CONTENTS

1 Introduction
  2 pirs.solids subpackage
    2.1 Simple model ......................................................... 3
    2.2 Assembly-like model ............................................... 4
    2.3 Axial distribution of dependent variables ....................... 6
    2.4 Docstrings .......................................................... 8

3 pirs.mcnp subpackage
  3.1 Materials ............................................................ 30
  3.2 Surfaces and volumes ............................................... 32
  3.3 Cells and models ..................................................... 34
  3.4 Start MCNP ........................................................... 35
  3.5 Docstrings ........................................................... 36

4 pirs.scf2 subpackage
  4.1 Docstrings ........................................................... 47

5 pirs.hli.mcnp subpackage
  5.1 Docstrings ........................................................... 49

6 pirs.hli.scf2 subpackage
  6.1 Docstrings ........................................................... 55

7 pirs.core.scheduler subpackage
  7.1 Jobs and schedulers .................................................. 58
  7.2 Docstrings ........................................................... 58

8 pirs.core.tramat subpackage
  8.1 Nuclides ............................................................... 63
  8.2 Mixtures ..................................................................... 64
  8.3 Data modules ............................................................ 67
  8.4 Docstrings ............................................................... 67

9 pirs.core.trageom subpackage

10 Indices and tables

Python Module Index

Index
PIRS (Python Interfaces for Reactor Simulations) is a package for the Python programming language developed at KIT/INR that contains bindings to nuclear reactor computational codes. This package simplifies workflow with the computational codes by providing a geometry constructor that can be used to describe a model geometry, and interfaces to computational codes.

Currently implemented are interfaces to the MCNP5 Monte-Carlo neutron transport code and to the SCF sub-channel thermo-hydraulics code.

This software was originally developed to provide a framework for coupled neutronics Monte-Carlo – thermo-hydraulics sub-channel calculations for PWR-like geometries. This has defined the choice of codes for which interfaces were implemented in the first place, and type of solids used for geometry construction.

The concept behind PIRS, however, allows to use it not only for organization of coupled calculations. Model description, all necessary calculation parameters and calls to computational codes are specified as a Python script, which make it suitable to e.g. organize parametric studies. Moreover, communication with a code is organized via its input and output files, thus a user not familiar with the code can get examples of valid input files and a tool to process its output.

PIRS is lightweight and simple to install. It was tested with Python versions 2.6 and 2.7 under different linux distributions. Its installation does not require any compilation (i.e. PIRS is pure Python). All its dependencies are optional: PIRS can utilize the uncertainties Python package to handle statistical results of Monte-Carlo calculations, and the Matplotlib package to plot geometries and results of calculations. This allows to install PIRS to a local account on a cluster and use it to run computational codes parallel.

On a linux computer with installed Python, the installation is done with the commands:

```
$> tar -xsf pirs-X.Y.Z.tar.gz
$> cd pirs-X.Y.Z
$> python setup.py install --user
```

where the `--user` command line option in the last command is optional and specifies that the package should be installed to the user’s local account. If the pip Python package manager is used to install PIRS, it will also install the optional uncertainties package.

There are several environment variables that PIRS relies on. The `$MCNP` and `$SCF` variables must show path to executables. Additionally, the path to default xsdir file must be specified in the `$DATAPATH` variable.

The PIRS package consists of several sub-packages that can be classified by their functionality into five groups: geometry constructor, low-level code interfaces, high-level code interfaces, subpackages with base classes used by others, and utilities.

The classes used for geometry construction are described in `pirs.solids subpackage`. They are used to describe model geometry in terms, independent on particular computational code. Thus, the same geometry definition can be used to setup MCNP and SCF models and to hold results of their calculations.
Low-level interfaces give possibility to set any parameter in the code’s input file, to start the code and to read the code’s output file(s). The low-level interfaces for MCNP and SCF are described in `pirs.mcnp subpackage` and `pirs.scf2 subpackage`.

A high-level code interface converts code-independent geometry to a code-specific model representation that can be used to setup code-specific date (like isotopic compositions for MCNP) and to perform respective calculations and to put calculation results back to the code-independent model. The high-level interfaces to MCNP and SCF are described in details in `pirs.hli.mcnp subpackage` and `pirs.hli.scf2 subpackage`.

Classes providing basic functionality that are used in the other subpackages, are described in `pirs.core.scheduler subpackage`, `pirs.core.tramat subpackage` and `pirs.core.trageom subpackage`.
PIRS.SOLIDS SUBPACKAGE

This package provides description of classes representing solids that can be used to describe model geometry. Currently there are Cylinder, Box and Sphere classes.

Model geometry is represented as a set of boxes or cylinders organized in a tree structure. One solid can be inserted into another one (in this case the latter is called a container of the former one), can be positioned arbitrarily within its container (no rotation implemented yet!), and can be partially or completely covered by another solid.

2.1 Simple model

In the following example, a box with two cylinders is described and plotted.

```python
from pirs.solids import Box, Cylinder
from pirs.tools.plots import colormap

b = Box()
b.X = 3
b.Y = 4
b.Z = 5
b.material = 'm1'

c1 = Cylinder()
c1.R = 1
c1.Z = 4
c1.material = 'm2'

b.insert(c1)

c2 = c1.copy_tree()
c2.material = 'm3'
c2.R = 0.8
c2.pos.x = 0.8
c2.pos.y = 0.6

b.insert(c2)

colormap(b, filename='sol1z.png')
colormap(b, plane={'x':0}, filename='sol1x.png')
```

The Box class describes a rectangular parallelepiped with facets perpendicular to the coordinate axes. Attributes Box.X, Box.Y and Box.Z describe dimensions of the box. The material attribute holds the material name (its particular meaning in the computational code must be defined separately in the code’s high-level interface).
Variable \( c_1 \) is a cylinder. It is inserted into \( b \) by the \texttt{insert()} method. The second cylinder, \( c_2 \), is a copy of \( c_1 \), except material and its position in container, represented by \texttt{pos} (an instance of the \texttt{pirs.core.trageom.Vector3} class) are changed. Cylinder \( c_2 \) is also inserted into \( b \).

Cross-sections of the model can be plotted with the help of \texttt{pirs.tools.plots.colormap()} function that takes as argument a solid and returns an instance of the Matplotlib’s \texttt{Axes} class. If optional argument \texttt{filename} is given, the plot will be saved to disk.

![Vertical cross-section](image1)

**Figure 2.1**: Vertical cross-section.

![Horizontal cross-section](image2)

**Figure 2.2**: Horizontal cross-section.

As one can see, cylinder \( c_2 \) partly covers \( c_1 \), while it was inserted into \( b \) after \( c_1 \). At the second plot showing horizontal cross-section, cylinder \( c_2 \) not seen while the cross-section plane, \( x=0 \) does not intersect it.

### 2.2 Assembly-like model

The next example shows how an assembly-like geometry can be modeled:
from pirs.solids import Box, Cylinder
from pirs.tools.plots import colormap

# pin model
pin = Cylinder(R=0.45, Z=100, material='clad')
pin.insert(Cylinder(R=0.4, Z=pin.Z-5, material='fuel'))

# assembly box
a = Box(X=5, Y=7, Z=pin.Z+10, material='water')
a.grid.x = 1
a.grid.y = 1.1
a.grid.z = a.Z

# insert pins
for i in range(5):
    for j in range(6):
        a.grid.insert((i,j,0), pin.copy_tree())

# center grid with respect to solid
a.grid.center()

colormap(a, filename='sol2z.png')
colormap(a, plane={'x':0}, filename='sol2x.png', aspect='auto')

The pin variable is a cylinder containing another, 5 cm shorter and 0.05 cm thinner coaxial cylinder. It represents a pin. Note that attributes of created solids can be set by passing correspondent arguments to the constructor.

Next, the box a is created. Its Box.grid attribute (an instance of the pirs.solids.positions.RGrid class) describes a rectangular grid (lattice) superimposed over the solid, which can be used to position inserted elements. In the example, we set grid pitches along axes using x, y and z attributes of the grid.

In the next loop over i and j indices a copy of pin is inserted into a using the insert() method of the grid. Unlike the Box.insert() method of the solid, the grid’s insert() method takes additional argument – a 3-tuple with indices (i, j, k), which define the grid element where the inserted solid will be positioned. A grid can be shifted as a whole with respect to its solid. By default, the grid is positioned so that the center of the (0,0,0) grid element coincides with the solid’s center. The grid’s origin attribute can be used to set grid position. Alternatively, there is the grid’s center() method that centers the grid in the solid. Note that we have not defined grid dimensions (number of elements in each direction), since they are defined automatically to include all inserted elements.

Figure 2.3: Vertical cross-section.

2.2. Assembly-like model
2.3 Axial distribution of dependent variables

Each solid has `heat`, `dens` and `temp` attributes, that are instances of the `pirs.solids.zmesh` class, representing axial distribution of heat deposition, density and temperature in the solid. All three axial distributions can be specified on independent axial grid.

```python
from pirs.solids import Cylinder
from pirs.tools.plots import colormap

c = Cylinder(Z=2)

# heat
c.heat.set_grid([1, 2, 1])
c.heat.set_values([0.1, 0.2, 0.3])

# temperature
c.temp.set_grid([1]*20)
c.temp.set_values(lambda z: 300 + 100*z)

# density
c.dens.set_grid([1]*5)
c.dens.set_values(1.)

colormap(c, var='heat', plane={'x':0}, filename='sol3h.png')
colormap(c, var='temp', plane={'x':0}, filename='sol3t.png')
colormap(c, var='dens', plane={'x':0}, filename='sol3d.png')
```

To specify the grid for axial distribution, the mesh’ `set_grid()` method is used. It takes a list of scalars which define relative height of axial layers (the first list element corresponds to the lowest axial layer). In the example, the axial grid to represent heat deposition has three axial layers, the middle one is two times thicker as the others. The temperature grid has 20 equal layers, and density grid – 5 layers.

Values of axial distribution are set with the help of the `set_values()` method. It accepts lists (as used for heat), mappings (as used for temperature) or scalars (as used for density).

Axial distribution of heat, temperature or density can be plotted with the `pirs.tools.plots.colormap()` by specifying the `var` argument.
Figure 2.5: Heat.

Figure 2.6: Temperature.

Figure 2.7: Density.
2.4 Docstrings

class pirs.solids.Box(**kwargs)
A box with facets perpendicular to the axes.

Box geometry is defined by three attributes, X, Y and Z, which represent box dimensions.

\( R \)
Radius of circumscribed sphere.

\( X = None \)
Box dimension along x axis

\( Y = None \)
Box dimension along y axis

\( Z = None \)
Box dimension along z axis

\( Zrange(cs='abs') \)
Returns tuple of floats (Zmin, Zmax) – range of self in z coordinate.

\( _\text{append}(\text{othr}) \)
Unconditionally inserts othr to the latest place.
Assumes that othr is not previously inserted

\( \text{abspos}(cs='abs', \text{coordinate}=None) \)
Returns absolute position of the element with respect to the tree’s root.
Optional argument cs (by default ‘abs’) specifies the coordinate system. Can be ‘abs’, ‘rel’, or an instance of the PositionedTree class. In the latter case, it must be a direct or indirect parent of self; the returned position is with respect to this parent.
Optional argument coordinate defines the coordinate’s name returned by the method. By default, coordinate is None and the vector itself is returned. If coordinate is one of ‘x’, ‘y’, ‘z’, ‘r’, etc. (the complete list of variables see in the description of the Vector3 class), the correspondent coordinate is returned.

\( \text{children} \)
An instance of the OrderedDict class where all children of self are stored.

\( \text{circumscribe}(s, \text{adjust_position}=True) \)
Changes inplace properties of the box, so it circumscribes solid s.

\( \text{circumscribed}(s, \text{adjust_position}=True) \)
Returns a new instance of the class self.__class__ that circumscribes s.

\( \text{common_zmesh}(\text{own}=False) \)
Returns an instance of the Mesh() class, which axial boundaries is the union of axial boundaries of all meshes of the solid itself and all its children.

\( \text{covering_sibling}() \)
Returns the younger sibling of the solid or the younger sibling of the solid’s parent(s) that covers completely the solid. If there are no such siblings, returns None.

\( \text{dens} \)
Density axial distribution.

\( \text{denss}() \)
Iterator. Yeilds child elements with density axial distribution defined, recursively.
**extension** \((a, s='abs')\)

Returns tuple \((\text{min}, \text{max})\) representing extension of the solid along axis \(a\).

Argument \(a\) can be ‘x’, ‘y’ or ‘z’.

Optional argument \(s\) specifies the coordinate system. ‘abs’ means the global coordinate system (with respect to the root of self), any other means the local element’s coordinate system.

**get_child** \((k)\)

Returns the node with compound or local key \(k\).

The compound key is relative to the instance the method bound to. Compare with the result of get_key() method.

If \(k\) is not a local key, can be indexed but has no 0-th element, the self is returned. For example, if \(k\) is an empty tuple or an empty string.

**get_key**

Returns the compound key of the node relative to the tree root.

```python
>>> t = Tree()
>>> t.insert(1, Tree()).insert(2, Tree())
>>> t.insert(3, Tree())
>>> for n in t.values():
...     print n.get_key()
...
(1,)
(1, 2)
(3,)
```

**get_parents** \((\text{reverse}=False)\)

Returns a list of all node parents, starting from the direct parent and ending with the tree root.

The optional boolean argument reverse changes the order of parents in the resulting list.

```python
>>> t = Tree()
>>> t.insert(1, Tree()).insert(2, Tree())
>>> t.insert(3, Tree())
>>> for n in t.values():
...     print 'for ', n.get_key(), ', parents are: ', map(lambda x:x.get_key(), n.get_parents())
...
for (1,) parents are: [()]
for (1, 2) parents are: [(1,), ()]
for (3,) parents are: [()]
```

**get_radius** \((\text{inscribed}=True)\)

Returns radius of circumscribed sphere. Deprecated, use circumscribe method.

