SimPhoNy Documentation

Release 0.2.1

SimPhoNy, EU FP7 Project (Nr. 604005)

December 09, 2015

Contents

CHAPTER 1

Simphony-common

The native implementation of the SimPhoNy cuds objects and io code (http://www.simphony-project.eu/).

1.1 Repository

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

1.2 Requirements

- enum34 >= 1.0.4
- stevedore >= 1.2.0
- numpy >= 1.4.1

1.2.1 Optional requirements

To support the cuba-generate script the following packages need to be installed prior to installing Simphony:

- click >= 3.3
- pyyaml >= 3.11
- tabulate >= 0.7.4

To support the HDF5 based native IO:

- PyTables >= 3.1.1

To support the documentation built you need the following packages:

- sphinx >= 1.2.3
- sphinxcontrib-napoleon >= 0.2.10
- mock

Note: Packages that depend on the optional features and use setuptools should append the `H5IO` and/or `CUBAGen` identifier next to simphony in their `setup_requires` configuration option. For example:

```
install_requires = ["simphony[H5IO, CUBAGen]"]
```

Will make sure that the requirements of H5IO and CUBAGen support are installed. (see setuptools extras for more information)

1.3 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
```

1.4 Testing

To run the full test-suite run:

```
python -m unittest discover -p test*
```

1.5 Documentation

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

```
python setup.py build_sphinx
```

If you recreate the uml diagrams you need to have java and xdot installed:

```
sudo apt-get install default-jre xdot
```

A copy of the plantuml.jar needs also to be available in the `doc/` folder. Running `make uml` inside the `doc/` directory will recreate all the UML diagrams.

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.
- Not all the png files of the UML diagrams are used.

1.6 Directory structure

There are four subpackages:

- core – used for common low level classes and utility code
- cuds – to hold all the native cuds implementations
- io – to hold the io specific code

- bench – holds basic benchmarking code
- examples – holds SimPhoNy example code
- 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.

1.7 SimPhoNy Framework

The `simphony` library is the core component of the SimPhoNy Framework; information on setting up the framework is provided on a separate repository https://github.com/simphony/simphony-framework.

CHAPTER 2

User Manual

2.1 Plugins

The SimPhoNy library can extended through two entry points for contributing python modules that contain engine and visualisation components:

- `simphony.engine` – A python module that provides one or more classes that implement the `ABCModelingEngine` interface.
- `simphony.visualisation` – A python module that provides a simple function to show (visualise the high level CUDS containers)

To declare that a package contains a visualisation or engine module for simphony, a developer has to add an entry point definition in the `setup.py` of the contributing package.

e.g.:

```
setup(
   entry_points={
    'simphony.engine': ['<name> = <module_path>'])
```

Where `<module_path>` is a module where the engine class(es) can be found like `my_cool_engine_plugin.cool_engine342` and `<name>` is the user visible name that the `cool_engine432` module will have inside the SimPhoNy framework. It is important that <name> is unique and specific to the contributed components (e.g. name == 'default' is probably a very bad choice)

e.g.:

```
setup(
   entry_points={
    'simphony.engine': ['cool = my_cool_engine_plugin.cool_engine342'])
```

Will allow the user to import the new engine from inside the `simphony` module as follows

