `fixed` mode:

- Any keys in `self.keys()` that have values in `data` will be stored (appended to the related key arrays).
- Missing keys will be stored with the returned value of the *default_cuba_value()*.

If the accumulator operates in `expand` mode:

- Any new keys in *Data* will be added to the `self.keys()` list and the related list of values with length equal to the current record size will be initialised with the returned value of the *default_cuba_value()*.
- Any keys in the modified `self.keys()` that have values in `data` will be stored (appended to the list of the related key).
- Missing keys will be stored with the returned value of the *default_cuba_value()*.

Parameters **data** (*DataContainer*) – The data information to append.

keys
The set of CUBA keys that this accumulator contains.

---

simphony_paraview.core.cuba_utils.**supported_cuba**()
Return a set of currently supported CUBA keys.

simphony_paraview.core.cuba_utils.**default_cuba_value**(*cuba*)
Return the default value of the CUBA key as a scalar or numpy array.

Int type values have `-1` as default, while float type values have `numpy.nan`.

simphony_paraview.core.cuds2vtk.**cuds2vtk**(*cuds*)
Create a vtk.Dataset from a CUDS container

---

simphony_paraview.core.constants.**points2edge**()
Return a mapping from number of points to line cells.

simphony_paraview.core.constants.**points2face**()
Return a mapping from number of points to face cells.

simphony_paraview.core.constants.**points2cell**()
Return a mapping from number of points to volume cells.

simphony_paraview.core.constants.**dataset2writer**()
Return a mapping from dataset type to writer instances.

simphony_paraview.core.constants.**cuba_value_types**()
Return a mapping from CUBA to VALUETYPE.

Symbols

A

C

D

K

P

S