**get_siblings**

Returns a tuple of two lists with nodes inserted into the node’s parent before and after the node itself.

For example, \(n0\) contains 4 nodes:

```python
>>> n0 = Tree()
>>> n1 = n0.insert(1, Tree())
>>> n2 = n0.insert(2, Tree())
>>> n3 = n0.insert(3, Tree())
>>> n4 = n0.insert(4, Tree())

>>> n2.get_siblings() == ([n1], [n3, n4])
True
>>> n1.get_siblings() == ([], [n2, n3, n4])
```
get_value_by_coord(var, xyz, cs='rel')
Returns value of axial distribution specified by var at position given by the xyz tuple.
Optional argument cs specifies whether xyz given in absolute ('abs') or relative ('rel') coordinate system.

get_value_by_index(var, i)
Returns i-th value in the axial distribution specified by var.
If axial distribution is not defined for the solid yet, return default value.

grid
Grid description. Its parameters are used to compute absolute position of children, if they have other than None i,j,k attributes.

has_var(name)
Return True, if axial distribution name (can be 'temp', 'heat' or 'dens') is defined for the solid.

heat
Heat axial distribution.

heats()
Iterator. Yeilds child elements with heat axial distribution, recursively.

hiding_parent()
Returns the container that completely hides the solid. If there are no such parent, returns None.

i
Index to position solid in the parent’s grid along x axis.

id()
Returns id(self).

ijk
Tuple with element indices specifying the parent’s grid element where the element is positioned.

indexed()
Returns True if self is positioned using the indices i, j and k.

insert(othr, i=None)
Insert element othr into self.
Optional argument i specifies index, where othr should be inserted. By default, othr is inserted as the last (most recent) child. One can specify i to set order fo the inserted element in the list of previously inserted.

intersect(othr)
Returns True if self positioned at self.abspos() intersects (i.e. has common points) with othr positioned at othr.abspos().

is_constant(var)
Checks if the axial distribution var is constant.

is_visible()
Checks if the solid is seen from its parent(s), and is not completely covered by the younger siblings. Only in this case the element and its interior can be “seen” in the model. Otherwise, it can be removed from the model tree, without actually changing the model.
Note that the element can also be covered by its child. In this case, however, its children (at least one that covers) are still visible and cannot be removed.

**items** *(selfInclusive=False)*

Returns a list of tuples (ckey, node) for all children in self recursively, where ckey is a compound key and node is the correspondent Tree instance.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by default, it is not included).

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.items()
[((1,), <__main__.Tree object at ...>), ((2,), <__main__.Tree object at ...>), ((2, 21), <...>)]
```

**j**

Index to position solid in the parent’s grid along x axis.

**k**

Index to position solid in the parent’s grid along x axis.

**keys()**

Returns a list of compound keys of all children in self recursively.

The order is depth-first.

The local key specified in the insert() method refers only to the direct child. To refer to a node inserted into a child node, the compound key is used. The compound key is a tuple of local keys.

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.insert(3, Tree()).insert(31, Tree()).insert(311, Tree())
>>> t.keys()
[(1,), (2,), (2, 21), (3,), (3, 31), (3, 31, 311)]
```

**lattice_elements()**

Returns list of boxes that represent lattice elements of the solid.

Returned values are tuples of the form (ijk, itype, Box()), where ijk is (i,j,k), itype is the type

**layers** *(temp=True, dens=True, heat=True)*

Returns iterator with elements describing each layer.

Each returned element is a tuple (Zmin, Zmax, values, children, (is_first, is_last))

**lies_in** *(othr)*

Returns True, if self lies completely inside othr.

**local_index**

The position of element in its parent.

**material**

Material name. Any immutable, e.g. integer or string.

**max** *(param='heat', filter_=<function <lambda> at 0x2b9e7d7bfa28>)*

Returns a tuple (v, key) where v is the maximal value of parameter param found in the solid itself and its children. Searched only elements that pass the `filter_` functions, i.e if filter_(element) returns True.
parent
If self is inserted into another tree element, parent points to this element.

parents (last=None)
Iterates over all parents of self, starting from the direct parent till the root.
If optional argument last is given, iterates until this parent, not until the root.

pos
Position of the element with respect to its parent. By default at the origin.

classmethod properties ()
Returns a list of property names defined in the class, including all inherited properties.

classmethod random_tree (N=20, seed=None)
Returns a random tree with N nodes.
Algorithm: until the number of nodes in the tree, Ne, less than N, sample an integer from [1..Ne] and
insert a new node as a child into the node with sampled number.
Note, this is a class method.

remove (obj)
Removes child with local key lkey from the node. The removed element is returned.
Note that this method provides functionality similar to the functionality of the withdraw method. The
difference is that remove() removes child from self, and withdraw removes self from its parent.

remove_by_criteria (**kwargs)
Removes direct children if they meet all criteria specified by kwargs. For example,
t.remove_by_criteria(name='fuel', i=3)
will remove from t all children with name set to ‘fuel’ and positioned in the grid with i index equal to 3.
Returns the list of removed elements.

remove_by_index (i)
Removes i-th child.

remove_child (element)
Removes child from self.

remove_invisible ()
Removes all invisible children of the element recursively.
The element itself remains in the model even if its is_visible() method returns False.

root
Link to the root element of the tree self belongs to.

setup (**kwargs)
Set attributes and properties specified in the keyword arguments.
If an attribute or a property does not exist, the AttributeError is raised.

shift_children (i, N, inew)
Shift children [i:i+N] to [inew:inew+N]

str_node (attr_=None)
Returns the string representing the node properties.
The optional argument attr_ specifies the name of an attribute to print out. Can be a list of attribute names.
```python
>>> t = Tree()
>>> t.str_node() == t.str_node('id()')  # by default, t.str_node() returns t.id()
True
>>> t.str_node('parent')  # prints out the value of the parent attribute.
"Tree 'parent': None"

str_tree (attr=None)
Returns a string representing the (sub)tree structure of the node and its children.

The optional argument attr defines what will be printed. Its meaning and default see in the description
of the str_node() method.

```}

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree())
>>> t.get_child(1).insert(3, Tree())
>>> t.insert(4, Tree())
>>> print t.str_tree()
Tree 'id()': ...
  - key 1: Tree 'id()': ...
  - key 3: Tree 'id()': ...
  - key 2: Tree 'id()': ...
  - key 4: Tree 'id()': ...
```

```python
style
Returns type of the solid.

temp
Temperature axial distribution.

temps ()
Iterator. Yields child elements with temperature axial distribution defined, recursively.

values (selfInclusive=False)
Returns a list of all children in self recursively.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by
default, it is not included).

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.values()
[<__main__.Tree object at ...>, <__main__.Tree object at ...>, <__main__.Tree object at ...>]
```}

withdraw ()
Removes self from its parent.

Method returns self.

class pirs.solids.Cylinder (**kwargs)
A finite-height cylinder with the axis parallel to the z axis. Cylinder geometry is defined by attributes R and Z.

X
Cylinder’s dimension along X axis.

Y
Cylinder’s dimension along Y axis.
Zrange (cs='abs')
Returns tuple of floats (Zmin, Zmax) – range of self in z coordinate.

__append__(othr)
Unconditionally inserts othr to the latest place.
Assumes that othr is not previously inserted

abspos (cs='abs', coordinate=None)
Returns absolute position of the element with respect to the tree’s root.
Optional argument cs (by default ‘abs’) specifies the coordinate system. Can be ‘abs’, ‘rel’, or an instance of the PositionedTree class. In the latter case, it must be a direct or indirect parent of self; the returned position is with respect to this parent.
Optional argument coordinate defines the coordinate’s name returned by the method. By default, coordinate is None and the vector itself is returned. If coordinate is one of ‘x’, ‘y’, ‘z’, ‘r’, etc. (the complete list of variables see in the description of the Vector3 class), the correspondent coordinate is returned.

children
An instance of the OrderedDict class where all children of self are stored.

circumscribe (s, adjust_position=True)
Change inplace properties of the cylinder so that it circumscribes solid s.

circumscribed (s, adjust_position=True)
Returns a new instance of the class self.__class__ that circumscribes s.

common_zmesh (own=False)
Returns an instance of the Mesh() class, which axial boundaries is the union of axial boundaries of all meshes of the solid itself and all its children.

covering_sibling()
Returns the younger sibling of the solid or the younger sibling of the solid’s parent(s) that covers completely the solid. If there are no such siblings, returns None.

dens
Density axial distribution.

denss()
Iterator. Yields child elements with density axial distribution defined, recursively.

extension (a, s='abs')
Returns tuple (min, max) representing extension of the solid along axis a.
Argument a can be ‘x’, ‘y’ or ‘z’.
Optional argument s specifies the coordinate system. ‘abs’ means the gloabal coordinate system (with respect to the root of self), any other means the local element’s coordinate system.

get_child (k)
Returns the node with compound or local key k.
The compound key is relative to the instance the method bound to. Compare with the result of get_key() method.
If k is not a local key, can be indexed but has no 0-th element, the self is returned. For example, if k is an empty tuple or an emptyly string.

get_key()
Returns the compound key of the node relative to the tree root.
>>> t = Tree()
>>> t.insert(1, Tree()).insert(2, Tree())
>>> t.insert(3, Tree())
>>> for n in t.values():
...    print n.get_key()
...
(1,)
(1, 2)
(3,)

def get_parents(reverse=False):
    """Returns a list of all node parents, starting from the direct parent and ending with the tree root."
    """The optional boolean argument reverse changes the order of parents in the resulting list.
    
    >>> t = Tree()
    >>> t.insert(1, Tree()).insert(2, Tree())
    >>> t.insert(3, Tree())
    >>> for n in t.values():
    ...    print 'for ', n.get_key(), ' parents are: ', map(lambda x:x.get_key(), n.get_parents())
    ...
    for (1,) parents are: [()]
    for (1, 2) parents are: [(1,), ()]
    for (3,) parents are: [()]

def get_radius(inscribed=True):
    """Returns radius of circumscribed sphere. Deprecated, use circumscribe method."""

def get_siblings():
    """Returns a tuple of two lists with nodes inserted into the node's parent before and after the node itself."
    
    For example, n0 contains 4 nodes:
    
    >>> n0 = Tree()
    >>> n1 = n0.insert(1, Tree())
    >>> n2 = n0.insert(2, Tree())
    >>> n3 = n0.insert(3, Tree())
    >>> n4 = n0.insert(4, Tree())
    >>> n2.get_siblings() == ([n1], [n3, n4])
    True
    >>> n1.get_siblings() == ([], [n2, n3, n4])
    True
    >>> n4.get_siblings() == ([n1, n2, n3], [])
    True

def get_value_by_coord(var, xyz, cs='rel'):
    """Returns value of axial distribution specified by var at position given by the xyz tuple."
    """Optional argument cs specifies whether xyz given in absolute ('abs') or relative ('rel') coordinate system.

def get_value_by_index(var, i):
    """Returns i-th value in the axial distribution specified by var. If axial distribution is not defined for the solid yet, return default value."""

grid
    """Grid description. Its parameters are used to compute absolute position of children, if they have other than None i,j,k attributes."""
has_var

Return True, if axial distribution name (can be 'temp', 'heat' or 'dens') is defined for the solid.

heat

Heat axial distribution.

heats()

Iterator. Yields child elements with heat axial distribution, recursively.

hiding_parent()

Returns the container that completely hides the solid. If there are no such parent, returns None.

i

Index to position solid in the parent’s grid along x axis.

id()

Returns id(self).

ijk

Tuple with element indices specifying the parent’s grid element where the element is positioned.

indexed()

Returns True if self is positioned using the indices i, j and k.

insert(other, i=None)

Insert element other into self.

Optional argument i specifies index, where other should be inserted. By default, other is inserted as the last (most recent) child. One can specify i to set order for the inserted element in the list of previously inserted.

intersect(other)

Returns True if self positioned at self.abspos() intersects (i.e. has common points) with other positioned at other.abspos().

is_constant(var)

Checks if the axial distribution var is constant.

is_visible()

Checks if the solid is seen from its parent(s), and is not completely covered by the younger siblings. Only in this case the element and its interior can be “seen” in the model. Otherwise, it can be removed from the model tree, without actually changing the model.

Note that the element can also be covered by its child. In this case, however, its children (at least one that covers) are still visible and cannot be removed.

items(selfInclusive=False)

Returns a list of tuples (ckey, node) for all children in self recursively, where ckey is a compound key and node is the correspondent Tree instance.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by default, it is not included).

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.items()
[((1,), <__main__.Tree object at ...>), ((2,), <__main__.Tree object at ...>), ((2, 21), <...>)]
```

j

Index to position solid in the parent’s grid along x axis.
Index to position solid in the parent’s grid along x axis.

**keys**

Returns a list of compound keys of all children in self recursively.

The order is depth-first.

The local key specified in the insert() method refers only to the direct child. To refer to a node inserted into a child node, the compound key is used. The compound key is a tuple of local keys.

