SimPhoNy-Mayavi Documentation

Release 0.6.0.dev21

SimPhoNy FP7 Collaboration

January 30, 2017

Contents

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

CHAPTER 1

Repository

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

CHAPTER 2

Requirements

- mayavi[app] >= 4.4.0
- simphony[H5IO] >= 0.3.0

2.1 Optional requirements

To support testing, you will need the following packages:

- PIL
- mock

Alternatively unning **`pip install -r dev-requirements.txt`** should install the packages needed for development purposes.

To support the documentation built you need the following packages:

- sphinx >= 1.2.3
- sectiondoc commit 8a0c2be, https://github.com/enthought/sectiondoc
- trait-documenter, https://github.com/enthought/trait-documenter

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 `mayavi` visualization plugin module by:

```
from simphony.visualisation import mayavi_tools
mayavi_tools.show(cuds)
```

Note:

- It is also recommended that the user uses qt4 as the user interface backends by setting the environment variable `ETS_TOOLKIT`. In Bash, that is:

```
export ETS_TOOLKIT=qt4
```

CHAPTER 7

Known Issues

- *Segmentation fault during loading or running test suites*

  This may be caused by installing BOTH simphony-paraview and simphony-mayavi in the same environment. Since paraview and mayavi use different versions of VTK, work-around is limited. Here are two possible solutions.

  - If you don't need both simphony-mayavi and simphony-paraview, uninstall one of them, e.g.:

```
pip uninstall simphony-paraview
```

  - If you must retain both plugins, choose to remove one of them from the `simphony.visualisation` entry points. The plugin removed from `simphony.visualisation` is still accessible via `import simphony_paraview.plugin` or `import simphony_mayavi.plugin`. Notice that this change would cause plugin loading tests to fail.

CHAPTER 8

Directory structure

- simphony-mayavi – Main package folder.
  - sources – Wrap CUDS objects to provide Mayavi Sources.
  - cuds – Wrap VTK Dataset objects to provide the CUDS container api.
  - core – Utility classes and tools to manipulate vtk and cuds objects.
  - plugins – GUI for Mayavi2
  - modules – default modules for visualising SimPhoNy objects
  - examples – Holds examples of loading and visualising SimPhoNy objects with simphony-mayavi.
- doc – Documentation related files: - The rst source files for the documentation

CHAPTER 9

User Manual

9.1 SimPhoNy

Mayavi tools are available in the simphony library through the visualisation plug-in named `mayavi_tools`.

e.g:

```
from simphony.visualisation import mayavi_tools
```

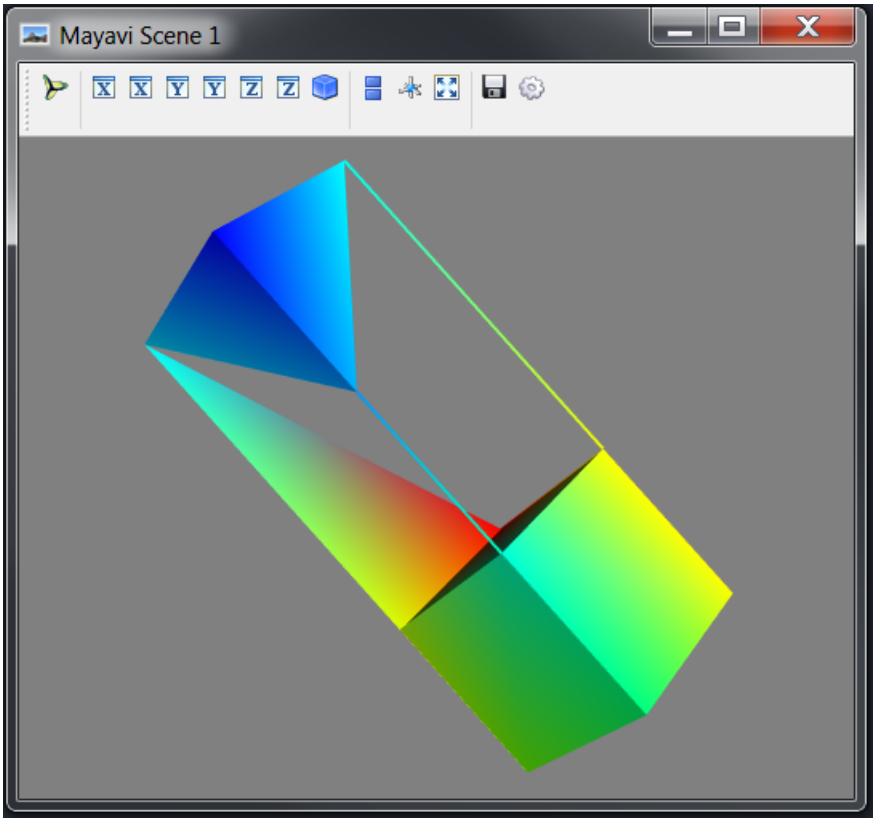
9.1.1 Visualizing CUDS

The `show()` function is available to visualise any top level CUDS container. The function will open a window containing a 3D view and a mayavi toolbar. Interaction allows the common mayavi operations.

Mesh example

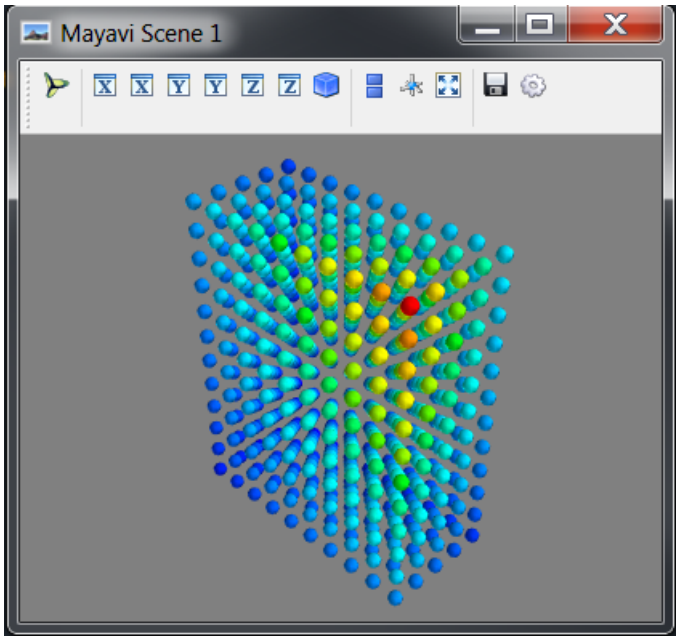
```
from numpy import array

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


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
point_iter = (Point(coordinates=point, data=DataContainer(TEMPERATURE=index))
              for index, point in enumerate(points))
```

```
uids = mesh.add(point_iter)

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

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

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


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

    # Visualise the Mesh object
    mayavi_tools.show(mesh)
```



Lattice example

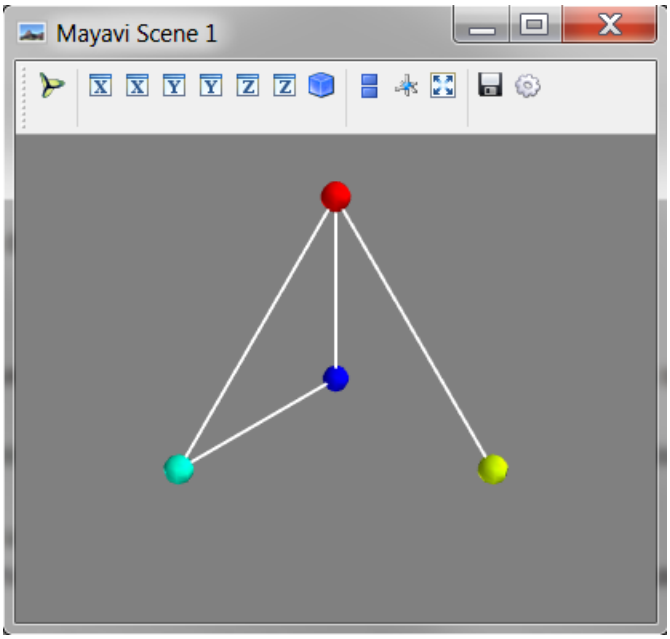
```
import numpy

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

lattice = make_cubic_lattice('test', 0.1, (5, 10, 12))

new_nodes = []
for node in lattice.iter(item_type=CUBA.NODE):
    index = numpy.array(node.index) + 1.0
    node.data[CUBA.TEMPERATURE] = numpy.prod(index)
    new_nodes.append(node)

lattice.update(new_nodes)


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

    # Visualise the Lattice object
    mayavi_tools.show(lattice)
```



Particles example

```
from numpy import array

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

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')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = particles.add(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
particles.add(bond_iter)

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

    # Visualise the Particles object
    mayavi_tools.show(particles)
```



9.1.2 Create VTK backed CUDS

Three objects (i.e `VTKMesh`, `VTKLattice`, `VTKParticles`) that wrap a VTK dataset and provide the CUDS top level container API are also available. The vtk backed objects are expected to provide memory and some speed advantages when Mayavi aided visualisation and processing is a major part of the working session. The provided examples are equivalent to the ones in section *Visualizing CUDS*.

Note: Note all CUBA keys are supported for the *data* attribute of the contained items. Please see documentation for more details.

VTK Mesh example

```
from numpy import array

from simphony.cuds.mesh import Point, Cell, Edge, Face
from simphony.core.data_container import DataContainer
from simphony.visualisation import mayavi_tools


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 = mayavi_tools.VTKMesh('example')

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

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

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

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


if __name__ == '__main__':

    # Visualise the Mesh object
    mayavi_tools.show(mesh)
```

VTK Lattice example

```
import numpy

from simphony.core.cuba import CUBA
from simphony.cuds.primitive_cell import PrimitiveCell
```

```
from simphony.visualisation import mayavi_tools

cubic = mayavi_tools.VTKLattice.empty(
    "test", PrimitiveCell.for_cubic_lattice(0.1),
    (5, 10, 12), (0, 0, 0))

lattice = cubic

new_nodes = []
for node in lattice.iter(item_type=CUBA.NODE):
    index = numpy.array(node.index) + 1.0
    node.data[CUBA.TEMPERATURE] = numpy.prod(index)
    new_nodes.append(node)

lattice.update(new_nodes)


if __name__ == '__main__':

    # Visualise the Lattice object
    mayavi_tools.show(lattice)
```

VTK Particles example

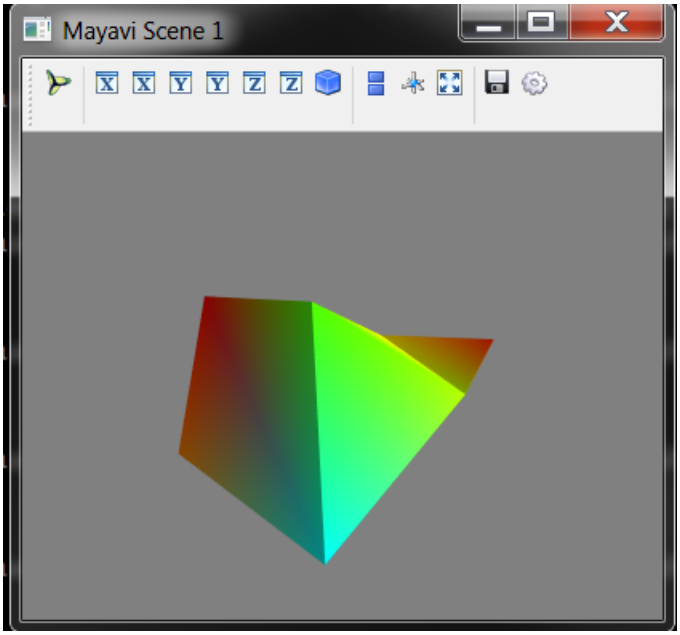
```
from numpy import array

from simphony.core.data_container import DataContainer
from simphony.cuds.particles import Particle, Bond
from simphony.visualisation import mayavi_tools

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 = mayavi_tools.VTKParticles('test')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = particles.add(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
particles.add(bond_iter)

if __name__ == '__main__':

    # Visualise the Particles object
    mayavi_tools.show(particles)
```

9.1.3 Adapting VTK datasets

The `adapt2cuds()` function is available to wrap common VTK datsets into top level CUDS containers. The function will attempt to automatically adapt the (t)vtk Dataset into a CUDS container. When automatic conversion fails the user can always force the kind of the container to adapt into. Furthermore, the user can define the mapping of the included attribute data into corresponding CUBA keys (a common case for vtk datasets that come from vtk reader objects).