```
from simphony.engine import cool
# cool is now a reference to the external module ``my_cool_engine_plugin.cool_engine342``
# If the name of the provided engine class is EngFast then the user should be able to do
engine = cool.EngFast()
```

Note: The `examples/plugin` folder of the simphony-common repository contains a dummy package that contributes python modules to both

2.2 CUBA-Keywords

Common Unified Base Attributes (CUBA) are a list of common keywords transcending across different scales, methods and modelling-engines. They provide a standard nomenclature for attributes (variables and parameters) in SimPhoNy.

More detailed information for each CUBA-keyword is provided in the following table.

Name	Description	Domain
Id	Universal unique id represented as a hex string size 32	['ATM', 'DEM', 'FEM', 'FV
Name	Naming of high-level objects (e.g. solver models)	['ATM', 'DEM', 'FEM', 'FV
Position	Position of a point or node or atom	['ATM', 'DEM', 'FEM', 'FV
Direction	Geometric (more general than, e.g., velocity) could be used for spin	['ATM', 'FEM', 'FVM', 'LB
Status	Status of a point or node	['DEM', 'LBM']
Label	Label for a point or node	['ATM', 'DEM', 'FEM', 'FV
MaterialId	Material identification number	['DEM', 'FEM', 'FVM', 'LE
ChemicalSpecie	Chemical Specie	['ATM', 'VIS']
MaterialType	Material dimension and type	['VIS']
ShapeCenter	Geometrical center of the shape of the material	['VIS']
ShapeLengthUC	Length in units cells of the shape of the material	['VIS']
ShapeLength	Length in angstroms of the shape of the materials	['VIS']
ShapeRadius	Radius for a spherical material	['VIS']
ShapeSide	Side length for a hexagonal material	['VIS']
CrystalStorage	Additional information for visualization	['VIS']
NameUC	Name of the unit cell of the component	['ATM', 'VIS']
LatticeVectors	Lattice vectors of unit cell of the component	['ATM', 'VIS']
SymmetryLatticeVectors	Symmetry Group	['ATM', 'VIS']
Occupancy	Occupancy of an atomic position	['ATM', 'VIS']
BondLabel	Unique ID of atoms	['ATM', 'VIS']
BondType	Type of label	['ATM', 'VIS']
Velocity	Velocity of a point or node	['ATM', 'DEM', 'FEM', 'FV
Acceleration	Acceleration of a point or node	['ATM', 'DEM', 'LBM', 'SP
NumberOfPoints	Number of points or nodes	['DEM', 'FEM', 'FVM', 'LE
Radius	Particle radius	['DEM', 'SPH']
Size	For non-spherical particles	['DEM', 'SPH']
Mass	Particle mass	['ATM', 'DEM']
Volume	Volume of a particle, cell, etc.	['DEM', 'FEM', 'FVM', 'LE
AngularVelocity	Angular velocity of a point or node	['DEM']
AngularAcceleration	Angular acceleration of a point or node	['DEM']
SimulationDomainDimensions	Size of the simulation domain	['DEM', 'FEM', 'FVM', 'LE
SimulationDomainOrigin	Offset for the simulation domain	['DEM', 'FEM', 'FVM', 'LE
DynamicViscosity	Dynamic viscosity of fluid	['DEM', 'FEM', 'FVM', 'LE
KinematicViscosity	Kinematic viscosity of fluid	['FEM', 'FVM', 'LBM']
DiffusionCoefficient	Diffusion coefficient	['FEM', 'FVM', 'LBM']
ProbabilityCoefficient	For stochastic processes (e.g. sorption)	['DEM', 'LBM']
FrictionCoefficient	Control particle friction	['DEM', 'LBM']
ScalingCoefficient	Coarsening or time-scale bridging	['DEM', 'LBM']
EquationOfStateCoefficient	Equation of state for multiphase fluids	['FEM', 'FVM', 'LBM', 'SP
ContanctAngle	Wettability in multiphase flows	['LBM']
Amphiphilicity	Hydrophilic/-phile behaviour of a particle	['DEM']
PhaseInteractionStrength	Strength of phase interactions on a particle	['DEM']

Table 2.1 – continued from previ

Name	Description	Domain
HamakerConstant	Van der Waals body-body interaction	['DEM']
ZetaPotential	Coulomb interaction between particles	['DEM']
IonValenceEffect	Coulomb interaction between particles	['DEM']
DebyeLength	Electrostatic effects of particles in solution	['DEM']
SmoothingLength	Half of kernel cut-off for all splines	['SPH']
LatticeSpacing	Distance between adjacent lattice nodes	['LBM']
TimeStep	Length of a discrete time step	['DEM', 'FEM', 'FVM', 'LE
NumberOfTimeSteps	Number of discrete time steps	['DEM', 'FEM', 'FVM', 'LE
Force	Force	['DEM', 'LBM', 'SPH']
Torque	Torque	['DEM']
Density	Density	['DEM', 'FEM', 'FVM', 'LE
Concentration	Concentration of a substance	['ATM', 'FEM', 'FVM', 'LB
Pressure	Pressure	['FEM', 'FVM', 'LBM', 'SP
Temperature	Temperature	['DEM', 'FEM', 'FVM', 'LE
Distribution	Single-particle distribution function	['ATM', 'LBM', 'VIS']
OrderParameter	Phase field in multiphase flows	['LBM']
OriginalPosition	Position at the beginning of the calculation	['DEM']
DeltaDisplacement	Displacement during the last time step	['DEM']
ExternalAppliedForce	Externally applied force (force fields, interactions, etc)	['DEM']
EulerAngles	Euler Angles	['DEM']
Sphericity	Sphericity of the particle	['DEM']
YoungModulus	Young Modulus	['DEM']
PoissonRatio	Poisson Ratio	['DEM']
LnOfRestitutionCoefficient	Natural Logarithm of the Restitution Coefficient	['DEM']
RollingFriction	Rolling Friction coefficient	['DEM']
VolumeFraction	Volume fraction	['FEM', 'FVM']

2.3 CUDS Data Structures

The Common Unified Data Structures (CUDS) define the expected interface for the modelling engine wrappers, the visualisation api the top level CUDS containers (i.e. Mesh, Lattice and Particles) and the low level CUDS components (e.g Point, Bond, LatticeNode)

Note: The CUDS API is constantly evolving and changes through the processes of the SSB and the designated workgroups.

2.3.1 Modelling engine

The CUDS engine supports operations to add, get and remove CUDS containers from the internal memory of the engine wrapper to setup the SD state of the simulations.