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.insert(3, Tree()).insert(31, Tree()).insert(311, Tree())
>>> t.keys()
[(1,), (2,), (2, 21), (3,), (3, 31), (3, 31, 311)]
```

**lattice_elements**

Returns list of boxes that represent lattice elements of the solid.

Returned values are tuples of the form (ijk, itype, Box()), where ijk is (i,j,k), itype is the type

**layers** *(temp=True, dens=True, heat=True)*

Returns iterator with elements describing each layer.

Each returned element is a tuple (Zmin, Zmax, values, children, (is_first, is_last))

**lies_in**(othr)

Returns True, if self lies completely inside othr.

**local_index**

The position of element in its parent.

**material**

Material name. Any immutable, e.g. integer or string.

**max**(param='heat', filter_=<function <lambda> at 0x2b9e7d7d7fa28>)

Returns a tuple (v, key) where v is the maximal value of parameter param found in the solid itself and its children. Searched only elements that pass the filter_ functions, i.e if filter_(element) returns True.

**parent**

If self is inserted into another tree element, parent points to this element.

**parents**(last=None)

Iterates over all parents of self, starting from the direct parent till the root.

If optional argument last is given, iterates untill this parent, not untill the root.

**pos**

Position of the element with respect to its parent. By default at the origin.

**classmethod** **properties**

Returns a list of property names defined in the class, including all inherited properties.

**classmethod** **random_tree**(N=20, seed=None)

Returns a random tree with N nodes.

Algorithm: untill the number of nodes in the tree, Ne, less than N, sample an integer from [1..Ne] and insert a new node as a child into the node with sampled number.

Note, this is a class method.
remove(obj)
Removes child with local key lkey from the node. The removed element is returned.

Note that this method provides functionality similar to the functionality of the withdraw method. The difference is that remove() removes child from self, and withdraw removes self from its parent.

remove_by_criteria(**kwargs)
Removes direct children if they meet all criteria specified by kwargs. For example,

```python
t.remove_by_criteria(name='fuel', i=3)
```

will remove from t all children with name set to ‘fuel’ and positioned in the grid with i index equal to 3.

Returns the list of removed elements.

remove_by_index(i)
Removes i-th child.

remove_child(element)
Removes child from self.

remove_invisible()
Removes all invisible children of the element recursively.

The element itself remains in the model even if its is_visible() method returns False.

root
Link to the root element of the tree self belongs to.

setp(**kwargs)
Set attributes and properties specified in the keyword arguments.

If an attribute or a property does not exist, the AttributeError is raised.

shift_children(i, N, inew)
Shift children [i:i+N] to [inew:inew+N]

str_node(attr_=None)
Returns the string representing the node properties.

The optional argument attr_ specifies the name of an attribute to print out. Can be a list of attribute names.

```python
>>> t = Tree()
>>> t.str_node() == t.str_node('id()')  # by default, t.str_node() returns t.id()
True
>>> t.str_node('parent')  # prints out the value of the parent attribute.
"Tree ‘parent’: None"
```

str_tree(attr_=None)
Returns a string representing the (sub)tree structure of the node and its children.

The optional argument attr_ defines what will be printed. Its meaning and default see in the description of the str_node() method.

```python
>>> t = Tree()
>>> t.inserT( 1, Tree())
>>> t.inserT( 2, Tree())
>>> t.get_child(1).inserT( 3, Tree())
>>> t.inserT( 4, Tree())
```
- key 2: Tree 'id()': ...
- key 4: Tree 'id()': ...

**stype**

Returns type of the solid.

**temp**

Temperature axial distribution.

**temps()**

Iterator. Yeilds child elements with temperature axial distribution defined, recursively.

**values(selfInclusive=False)**

Returns a list of all children in self recursively.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by default, it is not included).

```
>>> t = Tree()
>>> t.inserT(1, Tree())
>>> t.insert(2, Tree()).inset(21, Tree())
>>> t.values()
[<__main__.Tree object at ...>, <__main__.Tree object at ...>, <__main__.Tree object at ...>]
```

**withdraw()**

Removes self from its parent.

Method returns self.

```python
class pirs.solids.Sphere(**kwargs)
A sphere, which radius is defined by the attribute R.

X

Sphere’s dimension along axis X.

Y

Sphere’s dimension along axis Y.

Z

Sphere’s dimension along axis Z.

Zrange(cs='abs')

Returns tuple of floats (Zmin, Zmax) – range of self in z coordinate.

__append(othr)

Unconditionally inserts othr to the latest place.

Assumes that othr is not previously inserted

abspos(cs='abs', coordinate=None)

Returns absolute position of the element with respect to the tree’s root.

Optional argument cs (by default ‘abs’) specifies the coordinate system. Can be ‘abs’, ‘rel’, or an instance of the PositionedTree class. In the latter case, it must be a direct or indirect parent of self; the returned position is with respect to this parent.

Optional argument coordinate defines the coordinate’s name returned by the method. By default, coordinate is None and the vector itself is returned. If coordinate is one of ‘x’, ‘y’, ‘z’, ‘r’, etc. (the complete list of variables see in the description of the Vector3 class), the correspondent coordinate is returned.
children
An instance of the OrderedDict class where all children of self are stored.

circumscribe (s, adjust_position=True)
Changes in-place parameters of the sphere, so that it circumscribes solid s.

circumscribed (s, adjust_position=True)
Returns a new instance of the class self.__class__ that circumscribes s.

common_zmesh (own=False)
Returns an instance of the Mesh() class, which axial boundaries is the union of axial boundaries of all
meshes of the solid itself and all its children.

covering_sibling ()
Returns the younger sibling of the solid or the younger sibling of the solid’s parent(s) that covers com-
pletely the solid. If there are no such siblings, returns None.

dens
Density axial distribution.

denss ()
Iterator. Yeilds child elements with density axial distribution defined, recursively.

extension (a, s='abs')
Returns tuple (min, max) representing extension of the solid along axis a.
Argument a can be ‘x’, ‘y’ or ‘z’.
Optional argument s specifies the coordinate system. ‘abs’ means the global coordinate system (with
respect to the root of self), any other means the local element’s coordinate system.

get_child (k)
Returns the node with compound or local key k.
The compound key is relative to the instance the method bound to. Compare with the result of get_key() method.
If k is not a local key, can be indexed but has no 0-th element, the self is returned. For example, if k is an
empty tuple or an empty string.

get_key ()
Returns the compound key of the node relative to the tree root.

>>> t = Tree()
>>> t.insert(1, Tree()).insert(2, Tree())
>>> t.insert(3, Tree())
>>> for n in t.values():
...    print n.get_key()
...
(1,)
(1, 2)
(3,)

get_parents (reverse=False)
Returns a list of all node parents, starting from the direct parent and ending with the tree root.
The optional boolean argument reverse changes the order of parents in the resulting list.

>>> t = Tree()
>>> t.insert(1, Tree()).insert(2, Tree())
>>> t.insert(3, Tree())
>>> for n in t.values():
...    print ‘for ‘, n.get_key(), ‘ parents are: ‘, map(lambda x:x.get_key(), n.get_parents())
for (1,) parents are: [()]
for (1, 2) parents are: [(1,), ()]
for (3,) parents are: [()]

get_radius (inscribed=True)
Returns radius of circumscribed sphere. Deprecated, use circumscribe method.

get_siblings ()
Returns a tuple of two lists with nodes inserted into the node’s parent before and after the node itself.
For example, n0 contains 4 nodes:

>>> n0 = Tree()
>>> n1 = n0.insert(1, Tree())
>>> n2 = n0.insert(2, Tree())
>>> n3 = n0.insert(3, Tree())
>>> n4 = n0.insert(4, Tree())

>>> n2.get_siblings() == ([n1], [n3, n4])
True
>>> n1.get_siblings() == ([], [n2, n3, n4])
True
>>> n4.get_siblings() == ([n1, n2, n3], [])
True

get_value_by_coord (var, xyz, cs='rel')
Returns value of axial distribution specified by var at position given by the xyz tuple.
Optional argument cs specifies whether xyz given in absolute ('abs') or relative ('rel') coordinate system.

get_value_by_index (var, i)
Returns i-th value in the axial distribution specified by var.
If axial distribution is not defined for the solid yet, return default value.

grid
Grid description. Its parameters are used to compute absolute position of children, if they have other than None i,j,k attributes.

has_var (name)
Return True, if axial distribution name (can be 'temp', 'heat' or 'dens') is defined for the solid.

heat
Heat axial distribution.

heats ()
Iterator. Yeilds child elements with heat axial distribution, recursively.

hiding_parent ()
Returns the container that completely hides the solid. If there are no such parent, returns None.

i
Index to position solid in the parent’s grid along x axis.

id ()
Returns id(self).

ijk
Tuple with element indices spcifying the parent’s grid element where the element is positioned.
indexed()
Returns True if self is positioned using the indices i, j and k.

insert (othr, i=None)
Insert element othr into self.
Optional argument i specifies index, where othr should be inserted. By default, othr is inserted as the last (most recent) child. One can specify i to set order of the inserted element in the list of previously inserted.

intersect (othr)
Returns True if self positioned at self.abspos() intersects (i.e. has common points) with othr positioned at othr.abspos().

is_constant (var)
Checks if the axial distribution var is constant.

is_visible ()
Checks if the solid is seen from its parent(s), and is not completely covered by the younger siblings. Only in this case the element and its interior can be “seen” in the model. Otherwise, it can be removed from the model tree, without actually changing the model.

Note that the element can also be covered by its child. In this case, however, its children (at least one that covers) are still visible and cannot be removed.

items (selfInclusive=False)
Returns a list of tuples (ckey, node) for all children in self recursively, where ckey is a compound key and node is the correspondent Tree instance.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by default, it is not included).

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.items()
[((1,), <__main__.Tree object at ...>), ((2,), <__main__.Tree object at ...>), ((2, 21), <...>)]
```

j
Index to position solid in the parent’s grid along x axis.

k
Index to position solid in the parent’s grid along x axis.

keys ()
Returns a list of compound keys of all children in self recursively.

The order is depth-first.

The local key specified in the insert() method refers only to the direct child. To refer to a node inserted into a child node, the compound key is used. The compound key is a tuple of local keys.

```python
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.insert(3, Tree()).insert(31, Tree()).insert(311, Tree())
>>> t.keys()
[(1,), (2,), (2, 21), (3,), (3, 31), (3, 31, 311)]
```
lattice_elements()
Returns list of boxes that represent lattice elements of the solid.

Returned values are tuples of the form (ijk, itype, Box()), where ijk is (i,j,k), itype is the type

layers (temp=True, dens=True, heat=True)
Returns iterator with elements describing each layer.
Each returned element is a tuple (Zmin, Zmax, values, children, (is_first, is_last))

lies_in(othr)
Returns True, if self lies completely inside othr.

local_index
The position of element in its parent.

material
Material name. Any immutable, e.g. integer or string.

max (param='heat', filter_=<function <lambda> at 0x2b9e7d7bfa28>)
Returns a tuple (v, key) where v is the maximal value of parameter param found in the solid itself and its
children. Searched only elements that pass the filter_ functions, i.e if filter_(element) returns True.

parent
If self is inserted into another tree element, parent points to this element.

parents (last=None)
Iterates over all parents of self, starting from the direct parent till the root.
If optional argument last is given, iterates untill this parent, not untill the root.

pos
Position of the element with respect to its parent. By default at the origin.

classmethod properties()
Returns a list of property names defined in the class, including all inherited properties.

classmethod random_tree (N=20, seed=None)
Returns a random tree with N nodes.
Algorithm: untill the number of nodes in the tree, Ne, less than N, sample an integer from [1..Ne] and
insert a new node as a child into the node with sampled number.

Note, this is a class method.

remove (obj)
Removes child with local key lkey from the node. The removed element is returned.
Note that this method provides functionality similar to the functionality of the withdraw method. The
difference is that remove() removes child from self, and withdraw removes self from its parent.

remove_by_criteria (**kwargs)
Removes direct children if they meet all criteria specified by kwargs. For example,
t.remove_by_criteria(name='fuel', i=3)
will remove from t all children with name set to ‘fuel’ and positioned in the grid with i index equal to 3.

Returns the list of removed elements.

remove_by_index (i)
Removes i-th child.

remove_child (element)
Removes child from self.
**remove_invisible()**
Removes all invisible children of the element recursively.

The element itself remains in the model even if its is_visible() method returns False.

**root**
Link to the root element of the tree self belongs to.

**setup(** **kwargs **)**
Set attributes and properties specified in the keyword arguments.

If an attribute or a property does not exist, the AttributeError is raised.

**shift_children**(i, N, inew)
Shift children [i:i+N] to [inew:inew+N]

**str_node**(attr_=None)
Returns the string representing the node properties.

The optional argument attr_ specifies the name of an attribute to print out. Can be a list of attribute names.

```
>>> t = Tree()
>>> t.str_node() == t.str_node('id()')  # by default, t.str_node() returns t.id()
True
>>> t.str_node('parent')  # prints out the value of the parent attribute.
"Tree ‘parent’: None"
```

**str_tree**(attr_=None)
Returns a string representing the (sub)tree structure of the node and its children.

The optional argument attr_ defines what will be printed. Its meaning and default see in the description of the str_node() method.

```
>>> t = Tree()
>>> t.inserT( 1, Tree())
>>> t.inserT( 2, Tree())
>>> t.get_child(1).inserT( 3, Tree())
>>> t.inserT( 4, Tree())
>>> print t.str_tree()
Tree ‘id()’: ...
  - key 1: Tree ‘id()’: ...
    - key 3: Tree ‘id()’: ...
  - key 2: Tree ‘id()’: ...
  - key 4: Tree ‘id()’: ...
```

**stype**
Returns type of the solid.

**temp**
Temperature axial distribution.

**temps()**
Iterator. Yeilds child elements with temperature axial distribution defined, recursively.

**values**(selfInclusive=False)
Returns a list of all children in self recursively.

The order is depth-first.

If the optional argument selfInclusive is True, the first element of the returned list is the node itself (by default, it is not included).
>>> t = Tree()
>>> t.insert(1, Tree())
>>> t.insert(2, Tree()).insert(21, Tree())
>>> t.values()
[<__main__.Tree object at ...>, <__main__.Tree object at ...>, <__main__.Tree object at ...>]

withdraw()
Removes self from its parent.
Method returns self.

class pirs.solids.positions.RGrid(container, x=1, y=1, z=1, origin=(0, 0, 0))

boundaries (d='x')
Returns coordinates of boundaries in direction d with respect to the grid’s container.

center (log=False)
Positions the central element of the grid so that the box circumscribing all inserted grid elements is centered
with respect to the container.

container
Link to the solid containing the grid.

elements()
Iterates over index-positioned elements.

extension (a=None)
Returns tuple (Imin, Imax) of minimal and maximal grid element indices in the direction along axis a.
The argument a can be ‘x’, ‘y’ or ‘z’.
UPD: the default a is None; in this case the tuple (Imin, Imax, Jmin, Jmax, Kmin, Kmax) is returned.

index (x, y, z)
Returns index i,j,k of the element containing point p (with respect to the grid’s container)
index(x=5, y=7, z=8) # returns tuple (i, j, k)

insert (ijk, element, i=None)
Inserts element into the grid’s container and specifies for the inserted element that it should be positioned
with respect to the ijk-th grid element.

origin
Position of the grid’s central element (0,0,0) with respect to the container.

position (i, j, k, coordinate=None)
Returns position of element ijk with respect to the grid’s container.

set_origin ((i, j, k), (x, y, z))
Sets the grid origin so that the grid element (i,j,k) has position (x, y, z) with respect to the grid’s container.

used()
Returns True if at least one of the grid container’s local children has not None i,j,k attributes.

x
Grid pitch along x axis.

y
Grid pitch along y axis.

z
Grid pitch along z axis.

2.4. Docstrings
class `pirs.solids.zmesh (boundary)`

Class to represent axial mesh for density, temperature and heat in a solid with Z dimension.

An axial mesh is defined by giving a solid (a reference solid) and by giving relative height of mesh elements (a relative grid).

When a new instance of `zmesh` is created, an instance of one of the solids must be provided, which will be used to define the absolute height of the mesh elements.