Example

```
from numpy import array, random
from tvtk.api import tvtk
from simphony.core.cuba import CUBA
from simphony.visualisation import mayavi_tools


def create_unstructured_grid(array_name='scalars'):
    points = array(
        [[0, 1.2, 0.6], [1, 0, 0], [0, 1, 0], [1, 1, 1],  # tetra
         [1, 0, -0.5], [2, 0, 0], [2, 1.5, 0], [0, 1, 0],
         [1, 0, 0], [1.5, -0.2, 1], [1.6, 1, 1.5], [1, 1, 1]], 'f')  # Hex
    cells = array(
        [4, 0, 1, 2, 3,  # tetra
         8, 4, 5, 6, 7, 8, 9, 10, 11])  # hex
    offset = array([0, 5])
    tetra_type = tvtk.Tetra().cell_type  # VTK_TETRA == 10
    hex_type = tvtk.Hexahedron().cell_type  # VTK_HEXAHEDRON == 12
    cell_types = array([tetra_type, hex_type])
    cell_array = tvtk.CellArray()
    cell_array.set_cells(2, cells)
    ug = tvtk.UnstructuredGrid(points=points)
    ug.set_cells(cell_types, offset, cell_array)
    scalars = random.random(points.shape[0])
    ug.point_data.scalars = scalars
    ug.point_data.scalars.name = array_name
    scalars = random.random((2, 1))
    ug.cell_data.scalars = scalars
    ug.cell_data.scalars.name = array_name
    return ug

# Create an example
vtk_dataset = create_unstructured_grid()

# Adapt to a mesh by converting the scalars attribute to TEMPERATURE
container = mayavi_tools.adapt2cuds(
    vtk_dataset, 'test',
    rename_arrays={'scalars': CUBA.TEMPERATURE})

if __name__ == '__main__':

    # Visualise the Lattice object
    mayavi_tools.show(container)
```



9.1.4 Loading into CUDS

The `load()` function is available to load mayavi readable files (e.g. VTK xml format) into top level CUDS containers. Using `load` the user can import inside their simulation scripts files that have been created by other simulation application and export data into one of the Mayavi supported formats.

9.2 Mayavi2

The Simphony-Mayavi library provides a plugin for Mayavi2 to easily create mayavi `Source` instances from SimPhoNy CUDS datasets and files.

Any CUDS datastet can be adapted as a mayavi `Source` using `CUDSSource`. If CUDS datasets are to be loaded from a CUDS native file, it maybe easier to use `CUDSFileSource` which does the loading for you. Similarly, if the CUDS datasets are from a SimPhoNy engine wrapper, `EngineSource` may be used. All of these `Source` objects provide an `update` function that allows the user to refresh visualisation once the CUDS dataset is modified.

With the provided tools one can use the SimPhoNy libraries to work inside the Mayavi2 application, as it is demonstrated in the examples.

9.2.1 Open CUDS Files in Mayavi2

In order for mayavi2 to understand `*.cuds` files one needs to make sure that the simpony_mayavi plugin has been selected and activated in the Mayavi2 preferences dialog.

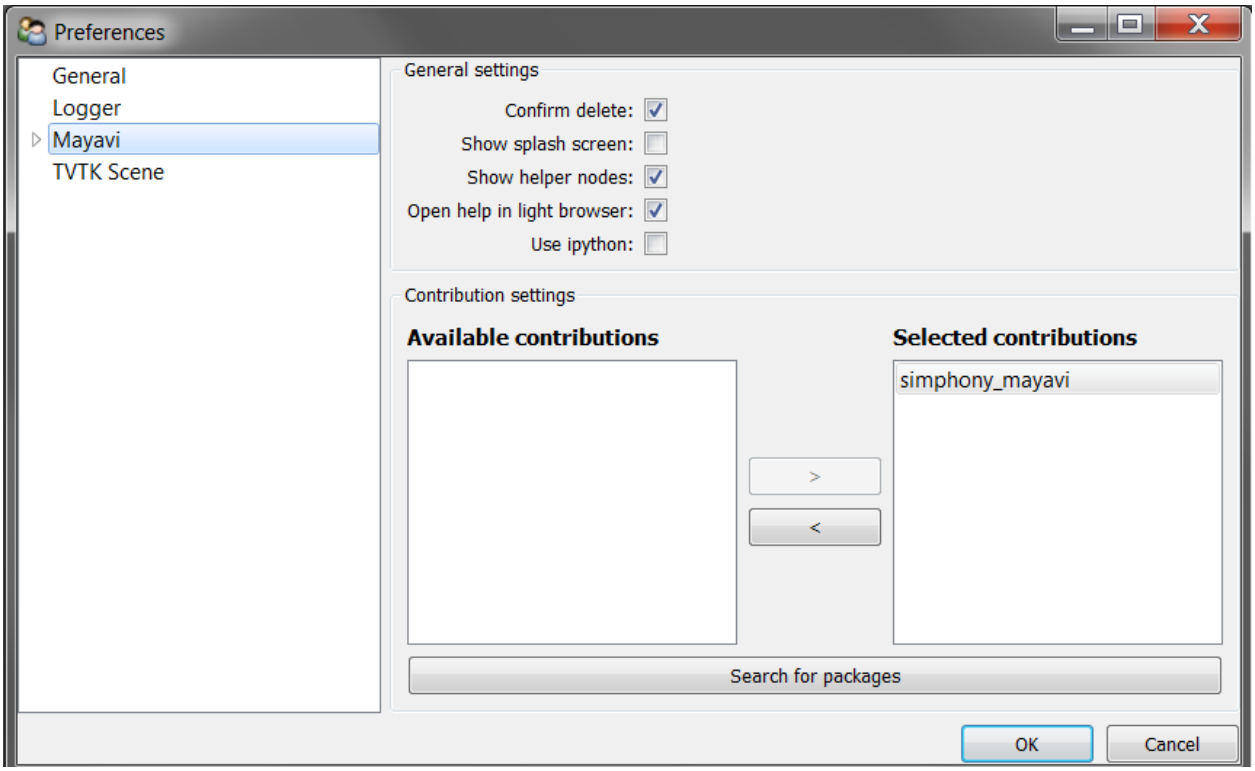
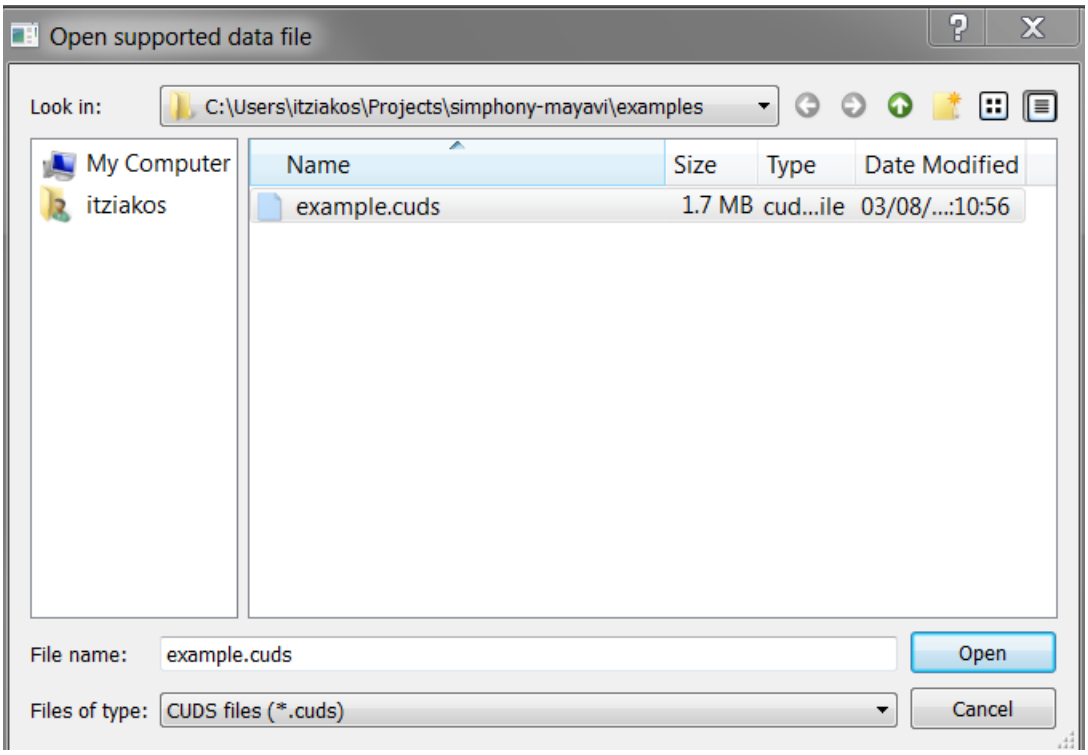


Fig. 9.1: Cuds files are supported in the `Open File..` dialog. After running the `provided example`, load the `example.cuds` file into Mayavi2.