When a CUDS container is added the CUDS modelling engine will copy into internal memory only the `uid` mapping and the CUBA information that it can support. The user needs to refer to the wrapper package to identify what information is retained.

When a CUDS container is returned as a result of a `get` operation in a modelling engine the returned Container is a proxy to the information stored in the internal memory if the wrapper/modelling engine.

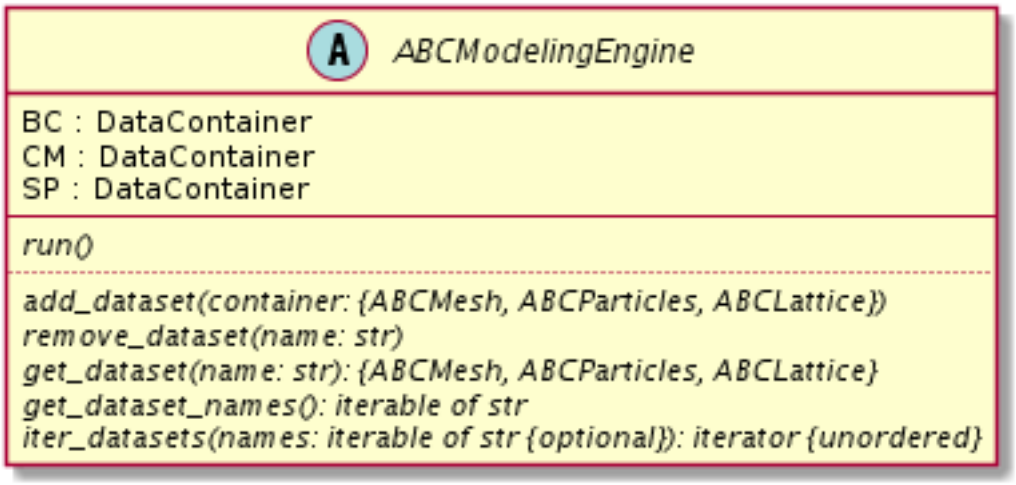

Fig. 2.1: **Figure. 1:** UML description of a CUDS modelling engine.

2.3.2 CUDS containers

The CUDS containers, with the exception of the Lattice, are multi-type container (i.e. contain different types of objects). For each type the following operations are currently supported:

- Add an item to the container
- Get an item from the container based on the `uid`
- Remove an item from the container based on the `uid`
- Update an item
- Iterate over all or some of the items given an iterable of `uids`

Note:

- Lattice items do not have a `uid` but are accessed based on `(i,j,k)` index tuples.
- For Mesh and Lattice containers the description does not define remove operations.
- The Lattice container does not have an add item operation.

Snapshot principle

All CUDS containers (native or proxy based) are owners of their data and will always return components that contain a copy (snapshot) of the internal data representation. As a result, the information of an item extracted with a `get` operation before and after a simulation run is not expected to be the same. The snapshot principle also means that the returned CUDS items do not depend anymore on the container instance they where extracted from.

2.3.3 CUDS Items

Low level items are smallest objects that can hold CUBA information in SimPhoNy simulations. Each CUDS container supports a specific set of these types as presented in **Fig. 2**.

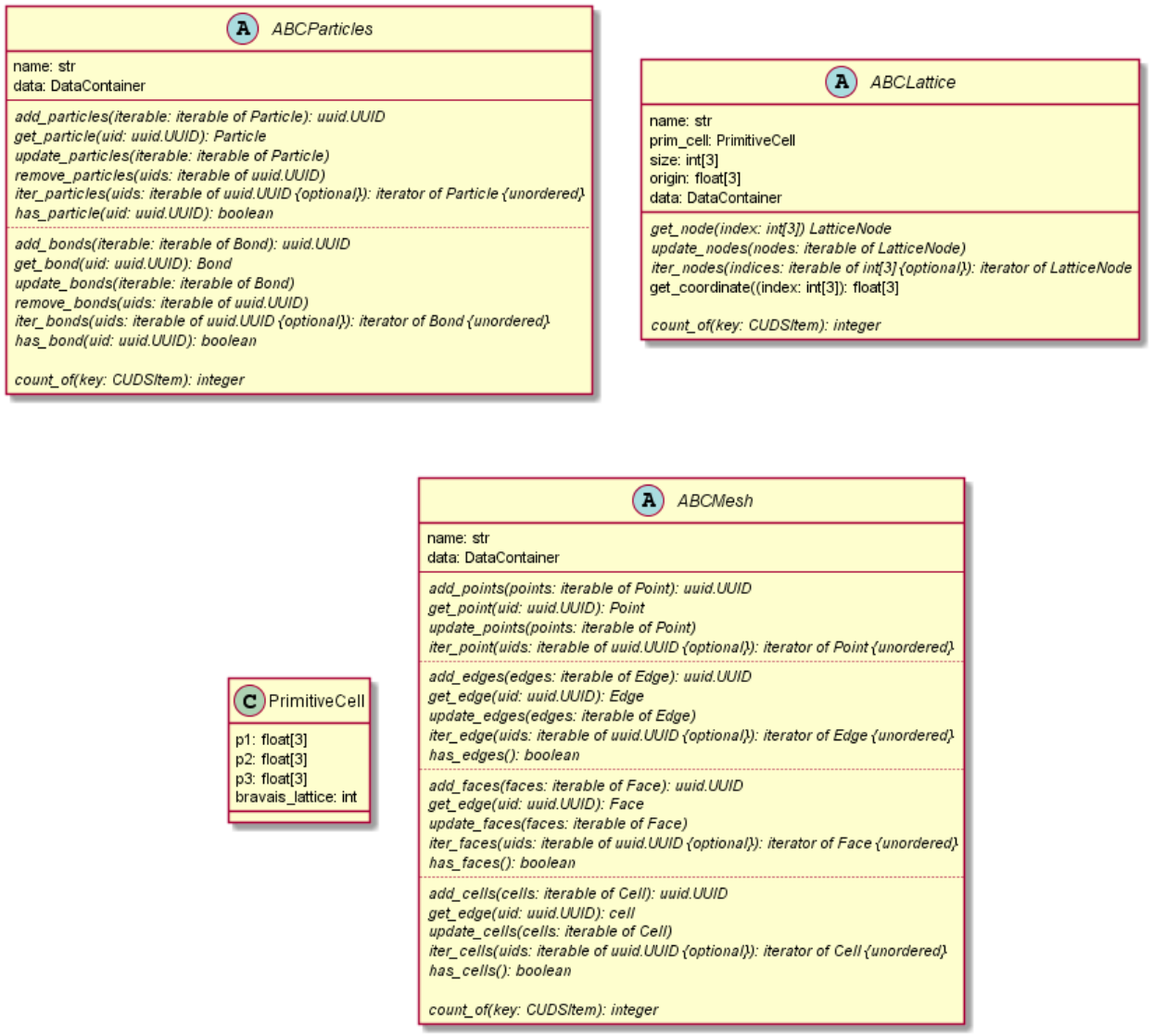

Fig. 2.2: **Figure. 2:** UML description of a CUDS Containers

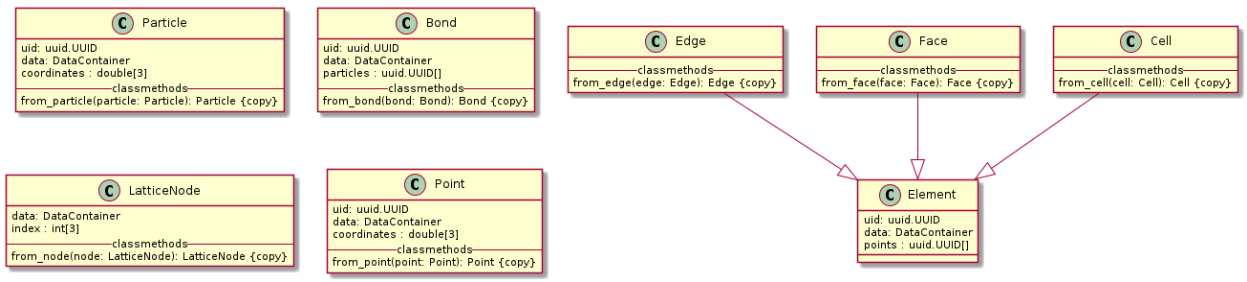

Fig. 2.3: **Figure 3:** UML diagram of the CUDS items and their relations.

2.3.4 Core items

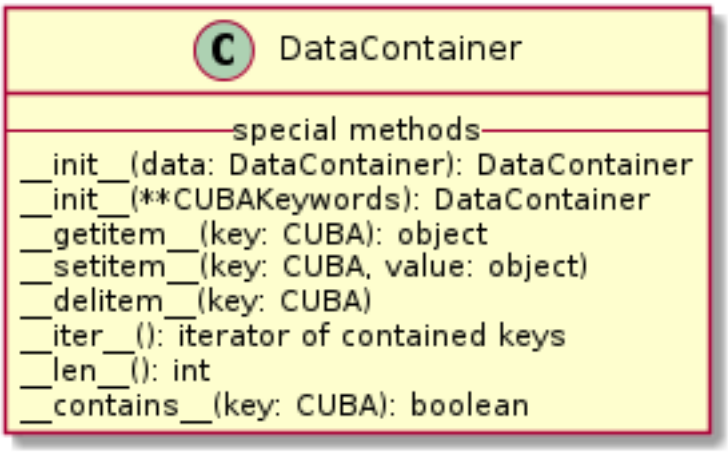

Fig. 2.4: **Figure 4:** UML diagram of the DataContainer.

The CUBA attribute container part of the SimPhoNy core. A dictionary like object maps `CUBA` enum keys to they values. In its native python implementation it can support all CUBA attributes.

2.4 HDF5 Storage

Cuds containers can be stored in HDF5 files using the *H5CUDS* class. The provided api is currently a reduced version of the Modelling Engine api and supports adding and manipulating CUDS containers. Please also note that returned containers from the get methods are live proxy objects on top of the HDF5 storage (in contrast to the common offline save and read operations).

2.4.1 HDF5 Stored Layout

Data are stored in HDF5 files using a separate layout for each type of CUDS container. The stored layout of the containers is provided below using a pseudo-uml description for the HDF5 based layout of the data stored in the files.

> **Warning:** This is the provisional storage layout and is under continuous development. Backwards compatibility is not expected to be supported before version 1.0.0 of the simphony-common library.

File

Lattice

Particles

Mesh

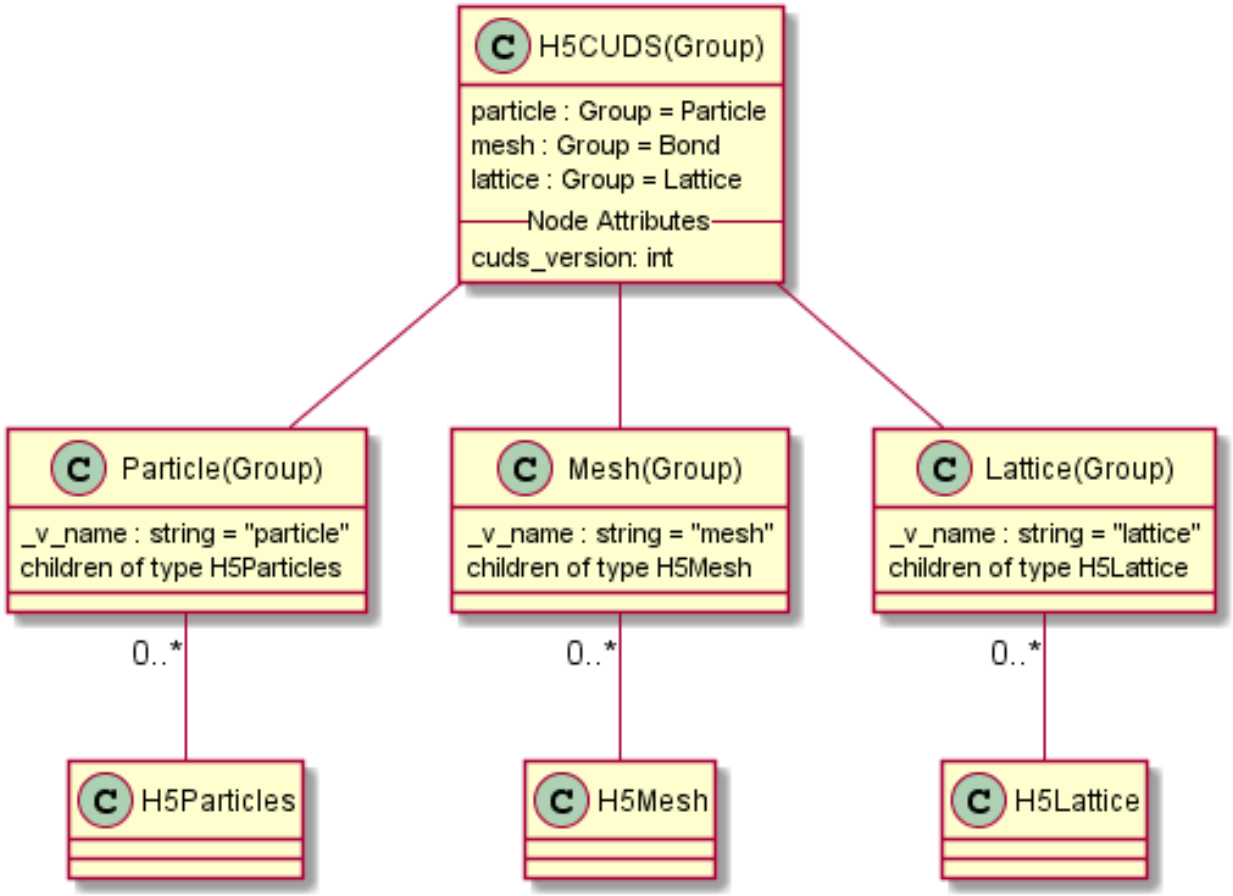

Fig. 2.5: **Figure 1:** Diagram of the top level layout of the HDF5 based files.
Each type of CUDS container is stored under the related section as an independent group.

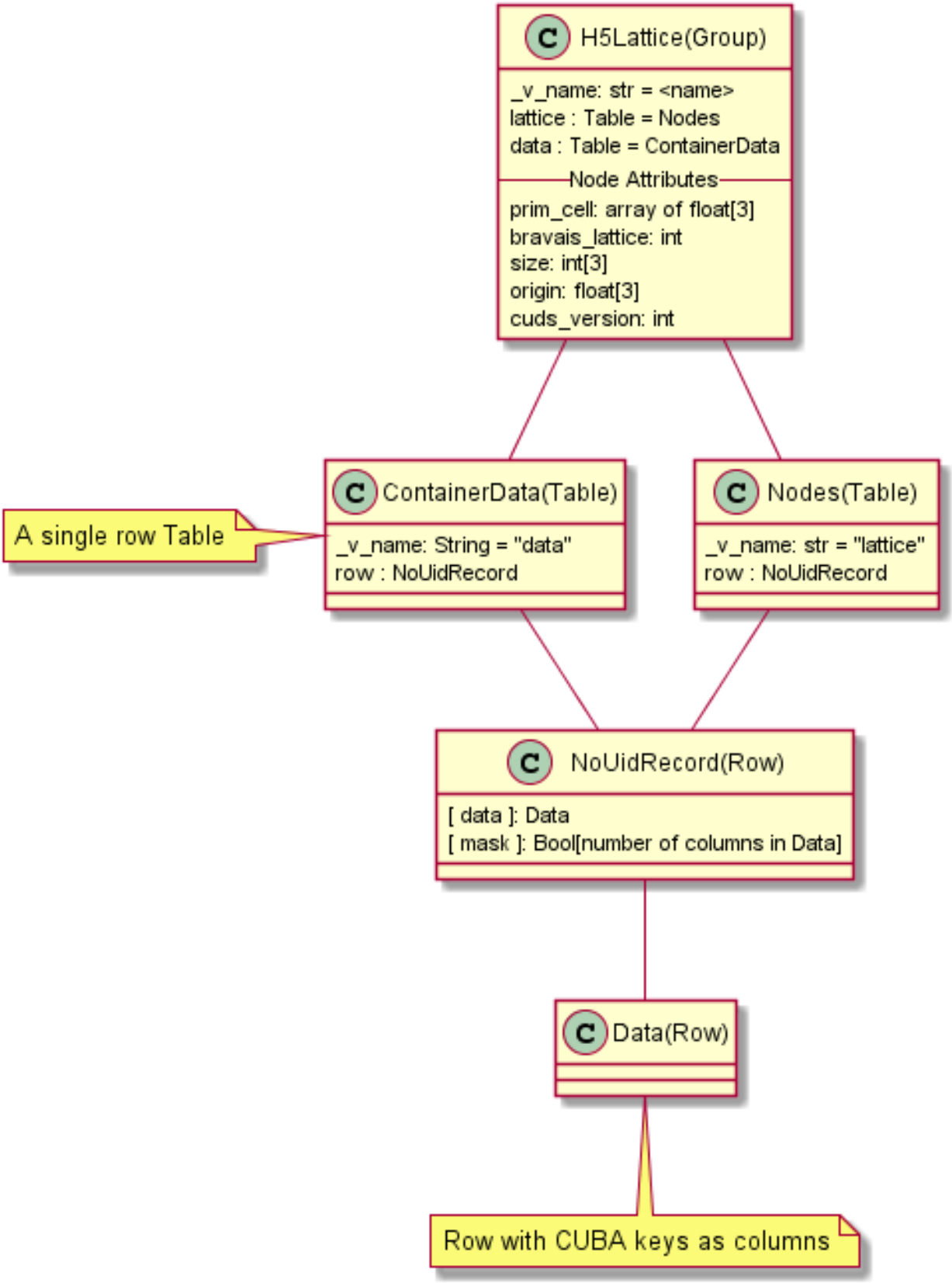

Fig. 2.6: **Figure 2:** Diagram of the Lattice based storage.

The Lattice is stored using two table nodes, one for the container *data* attribute and one for the lattice nodes data information. The nodes data are stored using the `numpy.ndenumerate` function to convert from i,j,k lattice coordinates to a flat index.

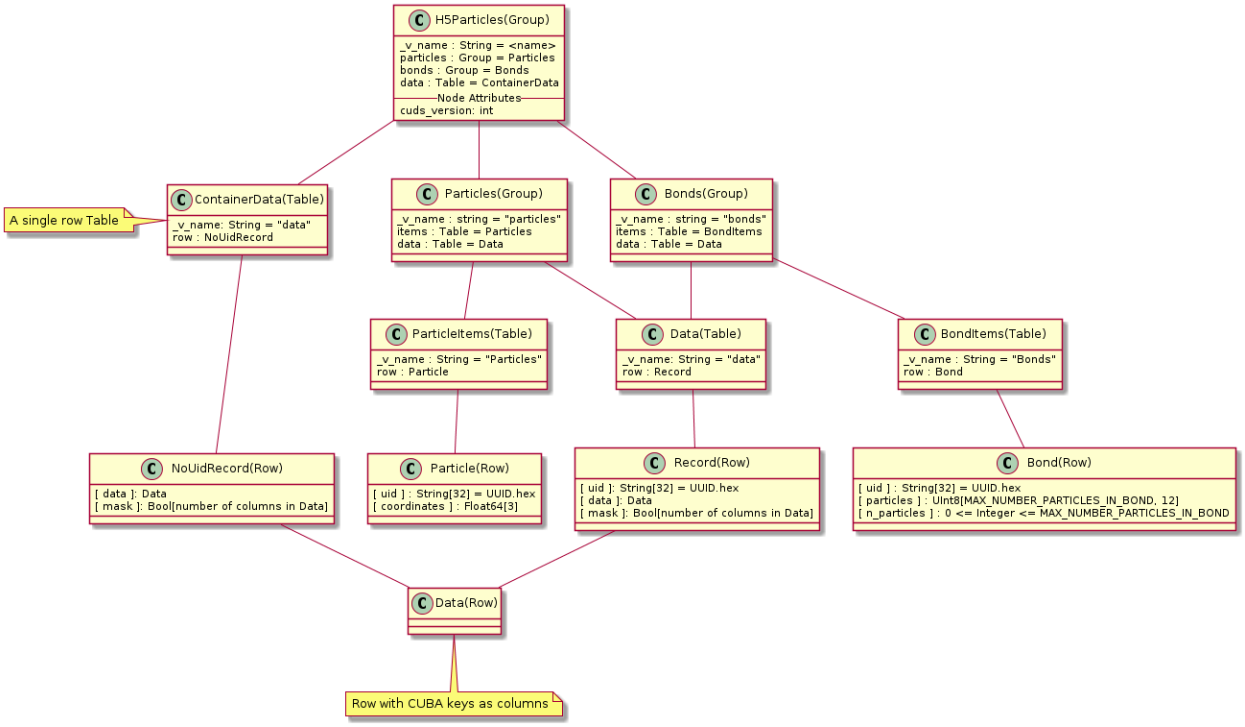

Fig. 2.7: **Figure 3:** Diagram of the Particles based storage.

The Particles container is stored using one table for the container *data* attribute and two groups to holding the particle and bond items separately. Each item group has two tables one for the item information (i.e. particle or bond) and one for the item *data*. Indexing into the item and data tables takes place by using the same uid hex for both.

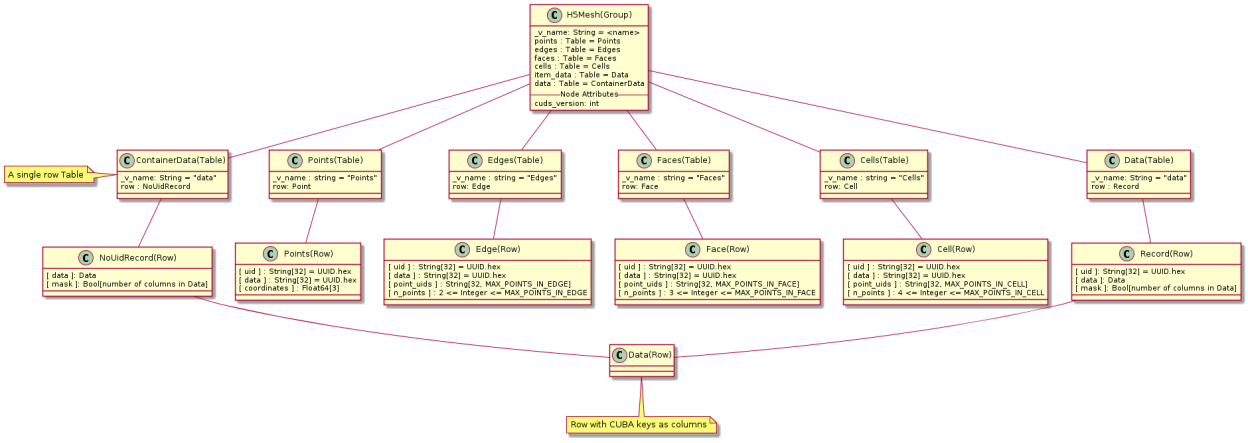

Fig. 2.8: **Figure 4:** Diagram of the Mesh based storage.

The Mesh container is stored using 6 tables, one for the container *data* attribute, one for all the item data information, one for the points and one for each type of elements (i.e. edge, face and cell). Indexing to the point or element tables is using the item uid while the item `data` information is accessed using a separate set of uids mapping to the entries in the `data` table.

CHAPTER 3

API Reference

3.1 Core

Core components and objects of the simphony package.

For a list of the CUBA-keywords, see CUBA-Keywords

Classes

DataContainer(*args, **kwargs)	A DataContainer instance

Implementation

class simphony.core.data_container.**DataContainer**(**args*, ***kwargs*)

Bases: dict

A DataContainer instance

The DataContainer object is implemented as a python dictionary whose keys are restricted to be members of the CUBA enum class.

The data container can be initialized like a typical python dict using the mapping and iterables where the keys are CUBA enum members.

For convenience keywords can be passed as capitalized CUBA enum members:

```
>>> DataContainer(ACCELERATION=234)  # CUBA.ACCELERATION is 22
{<CUBA.ACCELERATION: 22>: 234}
```

update(**args*, ***kwargs*)

3.2 CUDS

3.2.1 Abstract CUDS interfaces

Containers

abstractmesh.ABCMesh
abstractparticles.ABCParticles
abstractlattice.ABCLattice

Description

class simphony.cuds.abc_mesh.**ABCMesh**
Abstract base class for mesh.

name
str – name of mesh

add_cells(*cell*)
Adds a set of new cells to the mesh.

Parameters **cells** (*iterable of Cell*) – Cell to be added to the mesh

Raises *ValueError :* – If other cell with a duplicated uid was already in the mesh

add_edges(*edge*)
Adds a set of new edges to the mesh.

Parameters **edges** (*iterable of Edge*) – Edge to be added to the mesh

Raises *ValueError :* – If other edge with a duplicated uid was already in the mesh

add_faces(*face*)
Adds a set of new faces to the mesh.

Parameters **faces** (*iterable of Face*) – Face to be added to the mesh

Raises *ValueError :* – If other face with a duplicated uid was already in the mesh

add_points(*points*)
Adds a set of new points to the mesh.

Parameters **points** (*iterable of Point*) – Points to be added to the mesh

Raises *ValueError :* – If other point with a duplicated uid was already in the mesh.

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** – The number of items of item_type in the container.

Return type int

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

get_cell(*uid*)
Returns a cell with a given uid.

Returns the cell stored in the mesh identified by uid. If such a cell does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired cell.

Returns **cell** – Cell identified by uid

Return type *Cell*

Raises

- *KeyError :* – If the cell identified by uuid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_edge(*uid*)
Returns an edge with a given uid.

Returns the edge stored in the mesh identified by uid. If such edge do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired edge.

Returns **edge** – Edge identified by uid

Return type *Edge*

Raises

- *KeyError :* – If the edge identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_face(*uid*)
Returns a face with a given uid.

Returns the face stored in the mesh identified by uid. If such a face does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired face.

Returns **face** – Face identified by uid

Return type *Face*

Raises

- *KeyError :* – If the face identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_point(*uid*)
Returns a point with a given uid.

Returns the point stored in the mesh identified by uid. If such point do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired point.

Returns **point** – Mesh point identified by uuid

Return type *Point*

Raises

- *KeyError :* – If the point identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

has_cells()
Check if the mesh has cells

Returns **result** – True of there are cells inside the mesh, False otherwise

Return type bool

has_edges()
Check if the mesh has edges

Returns **result** – True of there are edges inside the mesh, False otherwise

Return type bool

has_faces()
Check if the mesh has faces

Returns **result** – True of there are faces inside the mesh, False otherwise

Return type bool

iter_cells(*uids=None*)
Returns an iterator over cells.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the cells are returned in the same order the uids are returned by the iterable. If uids is None, then all cells are returned by the interable and there is no restriction on the order that they are returned.

Yields **cell** (*Cell*)

iter_edges(*uids=None*)
Returns an iterator over edges.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the edges are returned in the same order the uids are returned by the iterable. If uids is None, then all edges are returned by the interable and there is no restriction on the order that they are returned.

Yields **edge** (*Edge*)

iter_faces(*uids=None*)
Returns an iterator over faces.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the faces are returned in the same order the uids are returned by the iterable. If uids is None, then all faces are returned by the interable and there is no restriction on the order that they are returned.

Yields **face** (*Face*)

iter_points(*uids=None*)
Returns an iterator over points.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the points are returned in the same order the uids are returned by the iterable. If uids is None, then all points are returned by the interable and there is no restriction on the order that they are returned.

Yields **point** (*Point*)

update_cells(*cell*)
Updates the information of a set of cells.

Gets the mesh cell identified by the same uid as the provided cell and updates its information with the one provided with the new cell.

Parameters **cells** (*iterable of Cell*) – Cell to be updated

Raises *ValueError :* – If the any cell was not found in the mesh

update_edges(*edge*)
Updates the information of a set of edges.

Gets the mesh edge identified by the same uid as the provided edge and updates its information with the one provided with the new edge.

Parameters **edges** (*iterable of Edge*) – Edge to be updated

Raises *ValueError :* – If the any edge was not found in the mesh

update_faces (*face*)
Updates the information of a set of faces.

Gets the mesh face identified by the same uid as the provided face and updates its information with the one provided with the new face.

Parameters **faces** (*iterable of Face*) – Face to be updated

Raises *ValueError :* – If the any face was not found in the mesh

update_points (*point*)
Updates the information of a set of points.

Gets the mesh point identified by the same uid as the provided point and updates its information with the one provided with the new point.

Parameters **points** (*iterable of Point*) – Point to be updated

Raises *ValueError :* – If the any point was not found in the mesh

class simphony.cuds.abc_particles.**ABCParticles**
Abstract base class for a container of particles items.

name
str – name of particles item.

data
DataContainer – The data associated with the container

add_bonds (*iterable*)
Adds a set of bonds to the container.

Also like with particles, if any bond has a defined uid, it won't add the bond if a bond with the same uid already exists, and if the bond has no uid the particle container will generate an uid. If the user wants to replace an existing bond in the container there is an 'update_bonds' method for that purpose.

Parameters **iterable** (*iterable of Bond objects*) – the new bond that will be included in the container.

Returns **uuid** – The uuids of the added bonds.

Return type list of uuid.UUID

Raises *ValueError :* – when there is a bond with an uuid that already exists in the container.

Examples

Add a set of bonds to a Particles container.