```
b = Box()  # default box with X,Y,Z = 1 m = zmesh(b)  # axial mesh m with b as the reference solid.
```

Length of mesh elements is specified in relative units. For example,
```
  m.set_grid([0.5, 1., 0.5])
```

sets number of mesh elements and their lengths. The argument is a list, which i-th element gives the relative length of the i-th mesh element along z axis. The absolute length is obtained first by normalizing the values in the list, so their sum gives one, and by multiplying by the dimension of the reference solid. For the above example, mesh elements along z will be 0.25 cm, 0.5 cm and 0.25 cm.

Initialize the axial mesh. Boundary must be an object with attribute Z, which defines the absolute height of the mesh (this is, for example, instance of `Box` class).

Initially, there is only one mesh element, and value is set to 0.

```
  adjust_grid (Nmax, dVmin, alpha=0.3333333333333333)
```

Changes the grid by inserting new mesh elements between elements with maximal dV, and by combining elements with dV less than dVmin.

```
b = solids.Box()
b.pos.z = 6
m = zmesh(b)
m.boundary_coords('abs')
[5.5, 6.5]
m.boundary_coords('rel')
[-0.5, 0.5]
m.set_grid([1.]*4)
m.boundary_coords('rel')
[-0.5, -0.25, 0.0, 0.25, 0.5]
m.boundary_coords('abs')
[5.5, 5.75, 6.0, 6.25, 6.5]
```

```
clear()
```

Set grid so that the mesh has only one element and set the value to 0.

```
common_grid (othr)
```

Deprecated. Use `unify()`.

```
convert (type_=<type 'float'>)
```

Converts values saved in `zmesh` to `type_`. Argument `type_` must be a function.

```
copy (boundary=None)
```

Return copy of self.

```
crop (othr)
```

Put to self data from `othr`, so that mesh elements coincide.
**element_coord** 

(\(k=0, \text{cs}=\text{'rel'}\))

Returns coordinates of \(k\)-th mesh element’s center.

The optional argument \(\text{cs}\) accepts the following values:

• 'rel' (default): the returned coordinates are in the coordinate system of the reference solid.
• 'abs': the returned coordinates are in the coordinate system of the root of the reference solid.
• '1': the returned coordinates are relative to the c.s. of the reference solid whose height is set to 1.

Between \(\text{rel}\), \(\text{abs}\) and \(\text{1}\) coordinates hold the following equalities:

\[ Z_{\text{abs}} = Z_{\text{rel}} - \text{self.__b.abspos()} \]
\[ Z_{\text{rel}} = Z_{\text{1}} \times \text{self.__b.Z} \]

**element_coords**

(cs='rel')

Returns the list of mesh-elements center coordinates.

**element_index**

(z=0.0, cs='rel')

Returns the index of the mesh element containing coordinate \(z\), relative or absolute.

**get_grid**

(rel=True)

Returns the list of mesh elements lengths

**get_max**

(func=None)

Returns \((V_{\text{max}}, i)\) tuple, where \(V_{\text{max}}\) is the maximal value, and \(i\) –its index.

When \(\text{func}\) is given, maximum is searched among \(\text{func}(vi)\).

**get_solid**

Returns the reference solid.

**get_value_by_coord**

(xyz, cs='rel')

Returns the value of mesh element, specified by the xyz coordinate.

The value of the mesh element covering point with axial coordinate \(z\) is returned.

**get_value_by_index**

(k)

Returns the value of mesh element specified by its index

**has_zeros**

Returns True if \text{self.values()} has one or more zeroes.

**integral**

(A=None, B=None, cs='rel')

Returns integral from \(A\) to \(B\) of the piecewise-constant function.

\(\text{cs}\) defines the meaning of \(A\) and \(B\). Can be ‘\text{rel}’, ‘\text{abs}’ and ‘\text{1}’.

```python
>>> m = zmesh(solids.Box(Z=4.))
>>> m.set_grid([1]*4)
>>> m.set_values([1, 2, 3, 4])
>>> Zlst = [-4.] + map(float, m.boundary_coords('rel')) + map(lambda x: x[2], m.element_coords('rel')) + [4.]
>>> Zlst.sort()
>>> for A in Zlst:
...     for B in Zlst:
...         print '{0:7.3f} {1:7.3f} {2:9.5f}'.format(A, B, m.integral(A, B, 'rel'))
...```
interpolate \((z, \text{cs}='rel')\)

Returns interpolated value at coordinate \(z\).

```python
>>> m = zmesh(solids.Box())
>>> m.set_grid([1]*5)
>>> m.set_values([1, 2, 3, 4, 5])
>>> for (x,y,z) in m.element_coords('1'):
...    print z, m.interpolate(z, '1')
...```

```python
>>> for z in m.boundary_coords('1'):
...    print z, m.interpolate(z, '1')
...```

is_constant()

Returns True if all values of the mesh are equal.

items(\text{key_type}='index', \text{cs}='rel')

Returns a list of tuples \((k, \text{val})\) in the order described in method set_values().

If key_type is 'index' (default), \(k\) is the mesh element index. If key_type is 'coord', \(k\) is the mesh center's coordinates, \(k = (x,y,z)\). In this case one can additionally specify coordinate system, 'rel' or 'abs'

set_grid(lst=[1.0])

Set relative grid.

\(lst\) is a list specifying the number of grid elements and their relative length.

set_value_by_coord(val, z, \text{cs}='rel')

Set value to the mesh element, specified by its \(z\) coordinate, relative or absolute.

The value set to the mesh element, which covers the given coordinate.

set_value_by_index(val, k)

Set value to mesh element specified by its index.

Index is an integer. Counting starts from zero.

set_values(val, \text{cs}='rel')

Set values of the mesh.

Accepted types of \(val\) are:

- a list or a tuple. Must have the same number of elements as the list returned by the get_grid() method.

- a mapping (function) used to calculate value at each mesh element center. The meaning of the mapping's argument can be set by the method's optional argument \(cs\), its meaning see in element_coords() method.

- another instance of the zmesh class. In this case, grid and values of this instance are copied to self. This is equal to:

  ```python
  self.update(othr)
  ```

- if not one of the above, transformed to the list \([val]\)*len(self.get_grid())

set_values_by_function(f, \text{cs}='rel')

Set values of the mesh by function \(f(z): z \rightarrow f\).

Value of each mesh element is set to \(f(z)\), where \(z\) is coordinate of the mesh element's center.
simplify()
Joins adjacent mesh elements if their values are within the precision prec.

unify(othr, log=False)
Changes self and othr so that their mesh boundaries coincide.

values()
Returns the list of values in the order described in method set_values()

pirs.tools.plots.colormap(model, plane={'z': 0}, axis=None, var=None, filter_=None, aspect='equal', colors=None, nmarker={}, mmarker={}, legend=True, filename=None, **kwargs)
Returns an instance of the matplotlib Axes class containing colormap that shows distribution of the system variable var on the cross-section plane defined by the argument plane.

Parameters
- **model** – Model to be plotted. Instance of Box, Cylinder or Sphere class.
- **plane** (dict) – Dictionary describing the cut plane.
- **axis** – Instance of the matplotlib.axes.Axes class where geometry plot will be added. If not specified, a new instance will be created.
- **var** (str) – Variable name, one of ‘material’, ‘temp’, ‘heat’ or ‘dens’ which will be used to color the geometry.
- **filter** (func) – boolean function taking a solid as argument. A solid will be plotted only if this function returns True for it. By default, all solids are plotted.
- **aspect** (str) – string ‘equal’ or ‘auto’.
- **colors** (dict) – dictionary for colors used for material names.
- **nmarker** (dict) – dictionary specifying solids names to be marked on the plot. ‘name’:*’ or ‘name’:dict, where ‘*’ or dict defines the marker shape, as in the matplotlib.pyplot.plot() function.
- **mmarker** (dict) – as nmarker, but to mark solids with particular material names.
- **legend** (bool) – Show the legend on the plot.
- **filename** (str) – File name where the plot will be stored. This attribute is passed to the matplotlib.pyplot.Figure.savefig() method.
- ****kwargs – keyword arguments describing contour lines.
This package provides a low-level interface to the MCNP5 code. It defines classes to represent cells, surfaces, materials and tallies. There is also a class representing the MCNP model as a collection of cells, surfaces, materials, etc; this class takes the task of setting cell, surface and material numbers (IDs) and has methods to generate valid MCNP input file. Moreover, there is a class that describes a workplace – a directory that contains all files necessary to start MCNP job and there is a method to start MCNP executable. Also, functions to read MCNP-generated files, are defined in this package.

Note: In contrast to the low-level interface, the high-level interface translates the code-independent geometry described with the help of the pirs.solids package into the low-interface MCNP model instance, and puts back results of MCNP run, read by the low-level interface methods, back to the code-independent geometry.

An MCNP model is represented by the pirs.mcnp.Model class. It consists of instances of the pirs.mcnp.Cell class, each referring to a material represented by the pirs.mcnp.Material and with geometry defined by instances of the pirs.mcnp.Surface and pirs.mcnp.Volume classes.

3.1 Materials

The pirs.mcnp.Material class is used to represent material composition, temperature and optional use of thermal data. It inherits the pirs.core.tramat.Mixture, described in details in pirs.core.tramat subpackage. Here we overview only MCNP-related features, added to the pirs.mcnp.Material class.

First of all, this class can take information about available cross-sections from an xsdir file. The pirs.mcnp.Material.xsdir attribute is an instance of the pirs.mcnp.Xsdir class able to read xsdir file and to store information. By default, each material instance is supplied with an xsdir containing data from $DATAPATH/xsdir file.

The pirs.mcnp.Material.card() method generates a multi-line string containing the material card describing the material in the MCNP input file. When the card() method is called, cross-section data sets are searched in the specified xsdir object for each nuclide and for the temperature specified in the pirs.mcnp.Material.T attribute. If a cross-section data set for particular nuclide and particular temperature is found, its suffix is used in the card. If there is no cross-section data exactly at the temperature T, two cross-sections are found with temperatures above and below T, and both suffices enter the material card.

```python
from pirs.mcnp import Material
fe = Material('Fe')  # Fe chemical element with nat. occurring isotopes
for T in [300, 350, 400]:
    fe.T = T
    print fe.card()
```
In this example, a material representing Iron with natural isotopic abundances is created and material card corresponding to temperatures 300, 350 and 400 K is printed. Default xsdir, read from $DATAPATH/xsdir file is used to find cross-section suffices. As one can note, cards cannot be used directly in the MCNP input file, since material numbers are not given explicitly, there are format placeholders instead.

For temperatures 300 and 400 K data for all Fe isotopes exist in the xsdir and denoted by suffices 31c and 32c (this example uses xsdir file from the multi-temperature data set JEFF-3.1).

For the material at 350 K, both suffices are used in proportions defined by the material temperature and temperatures of available cross-sections. How these proportions are computed, depends on the pirs.mcnp.Material.Tif attribute that by default set to the pirs.mcnp.auxiliary.xs_interpolation.sqrT() function that implements the square-root temperature interpolation, but can be changed by the user to anything else.

Use of thermal data is controlled by the pirs.mcnp.Material.thermal attribute. By default it is None and no thermal data is used. When this attribute is set to a string, thermal data with names containing this string will be searched in the xsdir file and, if found, will be mentioned in the mt card corresponding to the material. If there are more than one thermal data set, the set with the closest temperature will be chosen.

```python
from pirs.mcnp import Material

h = Material('H')
o = Material('O')

h2o = 2*h + o

# thermal data
h2o.thermal = 'lwtr'

# nuclide substitution
h2o.sdct[8018] = 8016

for t in [300, 350, 400]:
    h2o.T = t
    print(h2o.card())
```

```
m(0:<)
  26054.31c 5.84500e-02
  26056.31c 9.17540e-01
  26057.31c 2.11900e-02
  26058.31c 2.82000e-03
m(0:<)
  26054.32c 2.81792e-02
  26056.32c 4.42353e-01
  26057.32c 1.02159e-02
  26058.32c 1.35954e-03
m(0:<)
  26054.31c 2.81792e-02 26054.32c 3.02708e-02
  26056.31c 4.42353e-01 26056.32c 4.75187e-01
  26057.31c 1.02159e-02 26057.32c 1.09741e-02
  26058.31c 1.35954e-03 26058.32c 1.46046e-03
mt(0:<) lwtr01.31t
```

3.1. Materials
In this example, the material h2o is created. Thermal data for hydrogen bound in water are all named lwtr???.31t in the default xsdir file. To use them, we set the thermal attribute to the common part of the names, which is ‘lwtr’. Depending on the material temperature, particular data set is chosen. Note that thermal cross-sections are not interpolated. Note also that both m and mt cards followed by the format placeholder with the same index, to ensure that both cards will correspond to the same material.