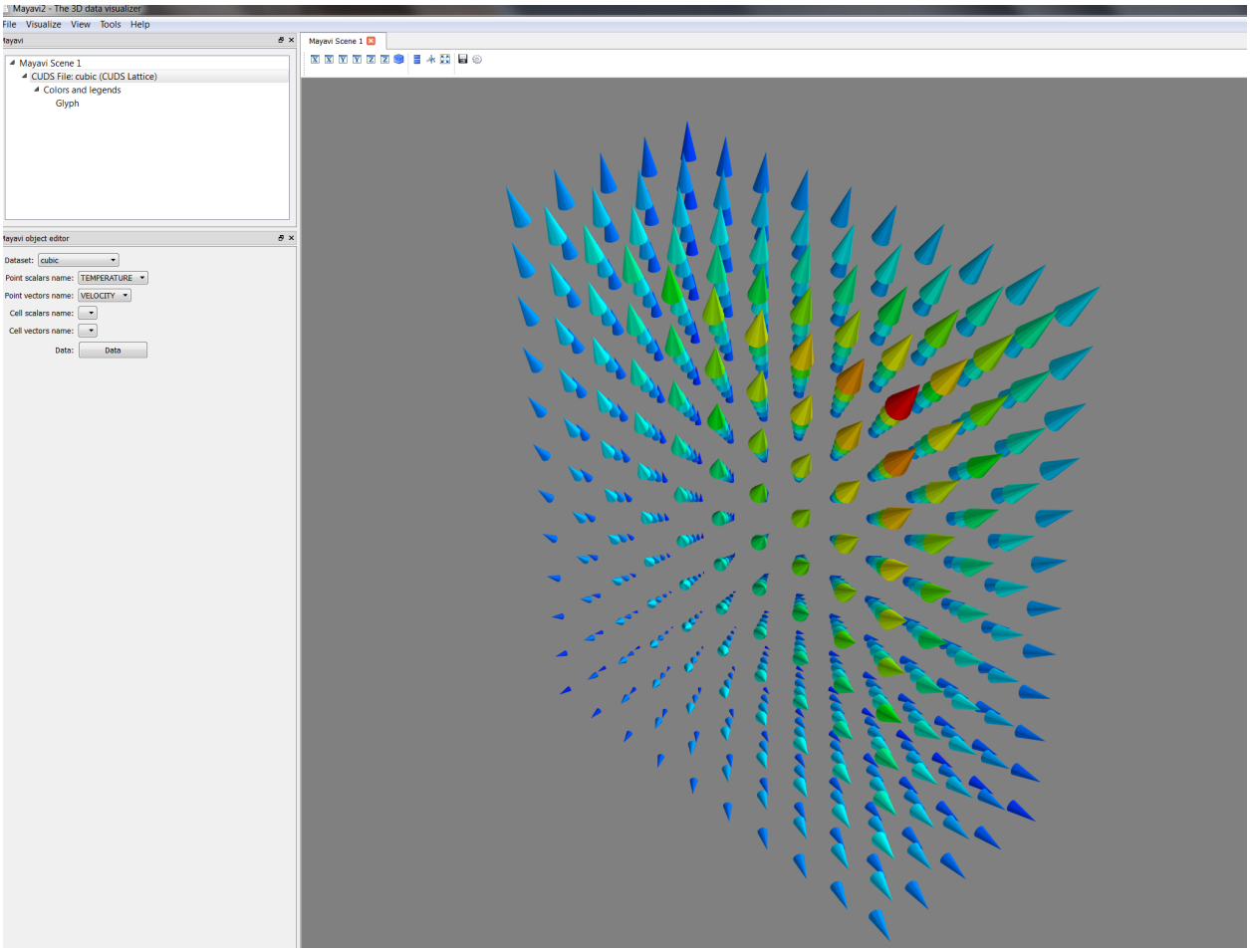


Fig. 9.2: When loaded a CUDSFile is converted into a Mayavi Source and the user can add normal Mayavi modules to visualise the currently selected CUDS container from the available containers in the file.

In the example we load the container named `cubic` and attach the Glyph module to draw a cone at each point to visualise `TEMPERATURE` and `VELOCITY` in the Mayavi Scene.

9.2.2 View CUDS in Mayavi2

Source from a CUDS Mesh

```
from numpy import array
from mayavi.scripts import mayavi2

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


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]]

container = Mesh('test')

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

# add edges
edge_iter = (Edge(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 20))
             for index, element in enumerate(edges))
edge_uids = container.add(edge_iter)

# add faces
face_iter = (Face(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 30))
             for index, element in enumerate(faces))
face_uids = container.add(face_iter)

# add cells
cell_iter = (Cell(points=[uids[index] for index in element],
                  data=DataContainer(TEMPERATURE=index + 40))
             for index, element in enumerate(cells))
cell_uids = container.add(cell_iter)


# Now view the data.
@mayavi2.standalone
def view():
    from mayavi.modules.surface import Surface
    from simphony_mayavi.sources.api import CUDSSource

    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=container)
```

```
    mayavi.add_source(src)  # noqa
    s = Surface()
    mayavi.add_module(s)  # noqa

if __name__ == '__main__':
    view()
```

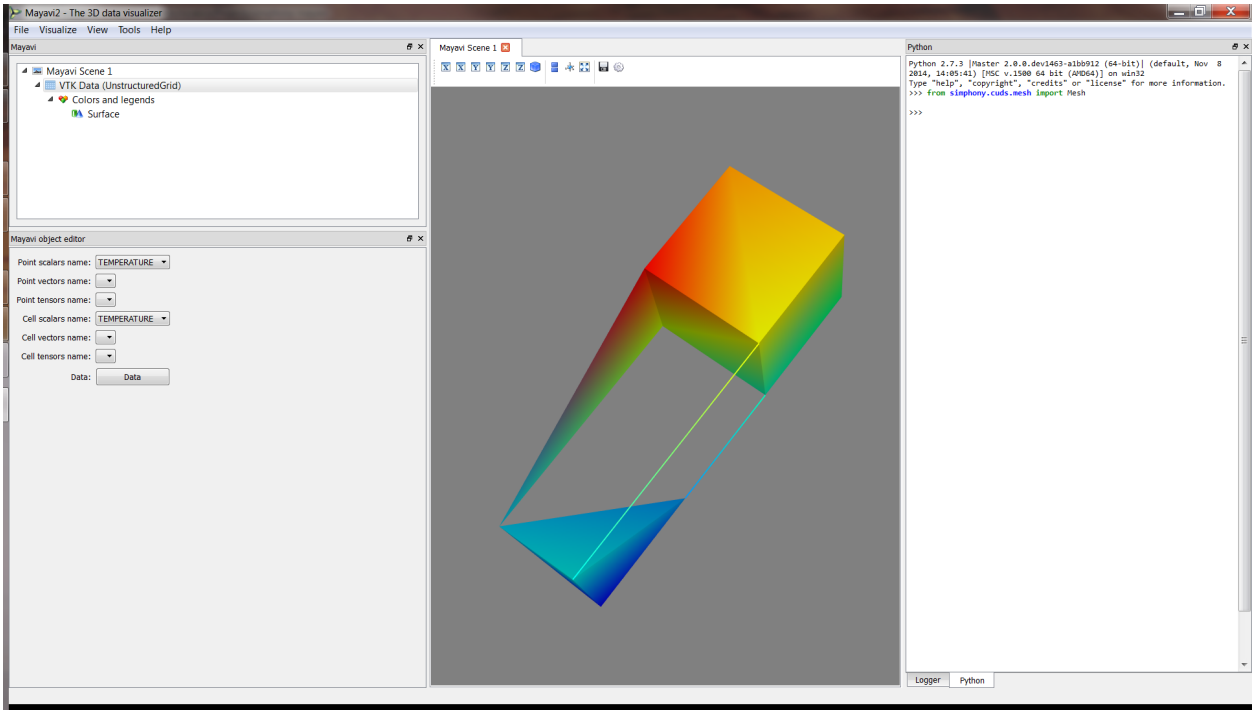


Fig. 9.3: Use the provided example to create a CUDS Mesh and visualise directly in Mayavi2.

Source from a CUDS Lattice

```
import numpy

from mayavi.scripts import mayavi2
from simphony.cuds.lattice import make_cubic_lattice
from simphony.core.cuba import CUBA

cubic = make_cubic_lattice("cubic", 0.1, (5, 10, 12))


def add_temperature(lattice):
    new_nodes = []
    for node in lattice.iter(item_type=CUBA.NODE):
        index = numpy.array(node.index) + 1.0
        node.data[CUBA.TEMPERATURE] = numpy.prod(index)
        new_nodes.append(node)
    lattice.update(new_nodes)

add_temperature(cubic)
```

```
# Now view the data.
@mayavi2.standalone
def view(lattice):
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import CUDSSource
    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=lattice)
    mayavi.add_source(src)  # noqa
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.02
    g.glyph.scale_mode = 'data_scaling_off'
    mayavi.add_module(g)  # noqa

if __name__ == '__main__':
    view(cubic)
```

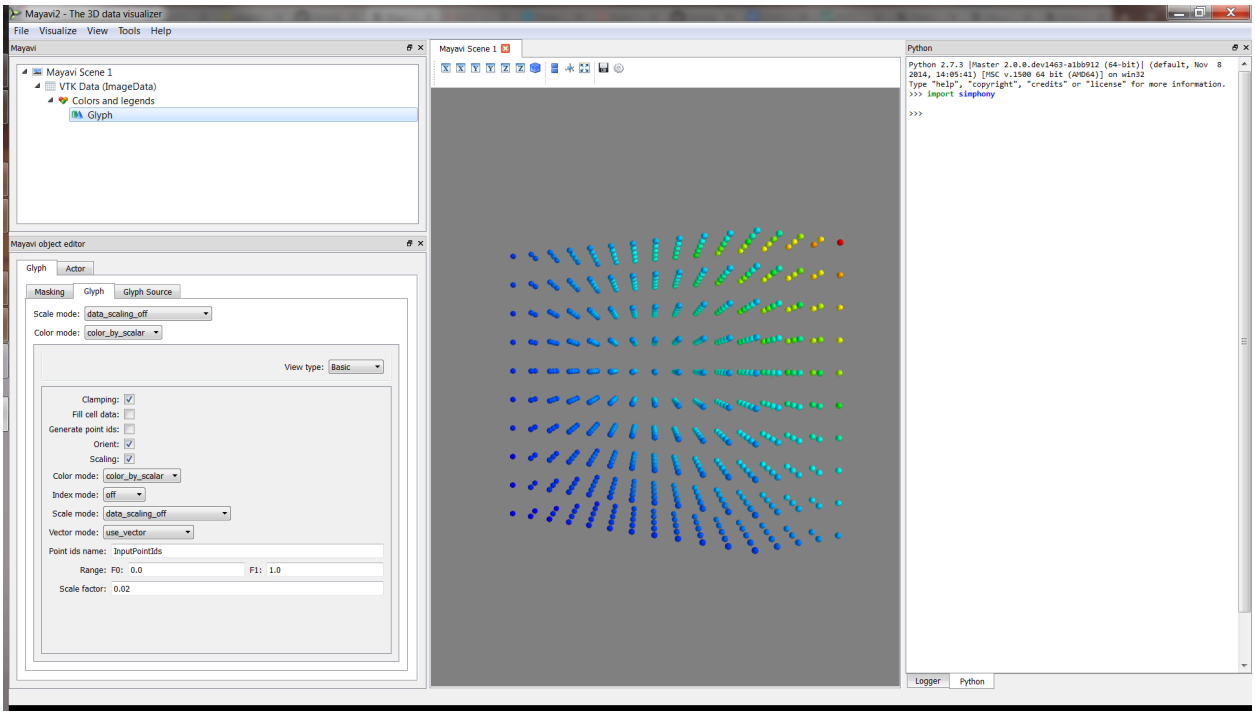


Fig. 9.4: Use the provided example to create a CUDS Lattice and visualise directly in Mayavi2.