```
>>> bonds_list = [Bond(), Bond()]
>>> particles = Particles(name="foo")
>>> particles.add_bonds(bonds_list)
```

add_particles (*iterable*)
Adds a set of particles from the provided iterable to the container.

If any particle have no uids, the container will generate a new uids for it. If the particle has already an uids, it won't add the particle if a particle with the same uid already exists. If the user wants to replace an existing particle in the container there is an 'update_particles' method for that purpose.

Parameters **iterable** (*iterable of Particle objects*) – the new set of particles that will be included in the container.

Returns **uids** – The uids of the added particles.

Return type list of uuid.UUID

Raises *ValueError :* – when there is a particle with an uids that already exists in the container.

Examples

Add a set of particles to a Particles container.

```
>>> particle_list = [Particle(), Particle()]
>>> particles = Particles(name="foo")
>>> uids = particles.add_particles(particle_list)
```

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** – The number of items of item_type in the container.

Return type int

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

get_bond(*uid*)

Returns a copy of the bond with the 'bond_id' id.

Parameters **uid** (*uuid.UUID*) – the uid of the bond

Raises *KeyError :* – when the bond is not in the container.

Returns **bond** – A copy of the internally stored bond info.

Return type *Bond*

get_particle(*uid*)

Returns a copy of the particle with the 'particle_id' id.

Parameters **uid** (*uuid.UUID*) – the uid of the particle

Raises *KeyError :* – when the particle is not in the container.

Returns **particle** – A copy of the internally stored particle info.

Return type *Particle*

has_bond(*uid*)

Checks if a bond with the given uid already exists in the container.

has_particle(*uid*)

Checks if a particle with the given uid already exists in the container.

iter_bonds(*uids=None*)

Generator method for iterating over the bonds of the container.

It can receive any kind of sequence of bond ids to iterate over those concrete bond. If nothing is passed as parameter, it will iterate over all the bonds.

Parameters **uids** (*iterable of uuid.UUID, optional*) – sequence containing the id's of the bond that will be iterated. When the uids are provided, then the bonds are returned in the same order the uids are returned by the iterable. If uids is None, then all bonds are returned by the interable and there is no restriction on the order that they are returned.

Yields **bond** (*Bond*) – The next Bond item

Raises *KeyError :* – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> particles = Particles(name="foo")
>>> ...
>>> for bond in particles.iter_bonds([id1, id2, id3]):
        ...  #do stuff
```

```
>>> for bond in particles.iter_bond():
        ...  #do stuff; it will iterate over all the bond
```

iter_particles(*uids=None*)

Generator method for iterating over the particles of the container.

It can receive any kind of sequence of particle uids to iterate over those concrete particles. If nothing is passed as parameter, it will iterate over all the particles.

Parameters **uids** (*iterable of uuid.UUID, optional*) – sequence containing the uids of the particles that will be iterated. When the uids are provided, then the particles are returned in the same order the uids are returned by the iterable. If uids is None, then all particles are returned by the interable and there is no restriction on the order that they are returned.

Yields **particle** (*Particle*) – The Particle item.

Raises *KeyError :* – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> particles = Particles(name="foo")
>>> ...
>>> for particle in particles.iter_particles([uid1, uid2, uid3]):
    ...  #do stuff
>>> for particle in particles.iter_particles():
    ...  #do stuff
```

remove_bonds(*uids*)

Remove the bonds with the provided uids.

The uids passed as parameter should exists in the container. If any uid doesn't exist, an exception will be raised.

Parameters **uids** (*uuid.UUID*) – the uid of the bond to be removed.

Examples

Having a set of uids of existing bonds, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> particles.remove_bonds([uid1, uid2])
```

remove_particles(*uids*)

Remove the particles with the provided uids from the container.

The uids inside the iterable should exists in the container. Otherwise an exception will be raised.

Parameters **uid** (*uuid.UUID*) – the uid of the particle to be removed.

Raises *KeyError :* – If any particle doesn't exist.

Examples

Having a set of uids of existing particles, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> particles.remove_particles([uid1, uid2])
```

update_bonds(*iterable*)

Updates a set of bonds from the provided iterable.

Takes the uids of the bonds and searches inside the container for those bond. If the bonds exists, they are replaced in the container. If any bond doesn't exist, it will raise an exception.

Parameters **iterable** (*iterable of Bond objects*) – the bonds that will be replaced.

Raises *ValueError :* – If any bond doesn't exist.

Examples

Given a set of Bond objects that already exists in the container (taken with the 'get_bond' method for example) just call the function passing the set of Bond as parameter.

```
>>> particles = Particles(name="foo")
>>> ...
>>> bond1 = particles.get_bond(uid1)
>>> bond2 = particles.get_bond(uid2)
>>> ... #do whatever you want with the bonds
>>> particles.update_bonds([bond1, bond2])
```

update_particles(*iterable*)

Updates a set of particles from the provided iterable.

Takes the uids of the particles and searches inside the container for those particles. If the particles exists, they are replaced in the container. If any particle doesn't exist, it will raise an exception.

Parameters **iterable** (*iterable of Particle objects*) – the particles that will be replaced.

Raises *ValueError :* – If any particle inside the iterable does not exist.

Examples

Given a set of Particle objects that already exists in the container (taken with the 'get_particle' method for example), just call the function passing the Particle items as parameter.