Another feature shown in this example is the pirs.mcnp.Material.sdict attribute. This dictionary specifies substitution rules for cases when particular nuclide cannot be found in the xsdir file. Particularly, the default xsdir file contains no cross-section data for nuclide 8018, which however, enters to the h2o material since it was defined using natural isotopic composition of oxygen. Without specifying the substitution rule, the call to the card method would result in error. With the help of sdict we can avoid this error.

### 3.2 Surfaces and volumes

There are two classes to describe geometry of an MCNP model.

The pirs.mcnp.Surface class is a container for data to describe an MCNP surface. It can hold data for both simple surfaces and macrobodies, ‘knows’ about the order of facets for macrobodies, can generate the surface card for the MCNP input file.

```python
from pirs.mcnp import Surface

s1 = Surface('px 1.0 $ a plane')
s2 = Surface('* pz 5.1')

s3 = Surface(type='c/z', plst=[0, 0, 6], cmnt='cylinder at z axis')
s4 = Surface('rcc 0 0 0 0 0 5 3')

#surface cards
for s in [s1, s2, s3, s4]:
    print s.card()
```

Surface parameters can be specified in two ways: the surface class constructor accepts surface cards (so it can be used as a parser), or surface type, list of parameters, reflection etc. can be specified as keyword arguments. At the initialization surfaces are simplified, when possible, as shown for surface s3. The surface card generated by the pirs.mcnp.Surface.card() method is a formatting string with placeholder for the surface ID number.
A surface separates the space into two volumes, ‘above’ and ‘below’. Representation these volumes and operations of union and intersections are implemented in the `pirs.mcnp.Volume` class. The constructor of this class takes two arguments: the first argument specifies part of the space (1 means ‘above’ and -1 – ‘below’), the second argument defines the surface. The main functionality of the volume class is to provide operations of union and intersection and to represent these operations in terms used in the MCNP input file; it should be noted however, that result of these operations is not evaluated. Thus, the second argument must not necessarily be an instance of the `pirs.mcnp.Surface` class, one can use abstract string names.

```python
from pirs.mcnp import Volume

v1 = Volume(1, 'a')
v2 = Volume(1, 'b')
v3 = Volume(-1, 'c')

# new volume as intersection and union
r = v1 & v2 | v3

# string representation of volume
print ' r: ', r
print '-r: ', -r

# surface definition substitution
s = {}
s['a'] = 1
s['b'] = 2
s['c'] = 3

print ' r: ', r.copy(s)
print '-r: ', -r.copy(s)
```

```
r: (a b):-c
-r: (-a:-b) c
 r: (1 2):-3
-r: (-1:-2) 3
```

String representation of a volume uses the space for intersection and the colon for union, and can therefore be used in the description of cell geometry in the MCNP input file, if surface ID numbers are used as abstract surface definitions. One can also use volumes with arbitrary (not necessarily integer) surface definitions together with the `pirs.mcnp.Volume.copy()` method. Optional argument of this method must be a mapping (function or dictionary) that will replace original surface definition in the returned copy.

This, for the first glance over-engineering approach allows to use macrobodies to define geometry and then, if necessary (for example, when different boundary conditions must be set to facets of a macrobody), to represent this geometry in the MCNP input file using simple surfaces. The following example illustrates this:

```python
from pirs.mcnp import Surface

c = Surface('rcc 0 0 0 0 0 10 4')

# mapping surface -> ID
l = []
for f in c.facets():
    l.append(f.a1[1])
m = lambda s: l.index(s) + 1

# macrobody exterior defined by simple surfaces
v = c.volume(m)
print ' c cells'
```

3.2. Surfaces and volumes
print '1 0 ', v, ' $ cylinder exterior'
print '2 0 ', -v, ' $ cylinder interior'

print ''
print 'c surfaces:'
for s in l:
    print str(s).format(m(s))

The c surface is a cylinder macrobody. Its pirs.mcnp.Surface.facets() method returns a list of volumes ‘above’ each macrobody’s facet. The pirs.mcnp.Volume.a1 attribute of a simple volume (i.e. defined directly by the class constructor, not as union or intersection) returns the constructor arguments, thus the second element of this attribute is the surface definition. In the loop all surfaces are collected into the list l that is used to map each surface instance to an integer number, see function m. The pirs.mcnp.Surface.volume() method returns a volume representing the macrobody’s exterior in terms of simple surfaces. It is used in the example to generate geometry description of cells representing both interior and exterior of the macrobody. Using the same mapping, we generate the surface cards, thus ensuring that surface IDs in the surface cards block is consistent with description of cells.

In this example, we defined explicitly the mapping to set surface IDs. This was done as illustration, there are other means to set automatically surface, cell and material numbers, see below.

**Warning:** The low-level interface was developed keeping in mind rather simple geometries that can be described by vertical cylinders and boxes with facets perpendicular to the coordinate axes. Therefore, only vertical cylinder macrobodies (i.e. parallel to the 3-rd coordinate axis) will be handled properly, although no parameters check is done at the initialization time.

### 3.3 Cells and models

The pirs.mcnp.Cell class represents a container that stores cell-related information: material, geometry and cell options. Cell geometry must be specified using the Volume and Surface classes, and material – using the Material class. Cell options (e.g lat, fill, imp:n etc.) are specified using the pirs.mcnp.Cell.opt attribute, which is a dictionary with only particular keys allowable (see CellOpts class). Although there is a method to generate cell cards for the MCNP input, pirs.mcnp.Cell.card(), its direct use makes no sense, since one still needs to specify surface, cell and material IDs.

```python
from pirs.mcnp import Material, Surface, Cell, Model

c1 = Cell()

# Cell 1
c1.mat = Material('Fe')
c1.rho = -10.
c1.vol = Surface('so 8.0').volume()
c1.opt['imp:n'] = 1

# Cell 2
```
c2 = Cell()
c2.vol = -c1.vol

# direct use of cells
print c1.card()
print c2.card()

# cells in a model
m = Model()
m.cells.append(c1)
m.cells.append(c2)

for c in m.cards():
    print c

{ID} {mat} {rho} {geom} imp:n=1 $ comment
{ID} 0 {geom} imp:n=0 $ comment
MESSAGE: datapath=/home/local/KIT/rx8040/data/mcnp/all_jeff

c title
1 1 -10.0 1 imp:n=1 $ comment
2 0 -1 imp:n=0 $ comment

c surfaces
1 so 8.0

c data cards
c materials
ml
  26054.31c  5.84500e-02
  26056.31c  9.17540e-01
  26057.31c  2.11900e-02
  26058.31c  2.82000e-03

c tallies
c kcode 500 1.0 20 100 j j 100000 j
prdmp j j 1 $ write mctal file

In this example, neither material nor geometry is resolved when cells c1 and c2 are printed directly, since there is no rule to set material and surface IDs. This task is accomplished by the `pirs.mcnp.Model` class that is basically a list of cells. Cells describing the model are added to the `pirs.mcnp.Model.cells` list attribute. When converting to a string, or as in the example, when calling the `pirs.mcnp.Model.cells()` method, all cells in the model are analyzed: IDs are set to unique materials and surfaces thus providing information for the cell, surface and material cards.

Setting of IDs for surfaces and materials is done with the help of the `pirs.mcnp.SurfaceCollection` and `pirs.mcnp.MaterialCollection` classes. They both have the `index()` method, which takes as argument a surface or material instance, adds it to the collection if it is not already there, and returns its index, which was attached to this particular instance when it was added to the collection.

### 3.4 Start MCNP

To manually add cards to the automatically generated input files, there are `Model.amc`, `Model.acc`, `Model.asc` and `Model.adc` list attributes containing strings that will be added to the message, cell, surface or data blocks. In this way one can define the source distribution or add manually cells.

The MCNP code can be started on the input file. The `Model.wp` attribute is an instance of the
that can be used to specify directory names where the input file will be written and MCNP code started. The `Model.run()` method prepares all necessary files and can be used to start MCNP in different modes (neutron transport, plot geometry).

TODO: example showing work with Model.wp and Model.run().

### 3.5 Docstrings

**class** `pirs.mcnp.Xsdir(path=None)`

Container for data from xsdir file.

Data can be added manually or read from existing file.

**awr**

Property represents the `atomic weight ratios` section of xsdir file. This is a dictionary: keys are ZAIDs, values are nuclide masses in terms of awr.

**clear()**

Removes data from awr, dir and path.

**datapath**

Returns path to the xsdir file, if the data in the xsdir instance was read from a file.

**classmethod default()**

Default system xsdir file.

Returns an instance of the `Xsdir` class containing data from the `$DATAPATH/xsdir` file.

**dir**

A list whose elements represent lines from the directory section of the xsdir file. Elements are instances of the `DirEntry()` class.

**filename**

Returns the name of the xsdir file, if the data in the xsdir instance were read from a file.

**find_thermal(namepart, T)**

Returns the name of thermal data containing string namepart, closest to temperature T (in K).

**read(path, append=False)**

Read existing xsdir file.

The path argument specifies relative or absolute path to the xsdir file.

If the optional argument append is True, data read from the xsdir file are appended to the data already stored in the instance of `Xsdir()`. Otherwise, the clear() method is called before reading the file.

**suffix(ZAID, T=None, xtype='v', smin=None, smax=None)**

Find suffices of the cross-section data of type xtype describing ZAID at temperature T. T must be specified in Kelvin.

Two suffices are returned, for cross-sections at temperatures closest to T, below and above T.

If smin or smax are specified, they define interval of suffixes that are searched for the closest temperature.

The returned value is always a list of two tuples, in the form [(T1, S1), (T2, S2)], where T1 and T2 are cross-section temperatures below and above T, and S1 and S2 are the correspondent suffices.

If T is not specified, T1 and S1 are parameters of the first cross-section data found in the directory section for nuclide defined by ZAID. In this case, T2 and S2 are the same as T1 and S1.
class `pirs.mcnp.Material`(*args)

Object-oriented representation of material composition for MCNP.

Constructor arguments are passed to the constructor of the parent class, see description of available arguments there.

One can setup material composition, temperature and specify xsdir file, which is used to find cross-section data suffixes.

**T**

Temperature of material, K

Material temperature is used to find proper suffix (or the pair of suffices) in xsdir file.

It is also used to find the most close thermal data, if the `self.thermal` attribute is specified.

**Tif**

Temperature interpolating function.

A function that takes as arguments temperatures $T$, $T_1$ and $T_2$, and returns fractions $f_1$ and $f_2$ of cross-sections evaluated at temperatures $T_1$ and $T_2$ used to represent a material at temperature $T$. It must have the following signature:

$$f_1, f_2 = \text{func}(T, T_1, T_2)$$

This function is used when the temperature of material is set to a value than cannot be found in the xsdir file. In this case, each isotope of the material at temperature $T$ is represented as a mixture of two, at temperatures $T_1$ and $T_2$. The temperatures $T_1$ and $T_2$ are found in xsdir automatically, being the closest to $T$ from below and above.

Note that values $T_1$ and $T_2$ are usually defined from the xsdir file, where temperature is given originally in MeV (as $kT$), and the material temperature $T$ is given in Kelvin. Thus, even if one specifies $T$ as “a temperature from xsdir”, it will not much $T_1$ or $T_2$ exactly. To avoid unnecessary cross-section data in the material specification, the user is responsible to provide the logic inside func to set $f_1$ or $f_2$ exactly to 1.0 or to 0.0 even if $T$ does not match exactly, but is close to $T_1$ or $T_2$.

Cross sections at temperature $T_1$ and $T_2$ appear in the material specification only if the correspondent fraction, $f_1$ or $f_2$, is nonzero.

By default, the `auxiliary.xs_interpolation.sqrT()` function is used, which describes the $\sqrt{T}$ interpolation.

**card** *(formatted=True, suffixes=True, smin=None, smax=None)*

Returns a multi-line string with the material cards for MCNP input.

The returned string generally represents two cards, m and mt.

The `formatted` optional argument defines if the lines in the string are wrapped to fit to 80 characters.

The `suffixes` optional argument defines if suffixes for particular cross-section sets are printed or not.

Optional parameters `smin` and `smax` can take integer values. If they are specified, only cross-sections with suffix numbers satisfying $smin \leq XX \leq smax$ are returned.

The material number is NOT defined explicitly, there is just a placeholder for it. Use the format() method of the returned string to put particular material ID:

```python
>>> m = Material(1001)
>>> print m.card().format(1)  # put 1 as material number
ml $ mixture H-001 at 300.0 K
  1001.31c 1.0000000e+00
```

**fmt**

Dictionary of format strings used to generate card representation of the material.
The following keys have sense:

- `zaid`: format string for ZAIDs
- `fraction`: format string for fraction.

If explicit field index is used (for Python versions < 2.6), it should be set to 0.

**sdict**

Substitution dictionary.

A dictionary of the form `{ZA1:za1, ZA2:za2, ...}`, where ZAi and zai are integer numbers representing ZAIDs.

If a nuclide with ZAID ZAi is not found in xsdir, it is substituted with cross-sections for nuclide zai.

By default, there is no substitutions, the dictionary is empty.

**thermal**

Part of the cross-section data set name for thermal scattering.

When this property is given, the string representing material in the MCNP input file, contains additionally to m card also mt card. The xsdir file is searched for thermal data with names containing `thermal` as a substring. If several data sets are found, the one with the closest temperature is chosen.

**xsdir**

Instance of Xsdir() class.

Suffices to represent material in the MCNP input are defined based on the content of this xsdir file and value of attribute T.

This property can be set to a string, in which case this string should represent path to existing xsdir file, or to an instance of the Xsdir() class.

---

**class pirs.mcnp.Surface**(card=None, **kwargs)**

MCNP surface.

Instances are immutable.

The constructor takes a string with usual definition of a surface as appears in an input file, except the surface ID. Alternatively, parameters of a surface can be specified as keyword arguments.

**Parameters**

- **card**(str) – is a string representing the surface card of the MCNP input, except the surface number should not be specified, for example `'+ px 1.0'`.
- **type**(str) – a string representing the surface type, for example `‘px’`.
- **refl**(char) – a character representing the reflection of the surface, for example `‘*’`.
- **plst**(list) – a list or tuple of parameters of the surface.
- **cmnt**(str) – a string representing the surface comment.