Source for a CUDS Particles

```
from numpy import array
from mayavi.scripts import mayavi2

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

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.])
```

```
container = Particles('test')

# add particles
particle_iter = (Particle(coordinates=point,
                          data=DataContainer(TEMPERATURE=temperature[index]))
                 for index, point in enumerate(points))
uids = container.add(particle_iter)

# add bonds
bond_iter = (Bond(particles=[uids[index] for index in indices])
             for indices in bonds)
container.add(bond_iter)


# Now view the data.
@mayavi2.standalone
def view():
    from mayavi.modules.surface import Surface
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import CUDSSource

    mayavi.new_scene()  # noqa
    src = CUDSSource(cuds=container)
    mayavi.add_source(src)  # noqa
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.05
    g.glyph.scale_mode = 'data_scaling_off'
    s = Surface()
    s.actor.mapper.scalar_visibility = False

    mayavi.add_module(g)  # noqa
    mayavi.add_module(s)  # noqa

if __name__ == '__main__':
    view()
```

Source from a CUDS native file

```
from contextlib import closing

import numpy
from mayavi.scripts import mayavi2

from simphony.core.cuba import CUBA
from simphony.cuds.lattice import (make_hexagonal_lattice,
                                   make_orthorhombic_lattice)
from simphony.io.h5_cuds import H5CUDS

# create some datasets to be saved in a file
hexagonal = make_hexagonal_lattice(
    'hexagonal', 0.1, 0.1, (5, 5, 5), (5, 4, 0))

orthorhombic = make_orthorhombic_lattice(
    'orthorhombic', (0.1, 0.2, 0.3), (5, 5, 5), (5, 4, 0))
```

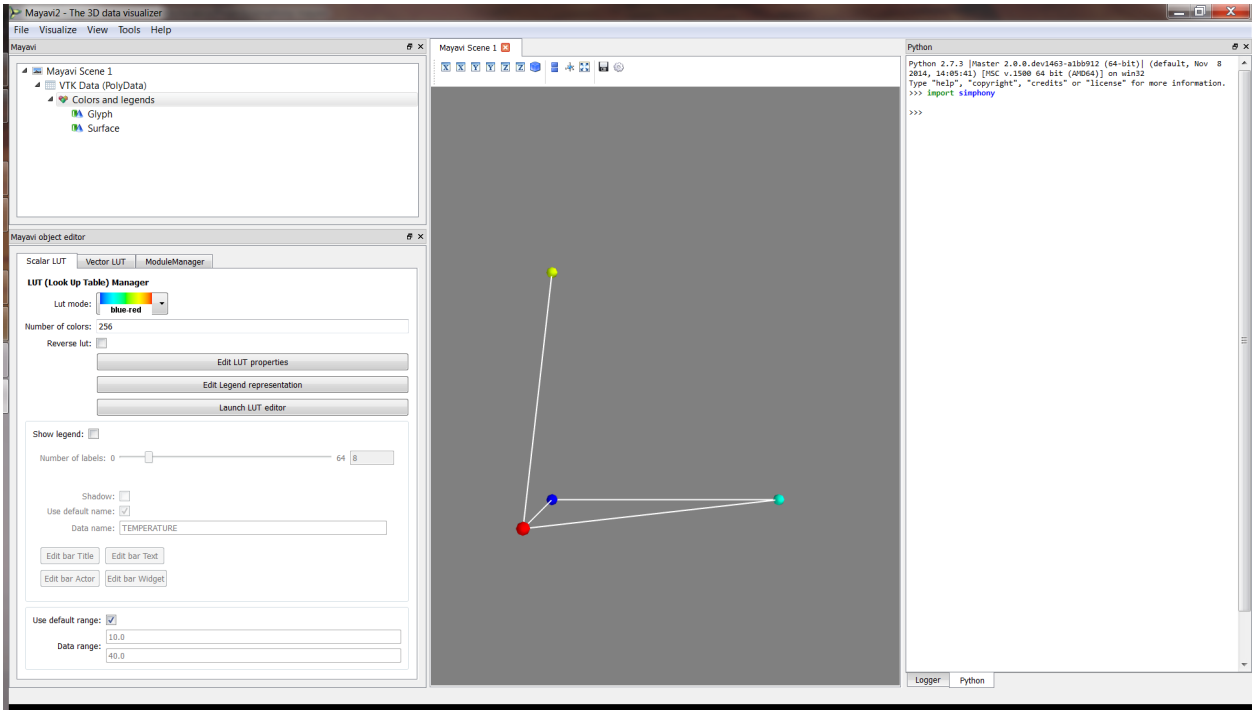


Fig. 9.5: Use the provided example to create a CUDS Particles and visualise directly in Mayavi2.

```
def add_temperature(lattice):
    new_nodes = []
    for node in lattice.iter(item_type=CUBA.NODE):
        index = numpy.array(node.index) + 1.0
        node.data[CUBA.TEMPERATURE] = numpy.prod(index)
        new_nodes.append(node)
    lattice.update(new_nodes)

# add some scalar data (i.e. temperature)
add_temperature(hexagonal)
add_temperature(orthorhombic)

# save the data into cuds.
with closing(H5CUDS.open('lattices.cuds', 'w')) as handle:
    handle.add_dataset(hexagonal)
    handle.add_dataset(orthorhombic)


@mayavi2.standalone
def view():
    from mayavi import mlab
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import CUDSFileSource

    mayavi.new_scene()

    # Mayavi Source
    src = CUDSFileSource()
    src.initialize('lattices.cuds')
```

```
    # choose a dataset for display
    src.dataset = 'orthorhombic'

    mayavi.add_source(src)

    # customise the visualisation
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.05
    g.glyph.scale_mode = 'data_scaling_off'
    mayavi.add_module(g)

    # add legend
    module_manager = src.children[0]
    module_manager.scalar_lut_manager.show_scalar_bar = True
    module_manager.scalar_lut_manager.show_legend = True

    # customise the camera
    mlab.view(63., 38., 3., [5., 4., 0.])


if __name__ == '__main__':
    view()
```

Source from a SimPhoNy engine wrapper

```
from mayavi.scripts import mayavi2
from simphony_mayavi.tests.testing_utils import DummyEngine

# Comply to SimPhoNy modeling engine API
engine_wrapper = DummyEngine()


@mayavi2.standalone
def view():
    from mayavi.modules.glyph import Glyph
    from simphony_mayavi.sources.api import EngineSource
    from mayavi import mlab

    # Define EngineSource, choose dataset
    src = EngineSource(engine=engine_wrapper,
                       dataset="particles")

    # choose the CUBA attribute for display
    src.point_scalars_name = "TEMPERATURE"

    mayavi.add_source(src)

    # customise the visualisation
    g = Glyph()
    gs = g.glyph.glyph_source
    gs.glyph_source = gs.glyph_dict['sphere_source']
    g.glyph.glyph.scale_factor = 0.2
    g.glyph.scale_mode = 'data_scaling_off'
    mayavi.add_module(g)
```

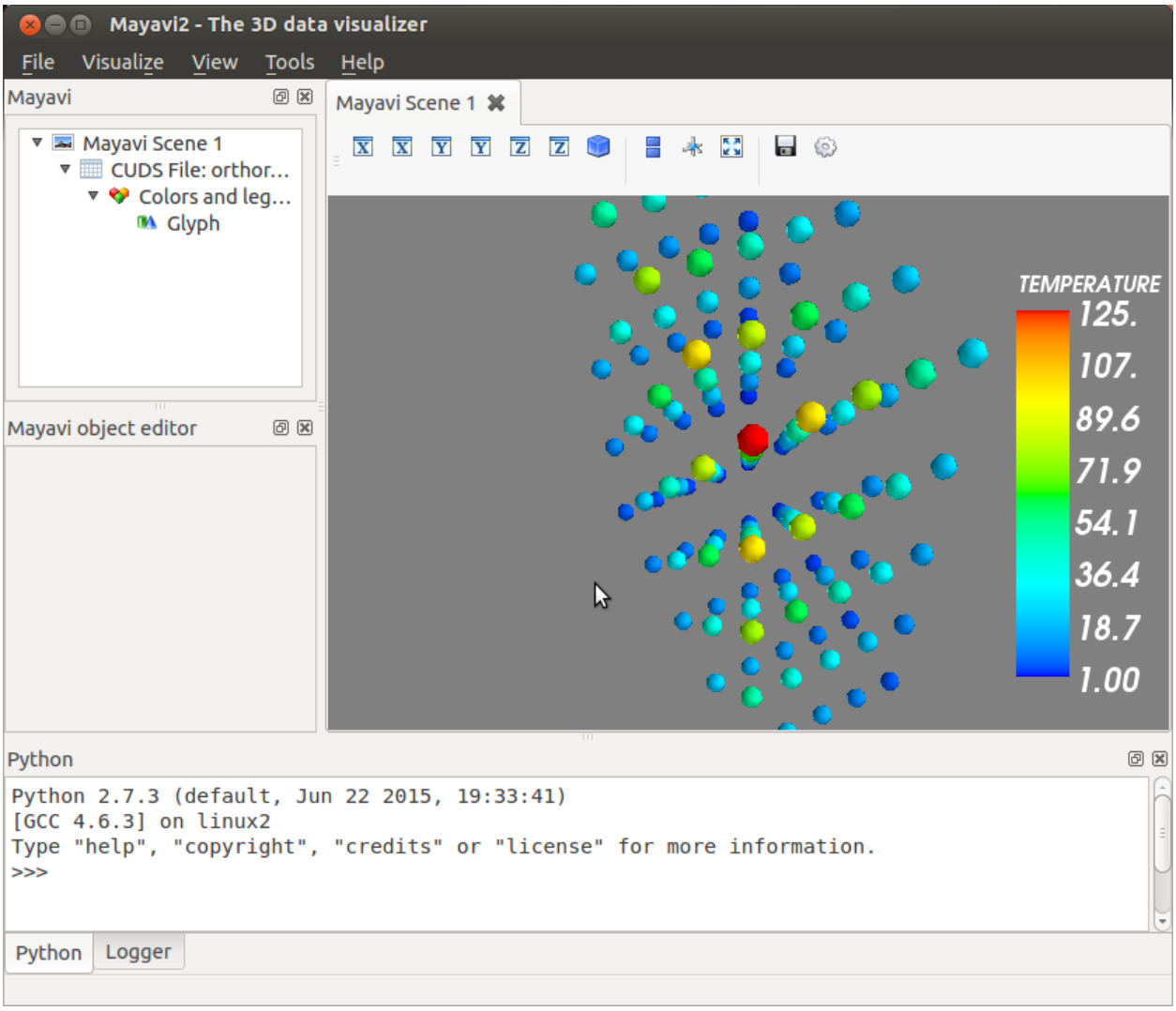


Fig. 9.6: Use the provided example to load data from a CUDS file and visualise directly in Mayavi2.

```
    # add legend
    module_manager = src.children[0]
    module_manager.scalar_lut_manager.show_scalar_bar = True
    module_manager.scalar_lut_manager.show_legend = True

    # set camera
    mlab.view(-65., 60., 14., [1.5, 2., 2.5])


if __name__ == '__main__':
    view()
```

9.3 Interacting with Simphony Engine

9.3.1 Batch scripting

Mayavi *mlab* library provides an easy way to visualise data in a script in ways similar to the matplotlib's pylab module. As illustrated with examples in *View CUDS in Mayavi2*, the user can easily adapt SimPhoNy CUDS datasets, files, or engines into a native Mayavi *Source* object, and then make use of the *mlab* library to set up the visualisation. More details on how to use *mlab* can be found on its documentation.