```
>>> part_container = Particles(name="foo")
>>> ... #do whatever you want with the particles
>>> part_container.update_particles([part1, part2])
```

class simphony.cuds.abc_lattice.**ABCLattice**

Abstract base class for a lattice.

name
str – name of lattice

primitive_cell
PrimitiveCell – primitive cell specifying the 3D Bravais lattice

size
int[3] – lattice dimensions

origin
float[3] – lattice origin

data
DataContainer – high level CUBA data assigned to lattice

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** – The number of items of item_type in the container.

Return type int

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

get_coordinate(*ind*)
Get coordinate of the given index coordinate.

Parameters **ind** (*int[3]*) – node index coordinate

Returns **coordinates**

Return type float[3]

get_node(*index*)
Get the lattice node corresponding to the given index.

Parameters **index** (*int[3]*) – node index coordinate

Returns **node**

Return type *LatticeNode*

iter_nodes(*indices=None*)
Get an iterator over the LatticeNodes described by the indices.

Parameters **indices** (*iterable set of int[3], optional*) – When indices (i.e. node index coordinates) are provided, then nodes are returned in the same order of the provided indices. If indices is None, there is no restriction on the order the nodes that are returned.

Returns An iterator over LatticeNode objects

Return type iterator

primitive_cell

update_nodes(*nodes*)
Update the corresponding lattice nodes.

Parameters **nodes** (*iterator of LatticeNodes*) –

3.2.2 Pure Python implementation

Classes

PrimitiveCell(p1, p2, p3, bravais_lattice)	A primitive cell of a Bravais lattice.
BravaisLattice	The 3D Bravais lattices
Lattice(name, primitive_cell, size, origin)	A Bravais lattice.
LatticeNode(index[, data])	A single node of a lattice.
Particles(name)	Class that represents a container of particles and bonds.
Bond(particles[, uid, data])	Class representing a bond.
Particle([coordinates, uid, data])	Class representing a particle.
Mesh(name)	Mesh object to store points and elements.
Point(coordinates[, uid, data])	Coordinates describing a point in the space
Edge(points[, uid, data])	Edge element
Face(points[, uid, data])	Face element
Cell(points[, uid, data])	Cell element

Functions

make_cubic_lattice(name, h, size[, origin])	Create and return a 3D cubic lattice.
make_body_centered_cubic_lattice(name, h, size)	Create and return a 3D body-centered cubic lattice.
make_face_centered_cubic_lattice(name, h, size)	Create and return a 3D face-centered cubic lattice.
make_rhombohedral_lattice(name, h, angle, size)	Create and return a 3D rhombohedral lattice.
make_tetragonal_lattice(name, hxy, hz, size)	Create and return a 3D tetragonal lattice.
make_body_centered_tetragonal_lattice(name, ...)	Create and return a 3D body-centered tetragonal lattice.
make_hexagonal_lattice(name, hxy, hz, size)	Create and return a 3D hexagonal lattice.
make_orthorhombic_lattice(name, hs, size[, ...])	Create and return a 3D orthorhombic lattice.
make_body_centered_orthorhombic_lattice(...)	Create and return a 3D body-centered orthorhombic lattice.
make_face_centered_orthorhombic_lattice(...)	Create and return a 3D face-centered orthorhombic lattice.
make_base_centered_orthorhombic_lattice(...)	Create and return a 3D base-centered orthorhombic lattice.
make_monoclinic_lattice(name, hs, beta, size)	Create and return a 3D monoclinic lattice.
make_base_centered_monoclinic_lattice(name, ...)	Create and return a 3D base-centered monoclinic lattice.
make_triclinic_lattice(name, hs, angles, size)	Create and return a 3D triclinic lattice.

Implementation

class simphony.cuds.lattice.**Lattice**(*name*, *primitive_cell*, *size*, *origin*)
A Bravais lattice. Stores references to data containers (node related data).

name
str – name of lattice

primitive_cell
PrimitiveCell – primitive cell specifying the 3D Bravais lattice

size
int[3] – lattice dimensions

origin
float[3] – lattice origin

data
DataContainer – high level CUBA data assigned to lattice

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** – The number of items of item_type in the container.

Return type int

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

data

get_node(*index*)
Get a copy of the node corresponding to the given index.

Parameters **index** (*int[3]*) – node index coordinate

Returns

Return type A reference to a LatticeNode object

iter_nodes(*indices=None*)
Get an iterator over the LatticeNodes described by the indices.

Parameters **indices** (*iterable set of int[3], optional*) – When indices (i.e. node index coordinates) are provided, then nodes are returned in the same order of the provided indices. If indices is None, there is no restriction on the order the nodes that are returned.

Returns

Return type A generator for LatticeNode objects

origin

size

update_nodes(*nodes*)
Update the corresponding lattice nodes (data copied).

Parameters **nodes** (*iterable of LatticeNode objects*) – reference to LatticeNode objects from where the data is copied to the Lattice

class simphony.cuds.lattice.**LatticeNode**(*index*, *data=None*)
A single node of a lattice.

index
tuple of int[3] – node index coordinate

data
DataContainer

simphony.cuds.lattice.**make_base_centered_monoclinic_lattice**(*name*, *hs*, *beta*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D base-centered monoclinic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float*) – lattice spacing in each axis direction
- **beta** (*float*) – angle between the (conventional) unit cell edges (in radians),
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_base_centered_orthorhombic_lattice**(*name*, *hs*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D base-centered orthorhombic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_body_centered_cubic_lattice**(*name*, *h*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D body-centered cubic lattice.

Parameters

- **name** (*str*) –
- **h** (*float*) – lattice spacing
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_body_centered_orthorhombic_lattice**(*name*, *hs*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D body-centered orthorhombic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_body_centered_tetragonal_lattice**(*name*, *hxy*, *hz*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D body-centered tetragonal lattice.

Parameters

- **name** (*str*) –
- **hxy** (*float*) – lattice spacing in the xy-plane
- **hz** (*float*) – lattice spacing in the z-direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_cubic_lattice**(*name*, *h*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D cubic lattice.

Parameters

- **name** (*str*) –
- **h** (*float*) – lattice spacing
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_face_centered_cubic_lattice**(*name*, *h*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D face-centered cubic lattice.

Parameters

- **name** (*str*) –
- **h** (*float*) – lattice spacing
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_face_centered_orthorhombic_lattice**(*name*, *hs*, *size*, *origin=(0, 0, 0)*)

Create and return a 3D face-centered orthorhombic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.

- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_hexagonal_lattice**(*name*, *hxy*, *hz*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D hexagonal lattice.

Parameters

- **name** (*str*) –
- **hxy** (*float*) – lattice spacing in the xy-plane
- **hz** (*float*) – lattice spacing in the z-direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_monoclinic_lattice**(*name*, *hs*, *beta*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D monoclinic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **beta** (*float*) – angle between the (conventional) unit cell edges (in radians),
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_orthorhombic_lattice**(*name*, *hs*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D orthorhombic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_rhombohedral_lattice**(*name*, *h*, *angle*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D rhombohedral lattice.

Parameters

- **name** (*str*) –
- **h** (*float*) – lattice spacing

- **angle** (*float*) – angle between the (conventional) unit cell edges (in radians)
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_tetragonal_lattice**(*name*, *hxy*, *hz*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D tetragonal lattice.

Parameters

- **name** (*str*) –
- **hxy** (*float*) – lattice spacing in the xy-plane
- **hz** (*float*) – lattice spacing in the z-direction
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

simphony.cuds.lattice.**make_triclinic_lattice**(*name*, *hs*, *angles*, *size*, *origin=(0, 0, 0)*)
Create and return a 3D triclinic lattice.

Parameters

- **name** (*str*) –
- **hs** (*float[3]*) – lattice spacings in each axis direction
- **angles** (*float[3]*) – angles between the (conventional) unit cell edges (in radians)
- **size** (*int[3]*) – Number of lattice nodes in each axis direction.
- **origin** (*float[3], default value = (0, 0, 0)*) – lattice origin

Returns **lattice** – A reference to a Lattice object.

Return type *Lattice*

Mesh module

This module contains the implementation to store, access, and modify a mesh

class simphony.cuds.mesh.**Cell**(*points*, *uid=None*, *data=None*)
Cell element

Element for storing 3D geometrical objects

Parameters

- **points** (*list of uid*) – list of points uids defining the cell.
- **uid** (*uuid.UUID*) – uid of the cell.
- **data** (DataContainer) – object to store data relative to the cell

classmethod **from_cell**(*cell*)

class simphony.cuds.mesh.**Edge**(*points*, *uid=None*, *data=None*)
Edge element

Element for storing 1D geometrical objects

Parameters

- **points** (*list of uid*) – list of points uids defining the edge.
- **uid** (*uuid.UUID*) – uid of the edge.
- **data** (DataContainer) – object to store data relative to the edge

classmethod from_edge(*edge*)

class simphony.cuds.mesh.**Element**(*points*, *uid=None*, *data=None*)
Element base class

Element for storing geometrical objects

Parameters

- **uid** – uid of the edge.
- **points** (*list of uid*) – list of points uids defining the edge.
- **data** (DataContainer) – object to store data relative to the element

points
list of uid – list of points uids defining the element.

uid
uuid.UUID – uid of the element

data
DataContainer – Element data

class simphony.cuds.mesh.**Face**(*points*, *uid=None*, *data=None*)
Face element

Element for storing 2D geometrical objects

Parameters

- **points** (*list of uid*) – list of points uids defining the face.
- **uid** (*uuid.UUID*) – uid of the face.
- **data** (DataContainer) – object to store data relative to the face

classmethod from_face(*face*)

class simphony.cuds.mesh.**Mesh**(*name*)
Mesh object to store points and elements.

Stores general mesh information Points and Elements such as Edges, Faces and Cells and provide the methods to interact with them. The methods are divided in four different blocks:

1.methods to get the related item with the provided uid;

2.methods to add a new item or replace;

3.generator methods that return iterators over all or some of the mesh items and;

4.inspection methods to identify if there are any edges, faces or cells described in the mesh.

Parameters **name** (*str*) – name of mesh

name
str – name of mesh

data
Data – Data relative to the mesh.

points
dictionary of Point – Points of the mesh.

edges
dictionary of Edge – Edges of the mesh.

faces
dictionary of Face – Faces of the mesh.

cells
dictionary of Cell – Cells of the mesh.

add_cells(*cells*)
Adds a set of new cells to the mesh.

Parameters **cells** (*iterable of Cell*) – Cell to be added to the mesh

Raises *ValueError :* – If other cell with a duplicated uid was already in the mesh

add_edges(*edges*)
Adds a set of new edges to the mesh.

Parameters **edges** (*iterable of Edge*) – Edge to be added to the mesh

Raises *ValueError :* – If other edge with a duplicated uid was already in the mesh

add_faces(*faces*)
Adds a set of new faces to the mesh.

Parameters **faces** (*iterable of Face*) – Face to be added to the mesh

Raises *ValueError :* – If other face with a duplicated uid was already in the mesh

add_points(*points*)
Adds a set of new points to the mesh.

Parameters **points** (*iterable of Point*) – Points to be added to the mesh

Raises *ValueError :* – If other point with a duplicated uid was already in the mesh.

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** – The number of items of item_type in the container.

Return type int

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

data

get_cell(*uid*)
Returns a cell with a given uid.

Returns the cell stored in the mesh identified by uid. If such a cell does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired cell.

Returns **cell** – Cell identified by uid

Return type *Cell*

Raises

- *KeyError :* – If the cell identified by uuid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_edge(*uid*)
Returns an edge with a given uid.

Returns the edge stored in the mesh identified by uid. If such edge do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired edge.

Returns **edge** – Edge identified by uid

Return type *Edge*

Raises

- *KeyError :* – If the edge identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_face(*uid*)
Returns a face with a given uid.

Returns the face stored in the mesh identified by uid. If such a face does not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired face.

Returns **face** – Face identified by uid

Return type *Face*

Raises

- *KeyError :* – If the face identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

get_point(*uid*)
Returns a point with a given uid.

Returns the point stored in the mesh identified by uid. If such point do not exists an exception is raised.

Parameters **uid** (*uuid.UUID*) – uid of the desired point.

Returns **point** – Mesh point identified by uuid

Return type *Point*

Raises

- *KeyError :* – If the point identified by uid was not found
- *TypeError :* – When `uid` is not uuid.UUID

has_cells()
Check if the mesh has cells

Returns **result** – True of there are cells inside the mesh, False otherwise

Return type bool

has_edges()
Check if the mesh has edges

Returns **result** – True of there are edges inside the mesh, False otherwise

Return type bool

has_faces()
Check if the mesh has faces

Returns **result** – True of there are faces inside the mesh, False otherwise

Return type bool

iter_cells(*uids=None*)
Returns an iterator over cells.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the cells are returned in the same order the uids are returned by the iterable. If uids is None, then all cells are returned by the interable and there is no restriction on the order that they are returned.

Yields **cell** (*Cell*)

iter_edges(*uids=None*)
Returns an iterator over edges.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the edges are returned in the same order the uids are returned by the iterable. If uids is None, then all edges are returned by the interable and there is no restriction on the order that they are returned.

Yields **edge** (*Edge*)

iter_faces(*uids=None*)
Returns an iterator over faces.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the faces are returned in the same order the uids are returned by the iterable. If uids is None, then all faces are returned by the interable and there is no restriction on the order that they are returned.

Yields **face** (*Face*)

iter_points(*uids=None*)
Returns an iterator over points.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the points are returned in the same order the uids are returned by the iterable. If uids is None, then all points are returned by the interable and there is no restriction on the order that they are returned.

Yields **cell** (*Cell*)

update_cells(*cells*)
Updates the information of a set of cells.

Gets the mesh cell identified by the same uid as the provided cell and updates its information with the one provided with the new cell.

Parameters **cells** (*iterable of Cell*) – Cell to be updated

Raises *ValueError :* – If the any cell was not found in the mesh

update_edges(*edges*)
Updates the information of a set of edges.

Gets the mesh edge identified by the same uid as the provided edge and updates its information with the one provided with the new edge.

Parameters **edges** (*iterable of Edge*) – Edge to be updated

Raises *ValueError :* – If the any edge was not found in the mesh

update_faces (*faces*)
Updates the information of a set of faces.

Gets the mesh face identified by the same uid as the provided face and updates its information with the one provided with the new face.

Parameters **faces** (*iterable of Face*) – Face to be updated

Raises *ValueError :* – If the any face was not found in the mesh

update_points (*points*)
Updates the information of a set of points.

Gets the mesh point identified by the same uid as the provided point and updates its information with the one provided with the new point.

Parameters **points** (*iterable of Point*) – Point to be updated

Raises *ValueError :* – If the any point was not found in the mesh

class simphony.cuds.mesh.**Point** (*coordinates*, *uid=None*, *data=None*)
Coordinates describing a point in the space

Set of coordinates (x,y,z) describing a point in the space and data about that point

Parameters

- **uid** (*uuid.UUID*) – uid of the point.
- **coordinates** (*list of double*) – set of coordinates (x,y,z) describing the point position.
- **data** (DataContainer) – object to store point data

uid
uuid.UUID – uid of the point.

data
DataContainer – object to store point data

coordinates
list of double – set of coordinates (x,y,z) describing the point position.

classmethod **from_point** (*point*)

class simphony.cuds.particles.**Bond** (*particles*, *uid=None*, *data=None*)
Class representing a bond.

uid
uuid.UUID – the uid of the bond

particles
tuple – tuple of uids of the particles that are participating in the bond.

data
DataContainer – DataContainer to store the attributes of the bond

classmethod **from_bond** (*bond*)

class simphony.cuds.particles.**Particle** (*coordinates=(0.0, 0.0, 0.0)*, *uid=None*, *data=None*)
Class representing a particle.

uid
uuid.UUID – the uid of the particle

coordinates
list / tuple – x,y,z coordinates of the particle

data
DataContainer – DataContainer to store the attributes of the particle

classmethod from_particle(*particle*)

class simphony.cuds.particles.**Particles**(*name*)
Class that represents a container of particles and bonds.

Class provides methods to add particles and bonds, remove them and update them.

name
str – name of the particle container

_particles
dict – data structure for particles storage

_bonds
dict – data structure for bonds storage

data
DataContainer – data attributes of the element

add_bonds(*iterable*)
Adds a set of bonds to the container.

Also like with particles, if any bond has a defined uid, it won't add the bond if a bond with the same uid already exists, and if the bond has no uid the particle container will generate an uid. If the user wants to replace an existing bond in the container there is an 'update_bonds' method for that purpose.

Parameters **iterable** (*iterable of Bond objects*) – the new bond that will be included in the container.

Returns **uuid** – The uuids of the added bonds.

Return type list of uuid.UUID

Raises *ValueError :* – when there is a bond with an uuid that already exists in the container.

Examples

Add a set of bonds to a Particles container.