**MBTYPES** = ['rpp', 'rcc']

Known macrobody surfaces

**PRECISION** = 9

Precision of surface parameters. Parameters rounded to this number of digits when printed to the card or when two surfaces are compared.

**SSTYPES** = ['px', 'py', 'pz', 'cz', 'c/z', 'so', 's']

Known simple surfaces

**card**(formatted=True)

Returns string representing the surface, valid for an MCNP input file, except the surface ID.
If optional argument formatted set to True (default), the returned string can contain new-line characters, so that the lines fit to 80-characters limit required by the MCNP input file syntax.

This method is used to transform an instance of the Surface() class to a string.

**facets** *(mapp=<function <lambda> at 0x2b9e7d7648c0>)*

Returns the list of volumes for each facet of the surface.

- **Parameters**
  - **mapp** *(func)* – a mapping applied to the facets before passing to the Volume() constructor.

- **Returns** list of volumes.

If the surface is a simple surface, the returned list contains single element.

**is_macrobody**

Returns True if the surface is a macrobody.

**prm**

The tuple of surface parameters.

**rfl**

Reflection of the surface. Can be ’ ’ (empty string), ’ *’ or ’ +’.

**tpe**

Type of the surface, for example, ’ px’.

**volume** *(mapp=<function <lambda> at 0x2b9e7d764758>)*

Volume ‘ above’ the surface.

- **Parameters**
  - **mapp** *(func)* – a mapping applied to the surface facets before passing to the Volume() constructor.

- **Returns** an instance of the Volume class. The Volume() class defines operations of union, intersect and negation.

Instances of the Volume() class returned by this method are always defined via simple (not macrobody) surfaces. To get an instance of the Volume() class defined via a macrobody surface, use the constructor Volume() directly.

**class** pirs.mcnp.Volume *(sign=1, surface=None)*

Representation of the cell geometry using signed surfaces and union, intersection and negation operations.

A new volume instance is created by specifying the sign and the surface description. The surface description can be of any type (although in real MCNP applications, an instance of the Surface class must be used):

```python
>>> v1 = Volume(1, 'a')  # 'a' here is an abstract surface definition
>>> print v1
a
```

There are two special volumes representing by tuples (1,) and (-1,). The first means ‘empty set’, the second is ‘whole space’. To create the empty set, multiply usual volume by 0. To create whole space, multiply empty set by -1:

```python
>>> e = Volume()*0  # empty set
>>> print e
Empty Set
```

```python
>>> w = -e  # whole space
>>> print w
Whole space
```

Instances of the Volume() class support operations of union ‘|’, intersection ‘&’ and negation ‘ -‘.
a
Returns the list of operands, [self.a1, self.a2].

a1
Returns the first operand of the volume, if the volume was defined using intersection or union operators. For a simple volume, i.e. defined directly by the Volume() constructor, this is a tuple (sign, surface).

>>> v = Volume(1, 'a')
>>> v.a1
(1, 'a')
>>> v = Volume(1, 'a') & Volume(1, 'b')
>>> v.a1
<__main__.Volume object at ...>

a2
Returns the second operand of the Volume class, if the volume was defined using union or intersection. For a simple Volume returns 0.

>>> Volume(1, 'a').a2
0
>>> (Volume(1, 'a') & Volume(1, 'b')).a2
<__main__.Volume object at ...>

copy (mapp=<function <lambda> at 0x2b9e7d766320>)
Returns a (deep) copy of the volume.

The returned volume has the same structure as the original one. The surface definitions are defined using the mapping mapp applied to the original surface definitions.

UPD: mapp can be a dictionary.

Lets create a complex volume:

>>> v = Volume(1, 'a')
>>> for c in ['b', 'c']:
...     v = v & Volume(-1, c)
...     v = v | v
...
>>> print v
(((a -b):(a -b)) -c):(((a -b):(a -b)) -c)

A copy of v with the upper() method of a string applied:

>>> print v.copy(lambda x: x.upper())
(((A -B):(A -B)) -C):(((A -B):(A -B)) -C)

intersection_operands ()
If a volume is an intersection of simple or compound volumes, returns two lists. The first contains simple volumes, the second – compound volumes that defined as a union.

is_empty ()
Checks if the volume is the specifal empty volume.

is_intersection ()
True if self was obtained only by intersection operations.

is_simple ()
Checks if the volume is simple, i.e. not a result of operation on another volumes.

is_special ()
Checks if the volume is the whole or empty volume.
is_union()
True if self was obtained only by union operations.

is_universal()
Checks if the volume is the whole volume (universal set).

op
Returns the operator used to create the volume. For a simple volume returns None.

>>> print Volume(1, 'a').op
None
>>> (Volume(1, 'a') & Volume(1, 'b')).op
'and'
>>> (Volume(1, 'a') | Volume(1, 'b')).op
'or'

static sort_operands(op1, op2)
Sorts operands for the & and | operations.

surfaces()
Returns a list of surface definitions used to define the volume.

>>> v1 = Volume(1, 'a')
>>> v2 = Volume(-1, 'b')
>>> v = v1 & v2
>>> v.surfaces()
['a', 'b']
>>> v = Volume(1, 'c') | v
>>> v.surfaces()
['c', 'a', 'b']

volumes(reference=None)
Returns the list of simple volumes used to define the volume.

TODO: describe the reference optional argument.

class pirs.mcnp.Cell(**kwargs)
Representation of the MCNP cell card.

This is a container for cell material, density, geometry description and options, that can generate string representation of the cell for the MCNP input file.

Constructor can take optional keyword arguments to specify cell parameters and options:

>>> c = Cell(mat=1, rho=-10., vol=(-1, 1), cmnt='comment', ID=10, 'imp:n'=2.5)
>>> print c
10 1 -10.0 -1 imp:n=2.5 $ comment

Cell parameters can be changed after initialization by setting the correspondent attributes:

>>> c.ID = 5
>>> c.mat = 4
>>> c.rho = -1.
>>> c.vol = (-1, 'a')
>>> c.opt['imp:n'] = 0
>>> print c
5 4 -1.0 -a imp:n=0 $ comment

ID
Cell ID.
At initialization set to the string ‘{ID}’.

3.5. Docstrings
card *(formatted=True)*

Returns a string representing the cell in the MCNP input file.

If optional argument formatted set to True (default), the returned string can contain new-line characters delimiting the string to lines that fit 80-characters limit imposed by the MCNP input file syntax.

Representation of cell ID, material and volume depends on the type of correspondent attributes.

If cell ID is a positive integer or a string, it is printed as is. Otherwise, placeholder `{ID}` is printed.

If mat is an nonnegative integer or a string, it is printed together with density rho. Otherwise, placeholder `{mat} {rho}` is printed.

If vol is an instance of the Volume() class containing definitions that utilize the Surface() class, or if it is an instance of the Surface() class, placeholder `{geom}` is printed. Otherwise, the string representation of vol is printed.

cmt

Cell comment.

mat

Material of the cell.

Can be an integer or an instance of the Material() class.

opt

Dictionary of cell options.

An instance of the CellOpt() class.

rho

Cell density.

vol

Cell geometry (cell volume).

Can be set to an integer, string, or to an instance of the Volume() class.

The setter method accepts also a tuple of the form (sign, def), which is transformed to an instance of the Volume() class.

```python
>>> c1 = Cell()
>>> c2 = Cell()
>>> c1.vol = Volume(-1, ['px', 0.])
>>> c2.vol = (-1, ['px', 0.])
>>> c1.vol == c2.vol
True
```

class pirs.mcnp.cells.CellOpts

A dictionary to store cell options.

This is a dictionary that allows only particular string keys. Additionally, its string representation can be used directly in the input file within cell card.

`VALIDKEYS = ('imp:n', 'u', 'fill', 'tmp', 'lat')`

tuple of valid cell option names.

getvalue *(key)*

Returns meaningfull part of the value for options ‘fill’, ‘u’ and ‘lat’.

In general, ‘fill’ is a (multi-line) string that can optionally contain comments. This method returns a list of values of this cell option.

If options were not defined, return 0.
class `pirs.mcnp.SurfaceCollection` \((iv=1, step=1)\)

Class to describe a collection of simple surfaces (SS) and macrobodies (MB).

One can ask for a surface index using the `index()` method of the collection. If a surface passed to this method is not in the collection yet, it is added with a unique index.

The `index()` method adds only the surfaces (or facets of MB) that are not already in the collection.

The `index()` method can take as argument an instance of the Surface class, or arguments valid for the `Surface()` constructor.

When a SS is added to the collection, it is compared to the previously added surfaces, including MB facets. Only if it is not found, a new entry is added to the collection.

When a MB is added to the collection, it is compared to the previously added SSs, MBs and MB facets. If all facets of MB are already defined, nothing is added. If some of the facets are defined and some are not, the missing facets are added as simple surfaces. If none of the MB facets are defined, the MB is added to the collection as MB.

Initializes new collection.

\(iv\): initial value, the first elements index. \(step\): step for indices.

```python
class `pirs.mcnp.MaterialCollection` \((xsdir=None, *args)\)

Collection of materials with common xsdir.

Elementes of this collection are instances of the Material class augmented with a set of keyword arguments specifying additional attributes for this material.

Giving attribute values separated from the material allows to use the same material instance to describe different temperatures, for example.

One can pass an integer to the `index()` method. If the collection already contains a material with this index, its is just returned. Otherwise, an index error is raised. Index 0 is always in the collection.

```python
cards (formatted=True)
```

Returns a list of multi-line strings with material cards.

If the optional argument `formatted` is True (default), the strings in the list are wrapped to fit to 80 characters of the MCNP input line maximal length.

```python
index (mat, **kwargs)
```

Returns index of material mat.

Argument `mat` can be an instance of the Material() class, or an integer.

When `mat` is an instance of the Material() class:
If mat not yet in the collection, it is added. Optional arguments specify material attributes to be changed, when material mat is processed. In this way, materials that differ only by temperature, can be represented with the same instance of the Material() class.

When mat is an integer:

If a material with this index already exists, this index is returned. If there is no element with this index, the IndexError is raised.

class pirs.mcnp.Model (SurfaceCollectionClass=<class 'pirs.mcnp.surfaces.SurfaceCollection'>, MaterialCollectionClass=<class 'pirs.mcnp.material.MaterialCollection'>, TallyCollectionClass=<class 'pirs.mcnp.tallies.TallyCollection'>, CellCounter-Class=<class 'pirs.mcnp.auxiliary.counters.Counter'>)

Model is a list of cells with common collection of materials, surfaces and tallies.

One sets the model by adding instances of the Cell class to the model.cells list. When the model is processed (for example, when converted to a string), each cell from the list is analysed: cells are added to a collection of cells, materials and surfaces used in the definition of cells are added to the material and surface collections. Collections are used to assign unique numbers (IDs) to cells, materials, surfaces, etc. When IDs are defined, a multi-line string representing the content of MCNP input file, is generated.

Cell, material, surface and tally IDs are assigned automatically.

In this way, cell and surface blocks, as well as part of the data block containing material and tally description of MCNP input file are generated automatically. One has also possibility to add lines to each block manually.

acc

List of additional cell cards. Each list element must be a string representing one cell card. Strings from the list are added to the cell block after the cell cards generated automatically.

adc

List of additional data cards. Each list element must be a string representing one data card. Strings from this list are added to the data block after the material and tally cards generated automatically.

Note, that the kcode card is treated specially, see the kcode attribute.

amc

List of additional message cards. Each list element must be a string representing one message line (do not forget to put 5 spaces at the beginning) String from this list are added to the message block after the automatically generated cards.

>>> m = Model()

>>> m.amc.append(' runtpe=rtpl')

>>> m.amc.append(' srctp=__s')

>>> print m

MESSAGE:

datapath=D:\MCNPDATA\jeff31
runtpe=rtpl
srctp=__s

c title
...

asc

List of additional surface cards. Each list element must be a string representing one surface card. Strings from the list are added to the surface block after the surface cards generated automatically.

cards (formatted=True)

Returns list of strings representing MCNP input file.
If optional argument formatted set to True (default), strings can contain the new-line characters so that the lines fit to the 80-characters limit imposed by the MCNP input file syntax.

**cellCounter**

The instance of the auxiliary.Counter class to enumerate cells.

**cells**

List of cells. The elements of the list must be instances of the Cell() class.

Elements can be added to and removed from the list. The processing of the cells in the list, i.e. adding the materials and surfaces to the common model collections, is done inside the method _process_cells. This method is called each time the model is converted to a string.

Note that the default value of the Cell.vol attribute is a string. This cannot be used directly in the MCNP model; geometry of cells that are used in the model must be defined with the help of the Volume() and Surface() classes.

**clear()**

Removes all cells, materials and surfaces from the correspondent collections. The lists of additional cards remain.

**kcode**

kcode card.

An instance of the card_classes.KcodeCard class.

**keff()**

Reads last metal file and returns the combined Keff and its st.dev.

**materialCollection**

Instance of the MaterialCollection class. Cell materials are collected in this object.

The MaterialCollection is used to assign unique IDs to each material and to ensure that a material does not appear several times in the data block of MCNP input file.

Run _process_model to ensure that the material collection corresponds to the actual list of cells.

**run**(mode='r', **kwargs)

Prepares content of the input file and starts MCNP job.

**surfaceCollection**

Instance of the SurfaceCollection class. Surfaces used to define geometry of cells are collected in this object.

This collection is used to define unique surface IDs and to ensure that a surface does not appear in the surface block of MCNP input file several times.

Run the _process_model() method to ensure that the surface collection corresponds to the actual list of cells.

**tallyCollection**

Instance of the TallyCollection class. Tallies (currently, only mesh tallies) can be added to the model by adding them manually to the collection.

**title**

String title of the problem. Goes to the title card in MCNP input file.

**wp**

Instance of the McnpWorkPlace() class.

Prepares working directory for MCNP and starts the code.

**xsdir**

xsdir of the material collection of the model. It is used to define suffices in the material cards. Note that
instances of Material class used in the description of cells can have their own xsdir objects that differ from the collection xsdir; the collection xsdir is always used to generate string representation of materials of the model.

Function(s) to represent doppler broadening temperature by mixing two other temperatures.

`pirs.mcnp.auxiliary.xs_interpolation.linT(T, T1, T2)`

Similar to `sqrT()`, but uses linear interpolation.