Here is an example for visualising a dataset from a SimPhoNy engine, updating the visualisation and saving the image while the engine is being run.

```
from mayavi import mlab

from simphony_mayavi.tests.testing_utils import DummyEngine
from simphony_mayavi.sources.api import EngineSource


# Comply to SimPhoNy modeling engine API
engine_wrapper = DummyEngine()

# Define EngineSource, choose dataset
src = EngineSource(engine=engine_wrapper,
                   dataset="particles")

# choose the CUBA attribute for display
src.point_scalars_name = "TEMPERATURE"

# use glyph to show the particles
mlab.pipeline.glyph(src, scale_factor=0.2, scale_mode='none')

# add legend
module_manager = src.children[0]
module_manager.scalar_lut_manager.show_scalar_bar = True
module_manager.scalar_lut_manager.show_legend = True

# set camera
mlab.view(-65., 60., 14., [1.5, 2., 2.5])

# save the figure
mlab.savefig("figures/particles_001.png")

# run the engine and update the visualisatioin
```

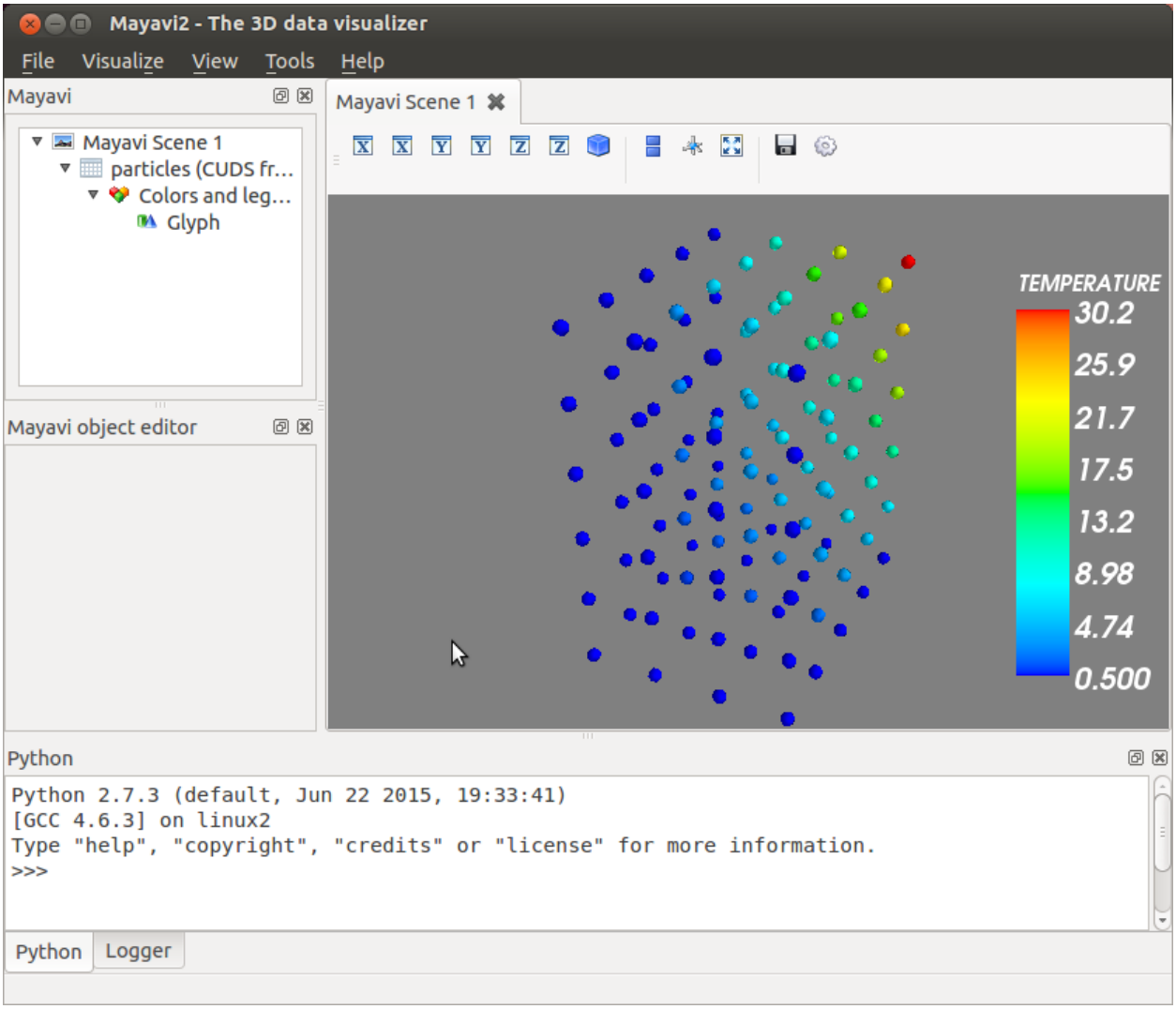


Fig. 9.7: Use the provided example to load data from a SimPhoNy engine and visualise directly in Mayavi2.

In the Mayavi2 embedded python interpreter, the user can access the SimPhoNy engine wrapper associated with the `EngineSource` via its `engine` attribute:

```
# Retrieve the EngineSource
source = engine.scenes[0].children[0]

# The SimPhoNy engine wrapper originally defined
source.engine

# Run the engine
source.engine.run()

# update the visualisation
source.update()
```

```
for i in range(2, 20):
    engine_wrapper.run()
    src.update()
    mlab.savefig("figures/particles_{:03d}.png".format(i))
```

Making a video is just a step away!

9.3.2 Interactive scripting

`EngineManagerStandaloneUI` provides a user-friendly and interactive approach to manage multiple engines, visualise datasets from a particular engine, locally run an engine and animate the results.

Multiple engines can be added to or removed from the manager using `add_engine` and `remove_engine`.

Example (Interactive: EngineManagerStandaloneUI)

```
from simphony_mayavi.tests.testing_utils import DummyEngine
from simphony.visualisation import mayavi_tools

# GUI for Interacting with the engine and mayavi
gui = mayavi_tools.EngineManagerStandaloneUI()

gui.show_config()

# you can add an engine from the python shell
engine_wrapper = DummyEngine()

# "test" is used as a label for representing the engine in the GUI
gui.add_engine("test", engine_wrapper)

# you can remove the engine from the GUI
# Notice that this may not destroy the instance if the instance
# is referenced elsewhere (i.e. ``engine_wrapper``)
gui.remove_engine("test")
```

9.3.3 Simphony GUI within Mayavi2

A GUI essentially identical to the `EngineManagerStandaloneUI` is provided for the Mayavi2 application. In order to use it, one needs to first activate the plugin in `Preferences`, following the instructions in *Open CUDS Files in Mayavi2*. After that, **restart** Mayavi2. Then the EngineManager panel can be added by selecting `View` –> `Other...` –> `Simphony`.

The Simphony panel is binded to the embedded Python shell within Mayavi2 as `simphony_panel`. Alternatively the user can access the panel from `simphony.visualisation.mayavi_tools.get_simphony_panel`. With that the user can use the same methods as described in enginemanagerstandaloneui, such as `add_engine` and `remove_engine`.

Alternatively, the user can setup and load a SimPhoNy engine to Mayavi2 by running a python script from a shell or via Mayavi2 (File->Run Python Script).

The `add_engine_to_mayavi2` method in the *simphony.visualisation.mayavi_tools* namespace is provided for this purpose as illustrated in the following example.

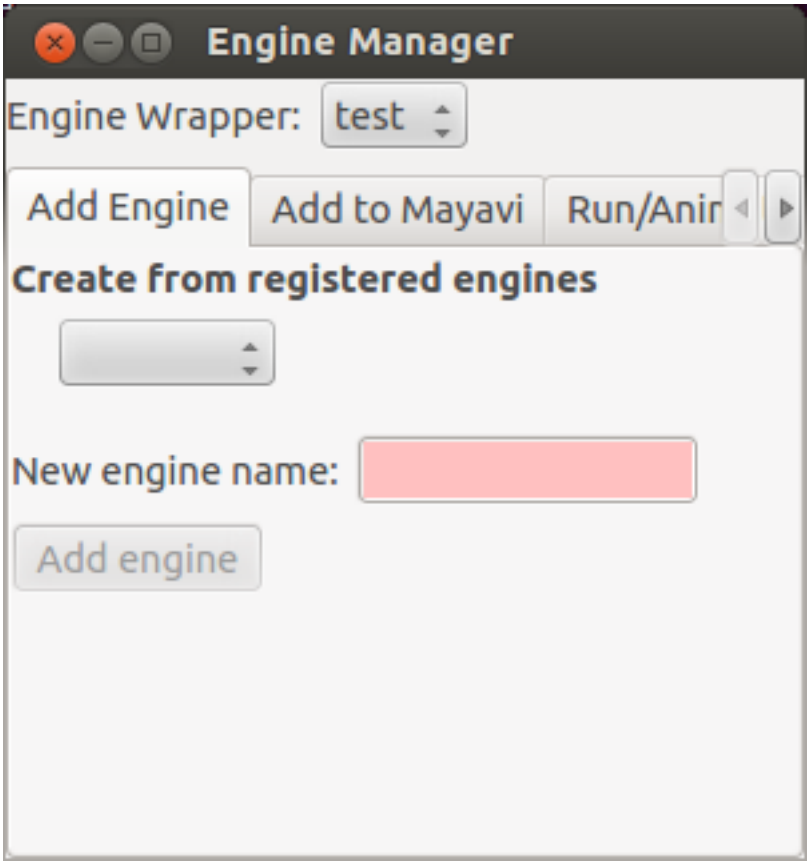


Fig. 9.8: Panel for adding more engine wrappers.

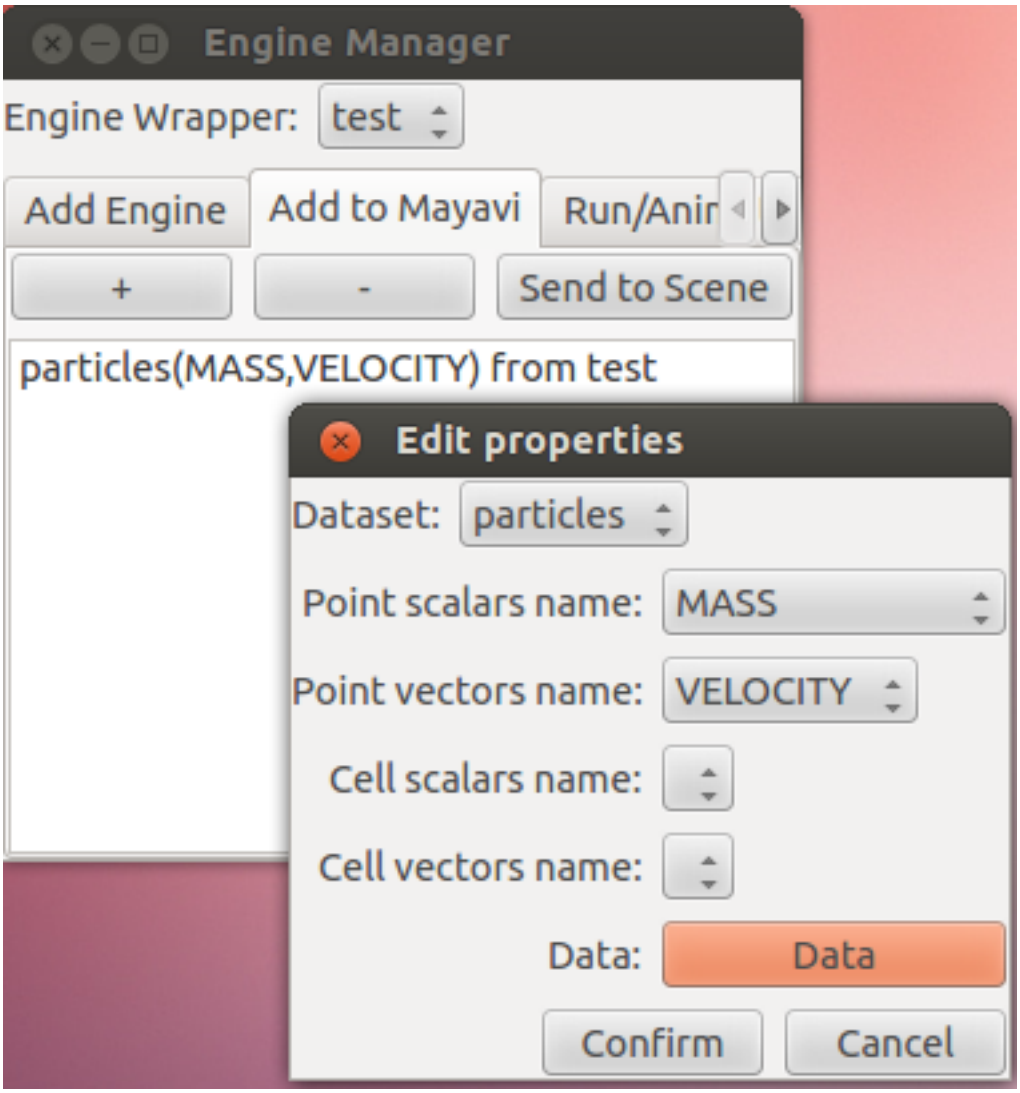


Fig. 9.9: Use *EngineManagerStandaloneUI* to add datasets to Mayavi.