```
>>> bonds_list = [Bond(), Bond()]
>>> particles = Particles(name="foo")
>>> particles.add_bond(bonds_list)
```

add_particles(*iterable*)
Adds a set of particles from the provided iterable to the container.

If any particle have no uids, the container will generate a new uids for it. If the particle has already an uids, it won't add the particle if a particle with the same uid already exists. If the user wants to replace an existing particle in the container there is an 'update_particles' method for that purpose.

Parameters **iterable** (*iterable of Particle objects*) – the new set of particles that will be included in the container.

Returns **uids** – The uids of the added particles.

Return type list of uuid.UUID

Raises *ValueError :* – when there is a particle with an uids that already exists in the container.

Examples

Add a set of particles to a Particles container.

```
>>> particle_list = [Particle(), Particle()]
>>> particles = Particles(name="foo")
>>> uids = particles.add_particles(particle_list)
```

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** – The number of items of item_type in the container.

Return type int

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

data

get_bond(*uid*)
Returns a copy of the bond with the 'bond_id' id.

Parameters **uid** (*uuid.UUID*) – the uid of the bond

Raises *KeyError :* – when the bond is not in the container.

Returns **bond** – A copy of the internally stored bond info.

Return type *Bond*

get_particle(*uid*)
Returns a copy of the particle with the 'particle_id' id.

Parameters **uid** (*uuid.UUID*) – the uid of the particle

Raises *KeyError :* – when the particle is not in the container.

Returns **particle** – A copy of the internally stored particle info.

Return type *Particle*

has_bond(*uid*)
Checks if a bond with the given uid already exists in the container.

has_particle(*uid*)
Checks if a particle with the given uid already exists in the container.

iter_bonds(*uids=None*)
Generator method for iterating over the bonds of the container.

It can receive any kind of sequence of bond ids to iterate over those concrete bond. If nothing is passed as parameter, it will iterate over all the bonds.

Parameters **uids** (*iterable of uuid.UUID, optional*) – sequence containing the id's of the bond that will be iterated. When the uids are provided, then the bonds are returned in the same order the uids are returned by the iterable. If uids is None, then all bonds are returned by the interable and there is no restriction on the order that they are returned.

Yields **bond** (*Bond*) – The next Bond item

Raises *KeyError :* – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> part_container = Particles(name="foo")
>>> ...
>>> for bond in part_container.iter_bonds([id1, id2, id3]):
        ...  #do stuff
        #take the bond back to the container so it will be updated
        #in case we need it
        part_container.update_bond(bond)
```

```
>>> for bond in part_container.iter_bond():
        ...  #do stuff; it will iterate over all the bond
        #take the bond back to the container so it will be updated
        #in case we need it
        part_container.update_bond(bond)
```

iter_particles(*uids=None*)

Generator method for iterating over the particles of the container.

It can receive any kind of sequence of particle uids to iterate over those concrete particles. If nothing is passed as parameter, it will iterate over all the particles.

Parameters **uids** (*iterable of uuid.UUID, optional*) – sequence containing the uids of the particles that will be iterated. When the uids are provided, then the particles are returned in the same order the uids are returned by the iterable. If uids is None, then all particles are returned by the interable and there is no restriction on the order that they are returned.

Yields **particle** (*Particle*) – The Particle item.

Raises *KeyError :* – if any of the ids passed as parameters are not in the container.

Examples

It can be used with a sequence as parameter or without it:

```
>>> part_container = Particles(name="foo")
>>> ...
>>> for particle in part_container.iter_particles([uid1, uid2, uid3]):
        ...  #do stuff
        #take the particle back to the container so it will be updated
        #in case we need it
        part_container.update_particle(particle)
```

```
>>> for particle in part_container.iter_particles():
        ...  #do stuff; it will iterate over all the particles
        #take the particle back to the container so it will be updated
        #in case we need it
        part_container.update_particle(particle)
```

remove_bonds(*uids*)

Remove the bonds with the provided uids.

The uids passed as parameter should exists in the container. If any uid doesn't exist, an exception will be raised.

Parameters **uids** (*uuid.UUID*) – the uid of the bond to be removed.

Examples

Having a set of uids of existing bonds, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> bond1 = particles.get_bond(uid1)
>>> bond2 = particles.get_bond(uid2)
>>> ...
>>> particles.remove_bonds([bond1.uid, bond2.uid])
or
>>> particles.remove_bond([uid1, uid2])
```

remove_particles(*uids*)

Remove the particles with the provided uids from the container.

The uids inside the iterable should exists in the container. Otherwise an exception will be raised.

Parameters **uid** (*uuid.UUID*) – the uid of the particle to be removed.

Raises *KeyError :* – If any particle doesn't exist.

Examples

Having a set of uids of existing particles, pass it to the method.

```
>>> particles = Particles(name="foo")
>>> ...
>>> particle1 = particles.get_particle(uid1)
>>> particle2 = particles.get_particle(uid2)
>>> ...
>>> particles.remove_particle([part1.uid, part2.uid)
or directly
>>> particles.remove_particle([uid1, uid2])
```

update_bonds(*iterable*)

Updates a set of bonds from the provided iterable.

Takes the uids of the bonds and searches inside the container for those bond. If the bonds exists, they are replaced in the container. If any bond doesn't exist, it will raise an exception.

Parameters **iterable** (*iterable of Bond objects*) – the bonds that will be replaced.

Raises *ValueError :* – If any bond doesn't exist.

Examples

Given a set of Bond objects that already exists in the container (taken with the 'get_bond' method for example) just call the function passing the set of Bond as parameter.

```
>>> particles = Particles(name="foo")
>>> ...
>>> bond1 = particles.get_bond(uid1)
>>> bond2 = particles.get_bond(uid2)
>>> ... #do whatever you want with the bonds
>>> particles.update_bond([bond1, bond2])
```

update_particles (*iterable*)

Updates a set of particles from the provided iterable.

Takes the uids of the particles and searches inside the container for those particles. If the particles exists, they are replaced in the container. If any particle doesn't exist, it will raise an exception.

Parameters **iterable** (*iterable of Particle objects*) – the particles that will be replaced.

Raises *ValueError :* – If any particle inside the iterable does not exist.

Examples

Given a set of Particle objects that already exists in the container (taken with the 'get_particle' method for example), just call the function passing the Particle items as parameter.