```python
>>> linT(310, 300, 400)
(0.9, 0.0999...)
>>> linT(300, 300, 400)
(1.0, 0.0)
>>> linT(400, 300, 400)
(0.0, 1.0)
>>> linT(310, 400, 300)
(0.1, 0.9)
```

`pirs.mcnp.auxiliary.xs_interpolation.sqrT(T, T1, T2, rtol=0.1)`

Returns fractions of XS at temperatures T1 and T2 to represent temperature T.

Computes fractions of cross-sections at T1 and T2 to represent temperature T using the square-root temperature interpolation. T, T1 and T2 must be given in absolute units (Kelvin or MeV, for example).

Returns a tuple (f1, f2), where f1 and f2 are fractions of cross-sections at T1 and T2, respectively.

Optional argument `rtol` is used to specify distance from T1 or T2, at which interpolation takes place. For example, if T is close to T1, i.e. when

|T-T1| < |T1-T2|*rtol,

interpolation is not done, tuple (1.0, 0.0) is returned. In this way one can exclude interpolation, when T differs from existing temperatures T1 or T2 only negligibly.

```python
>>> sqrT(350, 300, 400)
(0.48207..., 0.5179...)
>>> sqrT(350, 400, 300)
(0.5179..., 0.48207...)
>>> sqrT(300, 300, 400)
(1.0, 0.0)
>>> sqrT(400, 300, 400)
(0.0, 1.0)
```

Fractions f1 and f2 are defined from the following equations:

1. \( \sigma(T) = \sigma(T_1) \cdot f_1 + \sigma(T_2) \cdot f_2 \)  # this is how cross-sections can be mixed in MCNP
2. \( \sigma(T) \) is proportional to \( T^{1/2} \)  # see van der Marck, Meulekamp, Hogenbirk, M&C 2005
3. \( f_1 + f_2 = 1 \)  # this is how nuclide fractions normed in MCNP

from this equations, given T, T1 and T2, one can express f1 and f2:

4. \( f_1 = (T^{1/2} - T_2^{1/2}) / (T_1^{1/2} - T_2^{1/2}) \)
5. \( f_2 = 1 - f_1 \)
PIRS.SCF2 SUBPACKAGE

This package provides a low-level interface to the SCF code.

4.1 Docstrings

class pirs.scf2._Input (version='2.5')
O-O representation of all keywords, tables and switches of the SCF input file.
Input is a list, but string indexes are accepted as well.
>>> scf = Input()

Particular elements can be found using indexes or the find method.

>>> print scf.find('relative_heat')[0]
file = this_file
relative_axial_location relative_heat_flux
!

>>> print scf['relative_heat']
file = this_file
relative_axial_location relative_heat_flux
!

cler() Clears all data from variables, switchs, tables.

find(*names)
Returns list of all groups and group elements (i.e. switches, tables, variables) whose names contain name as a substring.

The first argument controls what is searched.

‘a’ – (default) search all: group names, variables, switches and tables. ‘g’ – search group names only ‘v’, ‘s’ or ‘t’ – search only among instances of the ScfVariable, ScfSwitch or ScfTable class, respectively.

If the first argument has another value, ‘a’ is assumed.

run(mode='r', **kwargs)
Prepares content of the input file and starts an SCF job.

wp
Instance of the ScfWorkPlace() class.

Prepares working directory for SCF and starts the code.
class pirs.scf2.OutputTable (columns, cnames, N=0)
   O-O representation of tables in the output.txt. It provides access to data and column names. Usually, instances of this class are created by the function read_output(), see below.

column (name_upper, name_lower=None)
   Returns a copy of the data column whose heading’s upper part is name_upper.

column_names
   List of column names. Each element of the list is a tuple specifying the upper and lower part of the column’s heading. Usually, it is the column data name and dimension.

columns
   List of data columns.

fmt
   Format string used to convert table to a string (when e.g. printing the table).

   Can be a string or a list of strings, representing valid format specifications, as understood by the str.format() method.

   If a string is given, it is used for all columns. If a list of strings is given, its length must coincide with the number of columns in the table. In this case, each column gets its own format string.

   not implemented: If fmt is None, each column’s width is defined by its header’s length.

row (i)
   Returns a copy of i-th row.

class pirs.scf2.RodMaterial
   Container for data necessary for the tables describing rod material properties.

   Physical units are those used in SCF.

   update (othr, part='all')
      Update properties of the RodMaterial instance by the values stored in othr.

      part: ‘all’ – all properties will be updated ‘fuel’ – only fuel-related properties will be updated ‘clad’ – only clad-related properties will be updated ‘gap’ – only gap-related properties will be updated.

class pirs.scf2.RodMaterialCollection (iv=1, step=1)
   Collection of rod materials.

   Initializes new collection.

   iv: initial value, the first elements index. step: step for indices.
CHAPTER

FIVE

PIRS.HLI.MCNP SUBPACKAGE

This package provides a high-level interface to the MCNP5 code.

5.1 Docstrings

class \texttt{pirs.hli.mcnp.interface.McnpInterface}(\texttt{gm=None, **kwargs})
McnpInterface is an MCNP model with ability to take an instance of one of the solids classes as input data.

Simple box and cylinder >>> print McnpInterface(Box()) MESSAGE:

datapath=C:Python27\lib\site-packages\mcnp

<BLANKLINE>
c title 1 0 -1 imp:n=1 $ model 2 0 1 imp:n=0 $ The other world <BLANKLINE>c surfaces 1 rpp -0.5 0.5 -0.5 0.5 -0.5 0.5 <BLANKLINE>c data cards c materials <BLANKLINE>

>>> print McnpInterface(Cylinder()) MESSAGE:

datapath=C:Python27\lib\site-packages\mcnp

<BLANKLINE>
c title 1 0 -1 imp:n=1 $ model 2 0 1 imp:n=0 $ The other world <BLANKLINE>c surfaces 1 rcc 0.0 0.0 -0.5 0.0 0.0 1.0 1.0 <BLANKLINE>c data cards c materials <BLANKLINE>

Box and cylinder with axially distributed density. Note that not only mesh itself must be set, but also the values, so the density is not constant:

```python
>>> s1 = Box()
>>> s1.dens.set_grid([1]*4)
>>> s1.dens.set_values([1.1, 1.2, 1.3, 1.4])
>>> s1.material = 'fuel'
>>> print McnpInterface(s1)
MESSAGE:

datapath=C:\Python27\lib\site-packages\mcnp

c title
1 0 -1 imp:n=1 fill=1 $ model
2 1 -1.1 2 -3 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
3 1 -1.2 3 -4 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
4 1 -1.3 4 -5 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
5 1 -1.4 5 -6 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
6 0 1 imp:n=0 $ The other world

c surfaces
1 rpp -0.5 0.5 -0.5 0.5 -0.5 0.5
2 pz -1000.5
3 pz -0.25
4 pz 0.0
```
5 pz 0.25
6 pz 1000.5

c data cards
c materials
ml $ mixture  H-001 at 300.0 K
  1001.31c 1.0

>>> s1 = Cylinder()
>>> s1.dens.set_grid([1]*4)
>>> s1.dens.set_values([1.1, 1.2, 1.3, 1.4])
>>> s1.material = 'fuel'

>>> print McnpInterface(s1)
MESSAGE:
  datapath=C:\Python27\lib\site-packages\mcnp
c title
1 0 -1 imp:n=1 fill=1 $ model
2 1 -1.1 2 -3 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
3 1 -1.2 3 -4 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
4 1 -1.3 4 -5 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
5 1 -1.4 5 -6 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
6 0 1 imp:n=0 $ The other world

c surfaces
1 rcc 0.0 0.0 -0.5 0.0 0.0 1.0 1.0
2 pz -1000.5
3 pz -0.25
4 pz 0.0
5 pz 0.25
6 pz 1000.5

c data cards
c materials
ml $ mixture  H-001 at 300.0 K
  1001.31c 1.0

The temperature mesh:

>>> b = Box()
>>> b.material = 'fuel'
>>> b.temp.set_grid([1]*5)
>>> b.temp.set_values([300, 320, 340, 400, 420])

>>> print McnpInterface(b)
MESSAGE:
  datapath=C:\Python27\lib\site-packages\mcnp
c title
1 0 -1 imp:n=1 fill=1 $ model
2 1 -1.0 2 -3 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
3 2 -1.0 3 -4 imp:n=1 tmp=2.75754976e-08 u=1 $ comment
4 3 -1.0 4 -5 imp:n=1 tmp=2.92989662e-08 u=1 $ comment
5 4 -1.0 5 -6 imp:n=1 tmp=3.4469372e-08 u=1 $ comment
6 5 -1.0 6 -7 imp:n=1 tmp=3.61928406e-08 u=1 $ comment
7 0 1 imp:n=0 $ The other world

c surfaces
1 rpp -0.5 0.5 -0.5 0.5 0.5 -0.5 0.5
2 pz -1000.5
3 pz -0.3
4 pz -0.1
5 pz  0.1
6 pz  0.3
7 pz  1000.5

c data cards
c materials
m1 $ mixture H-001 at 300.0 K
  1001.31c 1.0
m2 $ mixture H-001 at 320.0 K
  1001.31c 0.788017730687 1001.32c 0.211982269313 $ 0.788017730687 parts of 299.999663469K and
m3 $ mixture H-001 at 340.0 K
  1001.31c 0.5825662186 1001.32c 0.4174337814 $ 0.5825662186 parts of 299.999663469K and
m4 $ mixture H-001 at 400.0 K
  1001.32c 1.0
m5 $ mixture H-001 at 420.0 K
  1001.32c 0.790847540923 1001.33c 0.209152459077 $ 0.790847540923 parts of 400.007287629K and

model consisting of several solids, each with its own material:

>>> b = Box()
>>> b.X = 10
>>> b.Y = 10
>>> b.Z = 100
>>> c = b.insert(1, Cylinder())
>>> c.R = 1
>>> c.Z = b.Z
>>> b.material = 'water'
>>> c.material = 'pin'
>>> print McnpInterface(b)
MESSAGE:

datapath=C:\Python27\lib\site-packages\mcnp
c title
  1 0 -1 imp:n=1 fill=1 $ model
  2 1 -1.0 2 -3 (4:1.5:1.6) imp:n=1 tmp=2.5852029e-08 u=1 $ comment
  3 2 -1.0 -4 -1.5 -1.6 imp:n=1 u=1 tmp=2.5852029e-08 $ 1
  4 0 1 imp:n=0 $ The other world

c surfaces
  1 rpp -5.0 5.0 -5.0 5.0 -50.0 50.0
  2 pz -1005.0
  3 pz 1005.0
  4 cz 1.0

c data cards
c materials
m1 $ mixture H-001 at 300.0 K
  1001.31c 1.0
m2 $ mixture H-001 at 300.0 K
  1001.31c 1.0

A more complex model consisting of several solids and with density and temperature distributions.

>>> b = Box()
>>> b.set_radius(10.)
>>> b.material = 'm1'
>>> c1 = b.insert(1, Cylinder())
>>> c2 = b.insert(2, Cylinder())
>>> c3 = b.insert(3, Cylinder())
>>> c1.pos.z = -5
>>> c2.pos.z = 0
>>> c3.pos.z = 5
>>> b.dens.set_grid([1,1,1])
>>> b.dens.set_values([1, 2, 3])
>>> b.temp.set_grid([1,1,1, 1, 1, 1])
>>> b.temp.set_values([300, 320, 340, 360, 420, 400.])
>>> m = McnpInterface(b)
>>> print m
MESSAGE:
    datapath=C:\Python27\lib\site-packages\mcnp

c title
  1 0 -1 imp:n=1 fill=1 $ model
  2 1 -2 -3 imp:n=1 tmp=2.5852029e-08 u=1 $ comment
  3 2 -1 3 -4 5 imp:n=1 tmp=2.75754976e-08 u=1 $ comment
  4 3 -2 4 -6 (5.1:7:-8) imp:n=1 tmp=2.92989662e-08 u=1 $ comment
  5 4 -2 6 -9 (5.1:7:-8) imp:n=1 tmp=3.10224348e-08 u=1 $ comment
  6 5 -3 9 -10 (5.1:11:-12) imp:n=1 tmp=3.61928406e-08 u=1 $ comment
  7 6 -3 10 -13 imp:n=1 tmp=3.4469372e-08 u=1 $ comment
  8 0 -5 imp:n=1 u=1 § 1
  9 0 -5.1 -7 8 imp:n=1 u=1 § 2
 10 0 -5.1 -11 12 imp:n=1 u=1 § 3
11 0 1 imp:n=0 $ The other world

c surfaces
  1 rpp -10.0 10.0 -10.0 10.0 -10.0 10.0
  2 pz -1010.0
  3 pz -6.66666666
  4 pz -3.33333333
  5 rcc 0.0 0.0 -5.5 0.0 0.0 1.0 1.0
  6 pz 0.0
  7 pz 0.5
  8 pz -0.5
  9 pz 3.33333333
10 pz 6.66666667
 11 pz 5.5
 12 pz 4.5
 13 pz 1010.0

c data cards

c materials
m1 $ mixture H-001 at 300.0 K
  1001.31c 1.0
m2 $ mixture H-001 at 320.0 K
  1001.31c 0.788017730687 1001.32c 0.211982269313 $ 0.788017730687 parts of 299.999663469K and
  1001.32c 0.211982269313 $ 0.788017730687 parts of 299.999663469K and
m3 $ mixture H-001 at 340.0 K
  1001.31c 0.5825662186 1001.32c 0.4174337814 $ 0.5825662186 parts of 299.999663469K and
  1001.32c 0.4174337814 $ 0.5825662186 parts of 299.999663469K and
m4 $ mixture H-001 at 360.0 K
  1001.31c 0.383073636439 1001.32c 0.616926363561 $ 0.383073636439 parts of 299.999663469K and
  1001.32c 0.616926363561 $ 0.383073636439 parts of 299.999663469K and
m5 $ mixture H-001 at 420.0 K
  1001.32c 0.790847540923 1001.33c 0.209152459077 $ 0.790847540923 parts of 400.007287629K and
  1001.33c 0.209152459077 $ 0.790847540923 parts of 400.007287629K and
m6 $ mixture H-001 at 400.0 K
  1001.32c 1.0

bc
Dictionary specifying boundary conditions on the axial and radial boundaries of the model.

Two keys are meaningful: ‘axial’ and ‘radial’. Values can be one of ‘’, ‘*’ and ‘+’ meaning no reflection, mirror and white reflection, respectively.