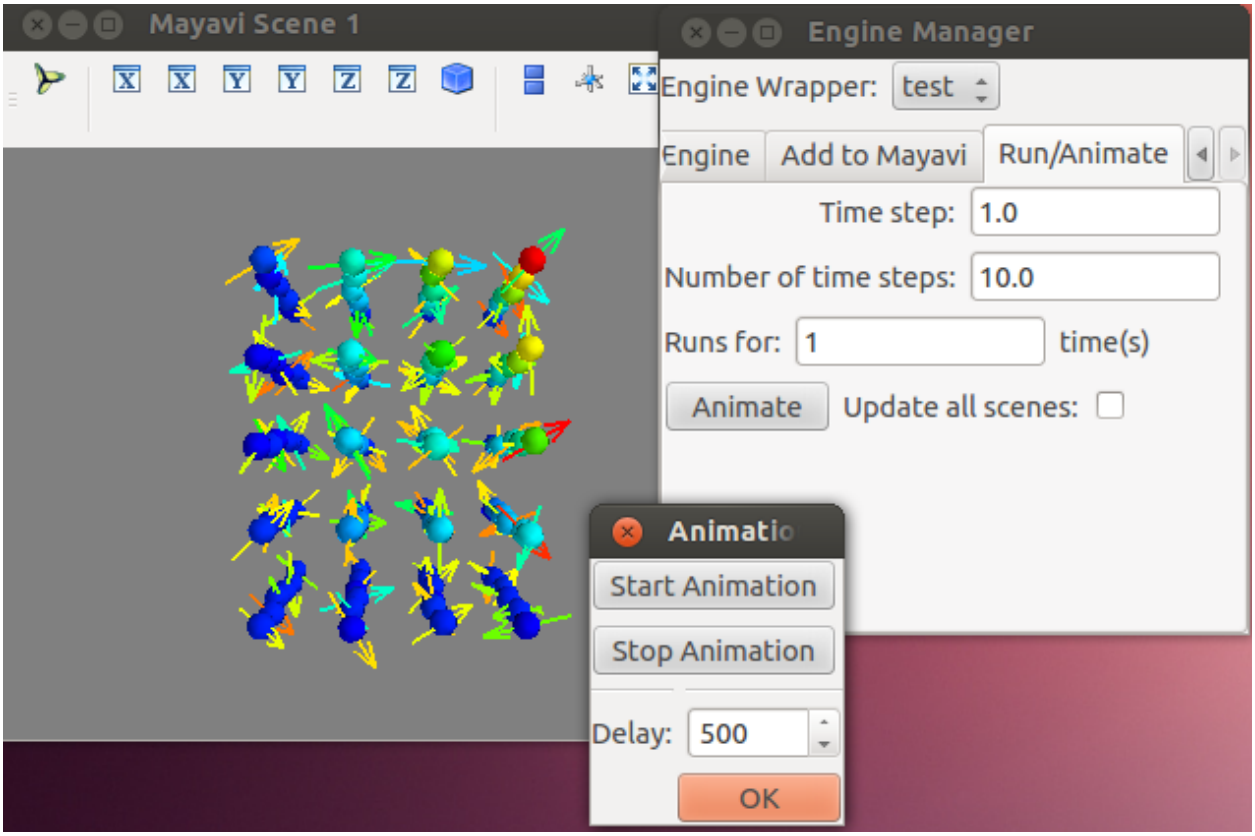


Fig. 9.10: Use *EngineManagerStandaloneUI* to run the engine and animate the results.