```
>>> part_container = Particles(name="foo")
>>> ...
>>> part1 = part_container.get_particle(uid1)
>>> part2 = part_container.get_particle(uid2)
>>> ... #do whatever you want with the particles
>>> part_container.update_particle([part1, part2])
```

3.3 HDF5 IO

The CUDS to HDF5 file adapters.

Classes

H5CUDS(handle)	Access to CUDS-hdf5 formatted files.
DataContainerTable(root[, name, record])	A proxy class to an HDF5 group node with serialised DataContainers.
IndexedDataContainerTable(root[, name, ...])	A proxy class to an HDF5 group node with serialised DataContainers.
H5Particles(group)	An HDF5 backed particle container.
H5Lattice(group)	H5Lattice object to use H5CUDS lattices.
H5Mesh(group, meshFile)	H5Mesh.
H5CUDSItems(root, record[, name])	A proxy class to an HDF5 group node with serialised CUDS items.

Table descriptions

Implementation

class simphony.io.h5_cuds.**H5CUDS**(*handle*)
Bases: object

Access to CUDS-hdf5 formatted files.

add_dataset(*container*)
Add a CUDS container

Parameters **container** (*{ABCMesh, ABCParticles, ABCLattice}*) – The CUDS container to be added.

Raises

- TypeError: – If the container type is not supported by the engine.
- ValueError: – If there is already a dataset with the given name.

close()
Closes a file

get_dataset(*name*)
Get the dataset

Parameters **name** (*str*) – name of CUDS container to be retrieved.

Returns A proxy of the dataset named name that is stored internally in the File.

Return type container

Raises ValueError: – If there is no dataset with the given name

get_dataset_names()
Returns the names of the all the datasets in the engine workspace.

iter_datasets(*names=None*)
Returns an iterator over a subset or all of the containers.

Parameters **names** (*sequence of str, optional*) – names of specific containers to be iterated over. If names is not given, then all containers will be iterated over.

classmethod **open**(*filename*, *mode='a'*, *title=''*)
Returns a SimPhony file and returns an opened CudsFile

Parameters

- **filename** (*str*) – Name of file to be opened.
- **mode** (*str*) – The mode to open the file:
  - w – Write; a new file is created (an existing file with the same name would be deleted).
  - a – Append; an existing file is opened for reading and writing, and if the file does not exist it is created.
  - r – ReadOnly; This is a very restrictive mode that will through errors at any attempt to modify the data.
- **title** (*str*) – Title attribute of root node (only applies to a file which is being created)
- **Raises** (*Raises*) –
- **------** –
- **ValueError** – If the file has an incompatible version

remove_dataset(*name*)
Remove a dataset from the engine

Parameters **name** (*str*) – name of CUDS container to be deleted

Raises ValueError: – If there is no dataset with the given name

valid()
Checks if file is valid (i.e. open)

class simphony.io.h5_particles.**H5BondItems**(*root*, *name='bonds'*)
Bases: *simphony.io.h5_cuds_items.H5CUDSItems*

A proxy class to an HDF5 group node with serialised Bonds

The class implements the Mutable-Mapping api where each Bond instance is mapped to uid.

class simphony.io.h5_particles.**H5ParticleItems**(*root*, *name='particles'*)
Bases: *simphony.io.h5_cuds_items.H5CUDSItems*

A proxy class to an HDF5 group node with serialised Particles

The class implements the Mutable-Mapping api where each Particle instance is mapped to uid.

class simphony.io.h5_particles.**H5Particles**(*group*)
Bases: *simphony.cuds.abc_particles.ABCParticles*

An HDF5 backed particle container.

add_bonds(*iterable*)
Add a set of bonds.

If the bonds have an uid then they are used. If any of the bond's uid is None then a uid is generated for the bond.

Returns **uid** – uid of bond

Return type uuid.UUID

Raises *ValueError :* – if an uid is given which already exists.

add_particles(*iterable*)
Add a set of particles.

If the particles have a uid set then they are used. If any of the particle's uid is None then a new uid is generated for the particle.

Returns **uid** – uid of particle.

Return type uuid.UUID

Raises *ValueError :* – Any particle uid already exists in the container.

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** – The number of items of item_type in the container.

Return type int

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

data

get_bond(*uid*)

get_particle(*uid*)

has_bond(*uid*)
Checks if a bond with uid "uid" exists in the container.

has_particle(*uid*)
Checks if a particle with uid "uid" exists in the container.

iter_bonds(*ids=None*)
Get iterator over particles

iter_particles(*ids=None*)
Get iterator over particles

name
The name of the container

remove_bonds(*uids*)

remove_particles(*uids*)

update_bonds(*iterable*)

update_particles(*iterable*)

class simphony.io.h5_lattice.**H5Lattice**(*group*)
Bases: *simphony.cuds.abc_lattice.ABCLattice*

H5Lattice object to use H5CUDS lattices.

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** – The number of items of item_type in the container.

Return type int

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

classmethod **create_new**(*group*, *primitive_cell*, *size*, *origin*, *record=None*)
Create a new lattice in H5CUDS file.

Parameters

- **group** (*HDF5 group in PyTables file*) – reference to a group (folder) in PyTables file where the tables for lattice and data will be located
- **primitive_cell** (*PrimitiveCell*) – primitive cell specifying the 3D Bravais lattice
- **size** (*int[3]*) – number of lattice nodes (in the direction of each axis).
- **origin** (*float[3]*) – origin of lattice
- **record** (*tables.IsDescription*) – A class that describes column types for PyTables table.

data

get_node(*index*)
Get a copy of the node corresponding to the given index.

Parameters **index** (*int[3]*) – node index coordinate

Returns

Return type A reference to a LatticeNode object

iter_nodes (*indices=None*)
Get an iterator over the LatticeNodes described by the ids.

Parameters **indices** (*iterable set of int[3], optional*) – node index coordinates

Returns

Return type A generator for LatticeNode objects

name

origin

size

update_nodes (*nodes*)
Updates H5Lattice data for a LatticeNode

Parameters **nodes** (*iterable of LatticeNode objects*) – reference to LatticeNode objects

Mesh File

This module contains the implentation to store, acces, and modify a file storing mesh data

class simphony.io.h5_mesh.**H5Mesh** (*group*, *meshFile*)
Bases: *simphony.cuds.abc_mesh.ABCMesh*

H5Mesh.

Interface of the mesh file driver. Stores general mesh information Points and Elements such as Edges, Faces and Cells and provide the methods to interact with them. The methods are divided in four diferent blocks:

1.methods to get the related item with the provided uid;

2.methods to add a new item or replace;

3.generator methods that return iterators over all or some of the mesh items and;

4.inspection methods to identify if there are any edges, faces or cells described in the mesh.

data
Data – Data relative to the mesh

name
String – Name of the mesh

See also:

get_point, *get_edge*, *get_face*, *get_cell*, add_point, add_edge, add_face, add_cell, update_point, update_edge, update_face, update_cell, *iter_points*, *iter_edges*, *iter_faces*, *iter_cells*, *has_edges*, *has_faces*, *has_cells*, _create_points_table, _create_edges_table, _create_faces_table, _create_cells_table

add_cells (*cells*)
Adds a new set of cells to the mesh container.

Parameters **cells** (*iterable of Cell*) – Cells to be added to the mesh container

Raises KeyError – If other cell with the same uid was already in the mesh

add_edges (*edges*)
Adds a new set of edges to the mesh container.

Parameters **edges** (*iterable of Edge*) – Edges to be added to the mesh container

Raises KeyError – If other edge with the same uid was already in the mesh

add_faces(*faces*)
Adds a new set of faces to the mesh container.

Parameters **faces** (*iterable of Face*) – Faces to be added to the mesh container

Raises KeyError – If other face with the same uid was already in the mesh

add_points(*points*)
Adds a new set of points to the mesh container.

Parameters **points** (*iterable of Point*) – Points to be added to the mesh container

Raises KeyError – If other point with the same uid was already in the mesh

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** – The number of items of item_type in the container.

Return type int

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

data

get_cell(*uid*)
Returns an cell with a given uid.

Returns the cell stored in the mesh identified by uid . If such cell do not exists a exception is raised.

Parameters **uid** (*UUID*) – uid of the desired cell.

Returns Cell identified by uid

Return type *Cell*

Raises Exception – If the cell identified by uid was not found

get_edge(*uid*)
Returns an edge with a given uid.

Returns the edge stored in the mesh identified by uid. If such edge do not exists a exception is raised.

Parameters **uid** (*UUID*) – uid of the desired edge.

Returns Edge identified by uid

Return type *Edge*

Raises Exception – If the edge identified by uid was not found

get_face(*uid*)
Returns an face with a given uid.

Returns the face stored in the mesh identified by uid. If such face do not exists a exception is raised.

Parameters **uid** (*UUID*) – uid of the desired face.

Returns Face identified by uid

Return type *Face*

Raises Exception – If the face identified by uid was not found

get_point(*uid*)
Returns a point with a given uid.

Returns the point stored in the mesh identified by uid. If such point do not exists an exception is raised.

Parameters **uid** (*UUID*) – uid of the desired point.

Returns Mesh point identified by uid

Return type *Point*

Raises `Exception` – If the point identified by uid was not found

has_cells()
Check if the mesh container has cells

Returns True of there are cells inside the mesh, False otherwise

Return type bool

has_edges()
Check if the mesh container has edges

Returns True of there are edges inside the mesh, False otherwise

Return type bool

has_faces()
Check if the mesh container has faces

Returns True of there are faces inside the mesh, False otherwise

Return type bool

iter_cells(*uids=None*)
Returns an iterator over cells.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the cells are returned in the same order the uids are returned by the iterable. If uids is None, then all cells are returned by the interable and there is no restriction on the order that they are returned.

Returns Iterator over the selected cells

Return type iter

iter_edges(*uids=None*)
Returns an iterator over edges.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the edges are returned in the same order the uids are returned by the iterable. If uids is None, then all edges are returned by the interable and there is no restriction on the order that they are returned.

Returns Iterator over the selected edges

Return type iter

iter_faces(*uids=None*)
Returns an iterator over faces.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the faces are returned in the same order the uids are returned by the iterable. If uids is None, then all faces are returned by the interable and there is no restriction on the order that they are returned.

Returns Iterator over the faces

Return type iter

iter_points(*uids=None*)
Returns an iterator over points.

Parameters **uids** (*iterable of uuid.UUID or None*) – When the uids are provided, then the points are returned in the same order the uids are returned by the iterable. If uids is None, then all points are returned by the interable and there is no restriction on the order that they are returned.

Returns Iterator over the points

Return type iter

name

update_cells(*cells*)
Updates the information of every cell in cells.

Gets the mesh cells identified by the same uids as the ones provided in cells and updates their information.

Parameters **cellss** (*iterable of Cell*) – Cells to be updated.

Raises KeyError – If any cell was not found in the mesh container.

update_edges(*edges*)
Updates the information of an edge.

Gets the mesh edges identified by the same uids as the ones provided edges and updates their information.

Parameters **edges** (*iterable of Edge*) – Edges to be updated.

Raises KeyError – If any edge was not found in the mesh container.

update_faces(*faces*)
Updates the information of a face.

Gets the mesh faces identified by the same uids as the ones provided in faces and updates their information.

Parameters **faces** (*iterable of Face*) – Faces to be updated.

Raises KeyError – If any face was not found in the mesh container.

update_points(*points*)
Updates the information of a point.

Gets the mesh points identified by the same uids as the ones provided points and updates their information.

Parameters **points** (*iterable of Point*) – Points to be updated

Raises KeyError – If any point was not found in the mesh container.

class simphony.io.h5_cuds_items.**H5CUDSItems**(*root*, *record*, *name='items'*)
Bases: _abcoll.MutableMapping

A proxy class to an HDF5 group node with serialised CUDS items.

The class implements the Mutable-Mapping api where each item instance is mapped to uuid.

add_safe(*item*)
Add item while checking for a unique uid.

Note: The item is expected to already have a uid set.

add_unsafe(*item*)
Add item without checking for a unique uid.

Note: The item is expected to already have a uid set.

itersequence(*sequence*)
Iterate over a sequence of row ids.

update_existing(*item*)
Update an item if it already exists.

valid
A PyTables table is opened/created and the object is valid.

class simphony.io.data_container_table.**DataContainerTable**(*root*, *name='data_containers'*, *record=None*)
Bases: _abcoll.MutableMapping

A proxy class to an HDF5 group node with serialised DataContainers.

The class implements the Mutable-Mapping api where each DataContainer instance is mapped to uuid.

append(*data*)
Append the data to the end of the table.

Parameters **data** (DataContainer) – The DataContainer instance to save.

Returns **uid** – The index of the saved row.

Return type uuid.UUID

itersequence(*sequence*)
Iterate over a sequence of row ids.

valid
A PyTables table is opened/created and the object is valid.

class simphony.io.indexed_data_container_table.**IndexedDataContainerTable**(*root*, *name='data_containers'*, *record=None*, *expected_number=None*)
Bases: _abcoll.Sequence

A proxy class to an HDF5 group node with serialised DataContainers.

The class implements the Sequence api where each DataContainer instance is mapped to the row. In addition the class implements update (i.e. __setitem__) and append.

append(*data*)
Append the data to the end of the table.

Parameters **data** (DataContainer) – The DataContainer instance to save.

Returns **index** – The index of the saved row.

Return type int

valid

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

U

V