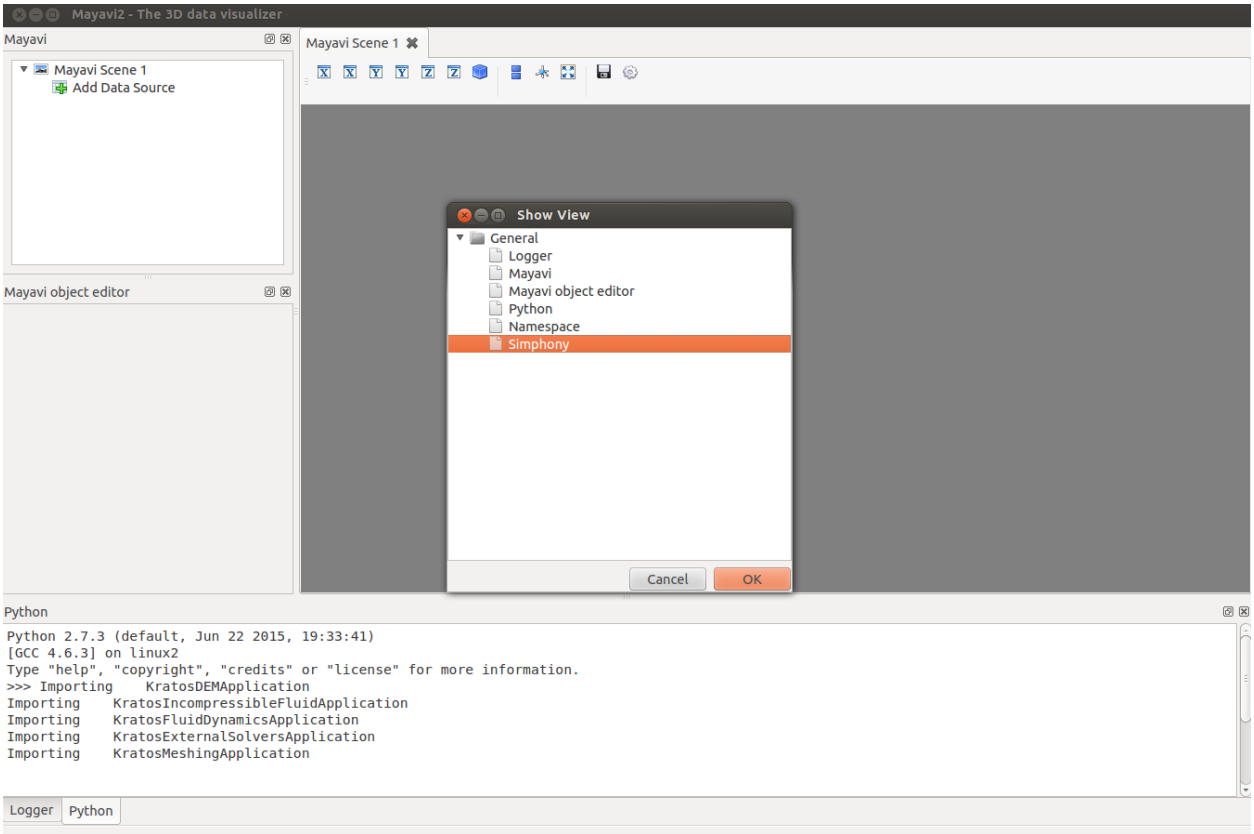


Fig. 9.11: Add the Simphony panel to Mayavi2

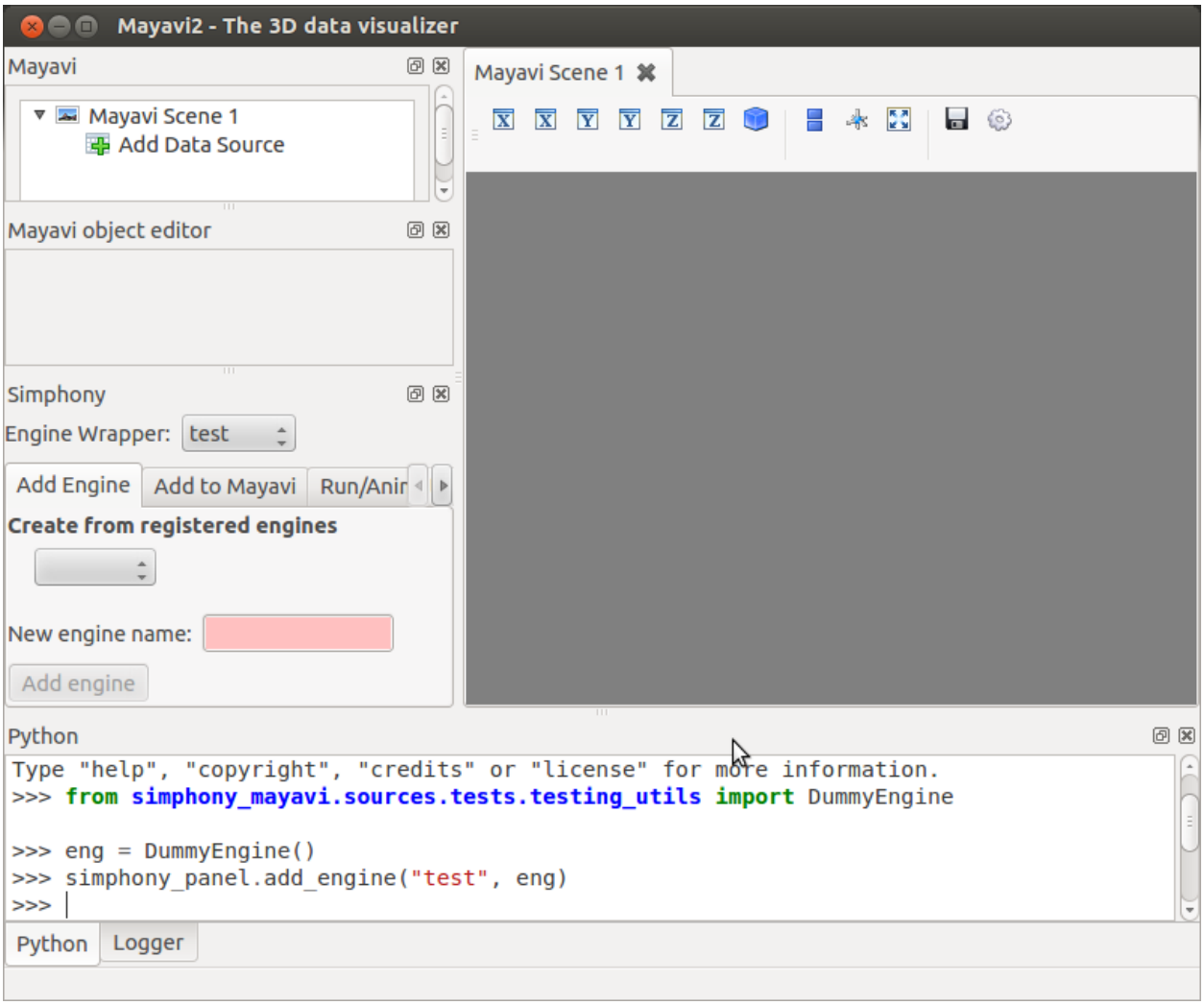


Fig. 9.12: The panel is identical to the *EngineManagerStandaloneUI*

```
""" Modified from simphony-lammps-md/examples/dem_billiards/dem_billiards.py
Requires file:
github.com/simphony/simphony-lammps-md/examples/dem_billiards/billiards_init.data
"""
import os

from mayavi.scripts import mayavi2

from simphony.engine import lammps
from simlammps import EngineType
from simphony.core.cuba import CUBA
from simphony.visualisation import mayavi_tools

# read data
particles = lammps.read_data_file(
    os.path.join(os.path.dirname(__file__),
                 "billiards_init.data"))[0]

# configure dem-wrapper
dem = lammps.LammpsWrapper(engine_type=EngineType.DEM)

dem.CM_extension[lammps.CUBAExtension.THERMODYNAMIC_ENSEMBLE] = "NVE"
dem.CM[CUBA.NUMBER_OF_TIME_STEPS] = 1000
dem.CM[CUBA.TIME_STEP] = 0.001

# Define the BC component of the SimPhoNy application model:
dem.BC_extension[lammps.CUBAExtension.BOX_FACES] = ["fixed",
                                                     "fixed",
                                                     "fixed"]
dem.BC_extension[lammps.CUBAExtension.BOX_VECTORS] = None

# add particles to engine
dem.add_dataset(particles)

# Run the engine
dem.run()


@mayavi2.standalone
def view():
    mayavi_tools.add_engine_to_mayavi2("lammps", dem)


if __name__ == "__main__":
    view()
```

This example sets up a Simphony LAMMPS engine and starts Mayavi2 with the engine loaded in the GUI.

CHAPTER 10

API Reference

10.1 Core module

A module containing core tools and wrappers for vtk data containers used in simphony_mayavi.

Classes

CubaData(attribute_data[, stored_cuba, ...])	Map a vtkCellData or vtkPointData object to a sequence of DataContainers.
CellCollection([cell_array])	A mutable sequence of cells wrapping a tvtk.CellArray.
mergedocs(other)	Merge the docstrings of other class to the decorated.
CUBADataAccumulator([keys])	Accumulate data information per CUBA key.
CUBADataExtractor(**traits)	Extract cuba data from cuds items iterable.

Functions

supported_cuba()	Return the list of CUBA keys that can be supported by vtk.
default_cuba_value(cuba)	Return the default value of the CUBA key as a scalar or numpy array.
cell_array_slicer(data)	Iterate over cell components on a vtk cell array
mergedoc(function, other)	Merge the docstring from the other function to the decorated function.

10.1.1 Description

class `simphony_mayavi.core.cuba_data.`**`CubaData`**(*attribute_data*, *stored_cuba=None*, *size=None*, *masks=None*)

Bases: `_abcoll.MutableSequence`

Map a vtkCellData or vtkPointData object to a sequence of DataContainers.

The class implements the `MutableSequence` api to wrap a `tvtk.CellData` or `tvtk.PointData` array where each CUBA key is a `tvtk.DataArray`. The aim is to help the conversion between column based structure of the `vtkCellData` or `vtkPointData` and the row based access provided by a list of `~.DataContainer`.

While the wrapped tvtk container is empty the following behaviour is active:

- Using `len` will return the `initial_size`, if defined, or 0.
- Using element access will return an empty *class:~.DataContainer*.

•No field arrays have been allocated.

When values are first added/updated with non-empty `DataContainers` then the necessary arrays are created and the `initial_size` info is not used anymore.

Note: Missing values for the attribute arrays are stored in separate attribute arrays named "<CUBA.name>-mask" as `0` while present values are designated with a `1`.

Constructor

attribute_data: tvtk.DataSetAttributes The vtk attribute container.

stored_cuba [set] The CUBA keys that are going to be stored default is the result of running `supported_cuba()`

size [int] The initial size of the container. Default is None. Setting a value will activate the virtual size behaviour of the container.

mask [tvtk.FieldData] A data arrays containing the mask of some of the CUBA data in `attribute_data`.

Raises

ValueError : When a non-empty `attribute_data` container is provided while size != None.

`cubas`

The set of currently stored CUBA keys.

For each cuba key there is an associated `DataArray` connected to the `PointData` or `CellData`

classmethod `empty`(*type_=<AttributeSetType.POINTS: 1>*, *size=0*)

Return an empty sequence based wrapping a vtkAttributeDataSet.

Parameters

- **`size`** (*`int`*) – The virtual size of the container.
- **`type_`** (*`AttributeSetType`*) – The type of the vtkAttributeSet to create.

`insert`(*index*, *value*)

Insert the values of the DataContainer in the arrays at row=''index''.

If the provided DataContainer contains new, but supported, cuba keys then a new empty array is created for them and updated with the associated values of `value`. Unsupported CUBA keys are ignored.

Note: The underline data structure is better suited for append operations. Inserting values in the middle or at the front will be less efficient.

---

class `simphony_mayavi.core.cell_collection.`**`CellCollection`**(*cell_array=None*)

Bases: `_abcoll.MutableSequence`

A mutable sequence of cells wrapping a tvtk.CellArray.

Constructor

Parameters **`cell_array`** (*`tvtk.CellArray`*) – The tvtk object to wrap. Default value is an empty tvtk.CellArray.

__delitem__(*index*)
Remove cell at `index`.

> **Note:** This operation will need to create temporary arrays in order to keep the data info consistent.

__getitem__(*index*)
Return the connectivity list for the cell at `index`.

__len__()
The number of contained cells.

__setitem__(*index*, *value*)
Update the connectivity list for cell at `index`.

> **Note:** If the size of the connectivity list changes a slower path creating temporary arrays is used.

insert(*index*, *value*)
Insert cell at `index`.

> **Note:** This operation needs to use a slower path based on temporary array when index < sequence length.

class `simphony_mayavi.core.cuba_data_accumulator.`**CUBADataAccumulator**(*keys=None*)
Bases: `object`

Accumulate data information per CUBA key.

A collector object that stores :class:`DataContainer` data into a list of values per CUBA key. By appending `DataContainer` instanced the user can effectively convert the per item mapping of data values in a CUDS container to a per CUBA key mapping of the data values (useful for coping data to vtk array containers).

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 and missing values will be saved as `None`. Where `expand` will extend the internal table of values whenever a new key is introduced.

expand operation

```
>>> accumulator = CUBADataAccumulator():
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.keys()
{CUBA.TEMPERATURE}
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.VELOCITY}
>>> accumulator[CUBA.TEMPERATURE]
[34, None, 56]
>>> accumulator[CUBA.VELOCITY]
[None, (0.1, 0.1, 0.1), None]
```

fixed operation

```
>>> accumulator = CUBADataAccumulator([CUBA.TEMPERATURE, CUBA.PRESSURE]):
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.PRESSURE}
>>> accumulator.append(DataContainer(TEMPERATURE=34))
>>> accumulator.append(DataContainer(VELOCITY=(0.1, 0.1, 0.1))
>>> accumulator.append(DataContainer(TEMPERATURE=56))
>>> accumulator.keys()
{CUBA.TEMPERATURE, CUBA.PRESSURE}
>>> accumulator[CUBA.TEMPERATURE]
[34, None, 56]
>>> accumulator[CUBA.PRESSURE]
[None, None, None]
>>> 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. An empty list is acceptable, and returns a trivial accumulator providing no data. If None is passed, then the 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** (*list*) – A list of data values collected for `key`. Missing values are designated with `None`.

__len__()

The number of values that are stored per key

Note: Behaviour is temporary and will probably change soon.

---

append(*data*)

Append info from a `DataContainer`.

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

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 lits).
- Missing keys will be stored as `None`

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 values of `None`.
- 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 store as `None`.

keys

The set of CUBA keys that this accumulator contains.

load_onto_vtk(*vtk_data*)
Load the stored information onto a vtk data container.

Parameters **vtk_data** (*vtkPointData or vtkCellData*) – The vtk container to load the value onto.

Data are loaded onto the vtk container based on their data type. The name of the added array is the name of the CUBA key (i.e. *CUBA*.name). Currently only scalars and three dimensional vectors are supported.

class simphony_mayavi.core.cuba_data_extractor.**CUBADataExtractor**(**traits)
Bases: traits.has_traits.HasStrictTraits

Extract cuba data from cuds items iterable.

The class that supports extracting data values of a specific CUBA key from an iterable that returns low level CUDS objects (e.g. Point).

available = Property(Set(CUBATrait), depends_on='_available')
The list of cuba keys that are available (read only). The value is recalculated at initialialisation and when the reset method is called.

data = Property(Dict(UUID, Any), depends_on='_data')
The dictionary mapping of item uid to the extracted data value. A change Event is fired for data when selected or keys change or the reset method is called.

function = ReadOnly
The function to call that returns a generator over the desired items (e.g. Mesh.iter_points). This value cannot be changed after initialisation.

keys = Either(None, Set(UUID))
The list of uuid keys to restrict the data extraction. This attribute is passed to the function generator method to restrict iteration over the provided keys (e.g Mesh.iter_points(uids=keys))

reset()
Reset the available and data attributes.

selected = CUBATrait
Currently selected CUBA key. Changing the selected key will fire events that will result in executing the generator function and extracting the related values from the CUDS items that the iterator yields. The resulting mapping of uid -> value will be stored in data.

class simphony_mayavi.core.doc_utils.**mergedocs**(*other*)
Bases: object

Merge the docstrings of other class to the decorated.

simphony_mayavi.core.cuba_utils.**supported_cuba**()
Return the list of CUBA keys that can be supported by vtk.

simphony_mayavi.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.

Note: Only vector and scalar values are currently supported.

simphony_mayavi.core.cell_array_tools.**cell_array_slicer**(*data*)
Iterate over cell components on a vtk cell array

VTK stores the associated point index for each cell in a one dimensional array based on the following template:

```
[n, id0, id1, id2, ..., idn, m, id0, ...]
```

The iterator takes a cell array and returns the point indices for each cell.

`simphony_mayavi.core.doc_utils.`**`mergedoc`**(*function*, *other*)
Merge the docstring from the other function to the decorated function.

10.2 Cuds module

A module containing tvtk dataset wrappers to simphony CUDS containers.

Classes

`VTKParticles`(name[, data, data_set, mappings])	Constructor.
`VTKMesh`(name[, data, data_set, mappings])	Constructor.
`VTKLattice`(name, primitive_cell, data_set[, ...])	Constructor.

10.2.1 Description

class `simphony_mayavi.cuds.vtk_particles.`**`VTKParticles`**(*name*, *data=None*, *data_set=None*, *mappings=None*)

Bases: `simphony.cuds.abc_particles.ABCParticles`

Constructor.

Parameters

- **`name`** (*`string`*) – The name of the container.
- **`data`** (*`DataContainer`*) – The data attribute to attach to the container. Default is None.
- **`data_set`** (*`tvtk.DataSet`*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.PolyData
- **`mappings`** (*`dict`*) – A dictionary of mappings for the particle2index, index2particle, bond2index and bond2element. Should be provided if the particles and bonds described in `data_set` are already assigned uids. Default is None and will result in the uid <-> index mappings being generated at construction.

`bond2index` = None
The mapping from uid to bond index

`count_of`(*item_type*)
Return the count of item_type in the container.

Parameters **`item_type`** (*`CUDSItem`*) – The CUDSItem enum of the type of the items to return the count of.

Returns **count** (*int*) – The number of items of item_type in the dataset.

Raises **ValueError** – If the type of the item is not supported in the current dataset.

`data`
Easy access to the vtk CellData structure

data_set = None
The vtk.PolyData dataset

classmethod from_dataset(*name*, *data_set*, *data=None*)
Wrap a plain dataset into a new VTKParticles.

The constructor makes some sanity checks to make sure that the tvtk.DataSet is compatible and all the information can be properly used.

Parameters

- **name** (*str*) – The name of the container.
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.PolyData
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Raises **TypeError** – When the sanity checks fail.

classmethod from_particles(*particles*, *particle_keys=None*, *bond_keys=None*)
Create a new VTKParticles copy from a CUDS particles instance.

Parameters

- **particles** (*ABCParticles*) – CUDS Particles dataset
- **particle_keys** (*list*) – A list of point CUBA keys that we want to copy, and only those. If None, all available and compatible keys will be copied.
- **bond_keys** (*list*) – A list of cell CUBA keys that we want to copy, and only those. If None, all available and compatible keys will be copied.

index2bond = None
The reverse mapping from index to bond uid

index2particle = None
The reverse mapping from index to point uid

is_connected(*bond*)
Test if the connectivity described in bonds is valid i.e. the particles are part of the container

Parameters **bond** (*Bond*) –

Returns **valid** (*bool*)

particle2index = None
The mapping from uid to point index

supported_cuba = None
The currently supported and stored CUBA keywords.

class simphony_mayavi.cuds.vtk_mesh.**VTKMesh**(*name*, *data=None*, *data_set=None*, *mappings=None*)
Bases: simphony.cuds.abc_mesh.ABCMesh

Constructor.

Parameters

- **name** (*string*) – The name of the container
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.UnstructuredGrid.

- **mappings** (*dict*) – A dictionary of mappings for the point2index, index2point, element2index and index2element. Should be provided if the points and elements described in `data_set` are already assigned uids. Default is None and will result in the uid <-> index mappings being generated at construction.

count_of(*item_type*)

Return the count of item_type in the container.

Parameters **item_type** (*CUDSItem*) – The CUDSItem enum of the type of the items to return the count of.

Returns **count** (*int*) – The number of items of item_type in the dataset.

Raises **ValueError** – If the type of the item is not supported in the current dataset.

data

Easy access to the vtk PointData structure

data_set = None

The vtk.PolyData dataset

element2index = None

The mapping from uid to bond index

element_data = None

Easy access to the vtk CellData structure

classmethod from_dataset(*name*, *data_set*, *data=None*)

Wrap a plain dataset into a new VTKMesh.

The constructor makes some sanity checks to make sure that the tvtk.DataSet is compatible and all the information can be properly used.

Parameters

- **name** (*string*) – The name of the container
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. Default is None which will create a tvtk.UnstructuredGrid.
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Raises **TypeError** – When the sanity checks fail.

classmethod from_mesh(*mesh*, *point_keys=None*, *cell_keys=None*)

Create a new VTKMesh copy from a CUDS mesh instance.

Parameters

- **mesh** (*ABCMesh*) – The original mesh to create the new one.
- **point_keys** (*list*) – A list of point CUBA keys that we want to copy, and only those. If None, all available and compatible keys will be copied.
- **cell_keys** (*list*) – A list of cell CUBA keys that we want to copy, and only those. If None, all available and compatible keys will be copied.

index2element = None

The reverse mapping from index to bond uid

index2point = None

The reverse mapping from index to point uid

point2index = None

The mapping from uid to point index

supported_cuba = None
The currently supported and stored CUBA keywords.

class simphony_mayavi.cuds.vtk_lattice.**VTKLattice**(*name*, *primitive_cell*, *data_set*, *data=None*)

Bases: simphony.cuds.abc_lattice.ABCLattice

Constructor.

Parameters

- **name** (*string*) – The name of the container.
- **primitive_cell** (*PrimitiveCell*) – primitive cell specifying the 3D Bravais lattice
- **data_set** (*tvtk.DataSet*) – The dataset to wrap in the CUDS api. If it is a tvtk.PolyData, the points are assumed to be arranged in C-contiguous order so that the first point is the origin and the last point is furthest away from the origin
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

count_of(*item_type*)
Return the count of item_type in the container.

Parameters **item_type** (*CUDSItem*) – The CUDSItem enum of the type of the items to return the count of.

Returns **count** (*int*) – The number of items of item_type in the dataset.

Raises **ValueError** – If the type of the item is not supported in the current dataset.

data
The container data

classmethod empty(*name*, *primitive_cell*, *size*, *origin*, *data=None*)
Create a new empty Lattice.

Parameters

- **name** (*string*) – The name of the container.
- **primitive_cell** (*PrimitiveCell*) – Primitive cell specifying the 3D Bravais lattice
- **size** (*tuple*) – lattice dimensions (nx, ny, nz)
- **origin** (*tuple*) – lattice origin (x, y, z)
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Returns **lattice** (*VTKLattice*)

classmethod from_dataset(*name*, *data_set*, *data=None*)
Create a new Lattice and try to guess the primitive_cell

Parameters

- **name** (*str*) –
- **data_set** (*tvtk.ImageData or tvtk.PolyData*) – The dataset to wrap in the CUDS api. If it is a PolyData, the points are assumed to be arranged in C-contiguous order
- **data** (*DataContainer*) – The data attribute to attach to the container. Default is None.

Returns **lattice** (*VTKLattice*)

Raises **TypeError** – If data_set is not either tvtk.ImageData or tvtk.PolyData

IndexError: If the lattice nodes are not arranged in C-contiguous order

classmethod **from_lattice**(*lattice*, *node_keys=None*)

Create a new Lattice from the provided one.

Parameters

- **lattice** (*simphony.cuds.lattice.Lattice*) –
- **node_keys** (*list*) – A list of point CUBA keys that we want to copy, and only those. If None, all available and compatible keys will be copied.

Returns **lattice** (*VTKLattice*)

Raises

ValueError

- if bravais_lattice attribute of the primitive cell indicates a cubic/tetragonal/orthorhombic lattice but the primitive vectors are inconsistent with this attribute
- if bravais_lattice is not a member of BravaisLattice

get_coordinate(*ind*)

Get coordinate of the given index coordinate.

ind [int[3]] node index coordinate

Returns

coordinates : float[3]

origin

lattice origin (x, y, z)

point_data = None

Easy access to the vtk PointData structure

primitive_cell

Primitive cell specifying the 3D Bravais lattice

size

lattice dimensions (nx, ny, nz)

supported_cuba = None

The currently supported and stored CUBA keywords.

10.3 Modules module

simphony_mayavi.modules.default_module.**default_module**(*source*)

Mapping for module appropriate for the selected data

Parameters **source** (*CUDSSource*) –

Returns **modules** (*list*) – mayavi modules to be added to the pipeline

simphony_mayavi.modules.default_module.**default_scalar_module**(*scale_factor=1.0*)

Returns a Glyph with a sphere glyph source and scale_mode turned off. Suitable for points/nodes with scalar data

`simphony_mayavi.modules.default_module.`**`default_vector_module`**(*scale_factor=1.0*)

Returns a Glyph in its original mayavi defaults plus the scale_mode turned off

10.4 Plugin module

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

Classes

`EngineManagerStandaloneUI`

Functions

`simphony_mayavi.plugin.show`
`simphony_mayavi.plugin.snapshot`
`simphony_mayavi.plugin.adapt2cuds`
`simphony_mayavi.plugin.load`
`simphony_mayavi.plugin.add_engine_to_mayavi2`
`simphony_mayavi.plugin.get_simphony_panel`
`simphony_mayavi.plugin.restore_scene`

10.4.1 Description

10.5 Plugins module

This module contains classes the Simphony plugins for the Mayavi2 application.

Classes

`simphony_mayavi.plugins.add_source_panel.AddSourcePanel`	
`simphony_mayavi.plugins.add_engine_panel.AddEnginePanel`	
`simphony_mayavi.plugins.engine_manager.EngineManager`	
`simphony_mayavi.plugins.engine_manager_mayavi2.EngineManagerMayavi2`	
`simphony_mayavi.plugins.engine_manager_standalone_ui.EngineManagerStandaloneUI`	
`RunAndAnimate`	Standalone no
`simphony_mayavi.plugins.run_and_animate_panel.RunAndAnimatePanel`	
`simphony_mayavi.plugins.tabbed_panel_collection.TabbedPanelCollection`	

Functions

`simphony_mayavi.plugins.add_source_panel.add_source_and_modules_to_scene`

10.5.1 Descriptions

class simphony_mayavi.plugins.run_and_animate.**RunAndAnimate**(*engine*, *mayavi_engine*)
Bases: object

Standalone non-GUI based controller for running a Simphony Modeling Engine and animating the CUDS dataset in Mayavi.

Precondition: The required CUDS datasets are already visible in the Mayavi scene(s)

Parameters

- **engine** (*ABCModelingEngine*) – Simphony Modeling Engine
- **mayavi_engine** (*mayavi.api.Engine*) – for retrieving scenes and visible datasets

animate(*number_of_runs*, *delay=None*, *ui=False*, *update_all_scenes=False*)
Run the modeling engine, and animate the scene. If there is no source in the scene or none of the sources belongs to the selected Engine engine, a RuntimeError is raised.

Parameters

- **number_of_runs** (*int*) – the number of times the engine.run() is called
- **delay** (*int*) – delay between each run. If None, use previous setting or the Mayavi's default (500)
- **ui** (*bool*) – whether an UI is shown, default is False
- **update_all_scenes** (*bool*) – whether all scenes are updated, default is False: i.e. only the current scene is updated

Raises

RuntimeError if nothing in scene(s) belongs to engine

10.5.2 Engine_wrapper module

class simphony_mayavi.plugins.engine_wrappers.abc_engine_factory.**ABCEngineFactory**
Bases: traits.has_traits.ABCHasStrictTraits

create()
Return a new engine wrapper instance

class simphony_mayavi.plugins.engine_wrappers.jyulb.**JyulbFileIOEngineFactory**
Bases: *simphony_mayavi.plugins.engine_wrappers.abc_engine_factory.ABCEngineFactory*

class simphony_mayavi.plugins.engine_wrappers.jyulb.**JyulbInternalEngineFactory**
Bases: *simphony_mayavi.plugins.engine_wrappers.abc_engine_factory.ABCEngineFactory*

class simphony_mayavi.plugins.engine_wrappers.openfoam.**OpenFoamFileIOEngineFactory**
Bases: *simphony_mayavi.plugins.engine_wrappers.abc_engine_factory.ABCEngineFactory*

class simphony_mayavi.plugins.engine_wrappers.openfoam.**OpenFoamInternalEngineFactory**
Bases: *simphony_mayavi.plugins.engine_wrappers.abc_engine_factory.ABCEngineFactory*

10.6 Sources module

A module containing objects that wrap CUDS objects and files to Mayavi compatible sources. Please use the simphony_mayavi.sources.api module to access the provided tools.

Classes

cuds_source.CUDSSource
cuds_file_source.CUDSFileSource
engine_source.EngineSource

10.6.1 Description

S

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

O

P

R

S

V