```python
>>> b = Cylinder()
>>> m = McnpInterface(b)
>>> m.bc['axial'] = '*'
>>> print m
...
1 0 -3 -2 l imp:n=1 $ model
2 0 3:2:-1 imp:n=0 $ The other world

c surfaces
*1 pz -0.5
*2 pz 0.5
3 cz 1.0
...

>>> m.bc['radial'] = '*'
>>> print m
...
1 0 -1 imp:n=1 $ model
2 0 1 imp:n=0 $ The other world

c surfaces
*1 rcc 0.0 0.0 -0.5 0.0 0.0 1.0 1.0
...

>>> m.bc['radial'] = '+'
>>> print m
...
1 0 -3 -2 l imp:n=1 $ model
2 0 3:2:-1 imp:n=0 $ The other world

c surfaces
*1 pz -0.5
*2 pz 0.5
+3 cz 1.0
...
```

gm
Link to the general model. This is the input data for the MCNP input file generated by the McnpInterface.

materials
Dictionary to define the actual meaning of the material names used in the general model. Keys are strings with material names from the general model, values are instances of mcnp.Material class.

Note that instances of mcnp.Material class have their own property T (temperature). This property is not used. The temperature is defined in the temp property of each element of the general model.

Note also that each instance of the mcnp.Material class has its own xsdir property. This will be replaced with the xsdir property set to the model.

plot_commands (element=None, XYopt={}, XZopt={}, YZopt={})
A multi-line string representing plot commands, which can be feed to MCNP to produce plots of the model.

By default, commands for three plots are written: section by XY, XZ and YZ planes.

element – optional argument. If of one of solids type, its dimensions will be used to set extent of the plot.
XYopt – optional dictionary to provide user-defined options to the XY plot. For example, XYopt={'ext':3.0 6.0}

read_meshtal (meshtal='meshtal')
Reads meshtall to the correspondent meshtallies and returns the copy of the general model containing read values.
This package provides a low-level interface to the SCF code.

6.1 Docstrings
To start a computational code, several things need to be done. Where is the code executable? What input data and other files does it need? Where it is started? To simplify tasks connected to these questions, there are classes defined in the `pirs.core.scheduler` module helping to create directories, set files necessary to start a code, to start it and to wait until it completes.

A directory containing all files necessary to start a code is described in the `WorkPlace` class. An instance of this class helps to define directory names where a code will be started.

```python
from pirs.core.scheduler import WorkPlace

w = WorkPlace()
w.prefix = 'wp'

# create directory
w.prepare()
print w.report

# create another directory,
# do not interfere with previously
# created
w.prepare()
print w.report

'wp1' created by script sched_wp1.py at 2014-09-24 11:53:35.587232
'wp2' created by script sched_wp1.py at 2014-09-24 11:53:35.587439
```

When the `WorkPlace.prepare()` is called, the current directory is searched for already existing directories starting with string specified in the `WorkPlace.prefix` attribute, in the example `wp`. If there are no such directories, the directory `wp0` is created. If there already some directories with prefix `wp`, an integer `N` is chosen so that directory `wpN` does not exist yet, and this directory is created. The `WorkPlace.report` attribute is a string containing the last created directory (see also `WorkPlace.lcd`) the name of Python script where the method was called, and creation time. The above example results in two newly created empty directories, `wp0` and `wp1`.

To put files in newly created directories, the `WorkPlace.files` list attribute must be populated with instances of the `pirs.core.scheduler.InputFile` class. This class describes content and name of the file to be written or copied to the workplace directory. The content can be specified as a string or as a link to existing file.

```python
from pirs.core.scheduler import InputFile, WorkPlace

# MCNP input file
i1 = InputFile()
i1.basename = 'inp'
i1.string = 'c input file'

# Srctp from previous run
```
In this example three input files are created. The first one, i1, will be written to a file in the workplace directory with the name defined by the InputFile.basename. With content defined by the string passed to the InputFile.string argument. The second input file, i2, is a link to an existing file. In this case, the file ./srctp1 will be copied from the current directory to the workplace under the new name, srctp2. The content of the third input file, i3 is defined again as a string. Additionally, the InputFile.executable attribute is set to True meaning that this file, when written to the workplace, will be added executable mode.

As in the previous example, we create an instance of the WorkPlace class, w, but now we append the defined input files i1, i2 and i3 to the list attribute files. The prepare() method creates a new directory, as we can see in the report that this directory is now called wp2 and has three files: ‘inp’ and ‘start.sh’ are generated from string and ‘srctp2’ is copied from another file. The WorkPlace.run() method called at the last line searches for input files with executable mode set to True and if founds, starts it and waits until particular criteria are satisfied. In the example we specified to wait for 1 second, other criteria are possible, see description of the Scheduler.wait() method. The run() method standard output generated by the script.
7.1 Jobs and schedulers

TODO: to write.

7.2 Docstrings

class pirs.core.scheduler.WorkPlace

Defines common directory name and a list of InputFiles to be put to the directory.

Method WorkPlace.prepare() creates folder ‘nameID’, where ‘name’ is the common directory name and ‘ID’ is an unique integer number. The name of the last created folder is saved in the WorkPlace.lcd property.

```python
>>> wp = WorkPlace()
>>> print(wp.prefix, wp.nextID, wp.lcd)
wp 0 None
>>> wp.prepare()
>>> print(wp.report)
'wp0' created
>>> print(wp.prefix, wp.nextID, wp.lcd)
wp 1 wp0
```

One can specify files to be put to the work place using the WorkPlace.files list:

```python
>>> f1 = InputFile()
>>> f1.basename = 'f1'
>>> f1.string = 'f1 content

>>> f2 = InputFile()
>>> f2.basename = 'f2'

>>> f3 = InputFile()
>>> f3.basename = 'f3'
>>> f3.string = 'f3 content'

>>> wp.files.append(f1)
>>> wp.files.append(f2)
>>> wp.files.append(f3)
>>> wp.prepare()
```

files
List of files to be written to the workplace directory.
Elements must be instances of the InputFile class.

lcd
The name of the last created directory.

nextID = None
Integer part of the name for the next directory

prefix = None
Directory name prefix

prepare()
Creates directory with the next available directory name. If necessary, changes the nextID attribute.
Puts input files.

report
String describing the last created folder: its name and files in it.
run(**kwargs)
Find executable input file in files and starts it using scheduler module.
Assumes that directory containing all input files already created with the prepare() method.
Keyword arguments are all passed to the scheduler.Scheduler.run() method, see description there.
scheduler = None
Default scheduler, used to run executable input file.
class pirs.core.scheduler.InputFile(**kwargs)
File to be written to a workplace.
Content of the file is specified as a string or as a link to existing file.
Keyword arguments can specify all properties of the instance.
If arguments ‘exfile’, ‘string’ and ‘psf’ are specified in kwargs, the order they are set is exactly as specified here.
>>> f = InputFile(basename='if.init', string='string', exfile='_inp')
>>> f.write()
>>> print open(f.basename).read()

>>> f = InputFile(basename='if.init', exfile='_inp', psf=True)
>>> f.write()
>>> print open(f.basename).read()

basename
 Basename of the file to be written.
defined
 Returns True if string or exfile is specified.
executable
 True this file should be marked executable (necessary for scripts); False otherwise (appropriate for input and output files). Default value is False.
exfile
 Path to the external file, to be copied to the target file.
mode
 Writing mode. Can be ‘w’ or ‘a’.
If mode is ‘w’, the target file is rewritten. In this case, exfile is copied.
If mode is ‘a’, file fileName is appended. In this case, exfile is read and appended to fileName line by line. This might be inappropriate for a binary exfile.
>>> f = InputFile(basename='if.mode', string='Line 1')
>>> f.write()
>>> print open(f.basename).read()

>>> f.mode = 'a'
>>> f.write()

psf
 The “Prefer String Flag”.
Specifies what to put to the target file if both string and exfile attributes are specified. If True, string is preffered, if False, exfile is preffered.
This property set automatically each time string or exfile is set.
Can be manually set by user.
>>> f = InputFile(string='string', exfile='existing.file', basename='if.psf')
>>> f.psf = True
>>> f.write()
>>> print open(f.basename).read()

>>> f.psf = False
>>> f.write()
>>> print open(f.basename).read()

report
Report about the last written target file.

string
String to be written into the target file.

write(path='')
Write file to disk.
Create a new file containing string if string attribute is specified, or copy existing file pointed to by the exfile attribute.

class pirs.core.scheduler.Scheduler
Scheduler for jobs.

A scheduler for running shell jobs. A scheduler consists of a dict of jobs and a queue. The add() method registers a new job, the queue() method adds a job to the queue and the wait() method waits for a queued job to finish. The convenience method run() does both queuing and waiting in one call and returns the result of the wait.

Currently, all jobs will be run in the OS shell using the subprocess module, this may change at some point in the future.

Create a new empty job scheduler.

add(name, job)
Add a job to the scheduler. It will be identified by name, any existing job with the same name is replaced.

get_job(j)
Get a job that was added to the scheduler, identified by its name. Alternatively just use dictionary syntax (see last example):

>>> s = Scheduler()
>>> j = Job('echo hello world')
>>> s.add('j', j)
>>> s.get_job('j')
<job cd '.'; echo hello world>
>>> s['j']
<job cd '.'; echo hello world>

queue(name)
Add the job identified by name to the current job queue. The same job may be queued several times.

If no job by the given name exists, a ValueError is raised, e.g:

run(name, **kwargs)
Queue the job identified by name to the job queue wait for it to finish and return the result. For example:

>>> s = Scheduler()
>>> s
{}
>>> s.add('hi', Job('echo hi'))
>>> s.run('hi')
'hi
'

You can also wait for a number of seconds or for the creation of a file in the execution directory of the Job. You can combine these options:

>>> s = Scheduler()
>>> s.add('t', Job('touch test'))
>>> s.run('t', sec=1, file='test')
''

The kwargs dictionary is passed to the wait() method, see description of allowed keyword arguments there.

`wait(name, **kwargs)`
Wait for the started job to finish.

Criteria to define whether job is finished or not, depend on the specified keyword arguments:

- `sec`: number of seconds to wait.
- `files`: list of filenames. Wait until all files specified in the list exist (created by the job). If this argument is specified, the sec argument defines the period to check file existence.
- `llines`: list of regular expressions (not regex objects!) to compare with the last string of file specified in the files argument. The llines argument implies that the files argument is given and that they have equal lengths. Criteria meet, if all files exist and their last lines match the regular expression strings.

`class pirs.core.scheduler.Job(cmd, dir=')`
A shell job to be executed by the OS shell. A job is described by a command and a working directory. The working directory is optional, if omitted, the current directory is used. For example:

```python
>>> j = Job('echo hello')
>>> j
<job cd '.'; echo hello>
```

```python
>>> j = Job('sleep 4', '/home')
>>> j
<job cd '/home'; sleep 4>
```

If the string you pass as working directory does not describe a directory on your current machine, a ValueError is raised, e.g.

```python
>>> j = Job('sleep 1', '+4w34\d')
Traceback (most recent call last):
  ...
ValueError: not a directory: +4w34\d
```

To execute a job, add it to a scheduler, queue() it and wait().

Create a new job consisting of a shell command cmd to run and a working directory dir in which to execute.

If dir is not a directory or not accessible by the current user, a ValueError is raised. If no directory is specified, the current working directory is used.

- `cmd`: Command to be run in the OS shell. Must be a string.
- `dir`: Working directory.
Directory where cmd OS shell command is started.
This package defines classes to represent mixture of nuclides. The classes are not used directly, they serve as parent classes providing base functionality for example, to the pirs.mcnp.Material class.

8.1 Nuclides

The basic element of any material mixture is a nuclide represented by the pirs.core.tramat.Nuclide class. Instances of this class have attributes for mass and charge numbers, for molar mass and for the isomeric state. The main functionality of this class is to ‘understand’ nuclide specifications of different forms; it also defines arithmetical operations of addition of two nuclides and multiplication of a nuclide by scalar.

How nuclides can be defined:

```python
from pirs.core.tramat import Nuclide

n1 = Nuclide((2, 4, 0))  # tuple (Z, A, I)
n2 = Nuclide('He-4')  # string
n3 = Nuclide('Ag-110m')  # string for isomer
n4 = Nuclide(2004)  # ZAID integer
n5 = Nuclide(n4)  # copy of n4.

for n in [n1, n2, n3, n4, n5]:
    print(repr(n), n.name, n.ZAID, n.M())
```

In all these examples the molar mass was not specified. In this case, it is taken from the class attribute pirs.core.tramat.Nuclide.AWR_SET, which is a dictionary of the form ZAID: awr. By default, this class attribute links to the pirs.core.tramat.data_masses.xsdir1 dictionary, which contains awr masses from the xsdir file distributed with MCNP5.

Two nuclide instances can be added and an instance can be multiplied by a scalar:

```python
a = n1 + n2
b = 3*n1
```

Both operations result in an instance of the pirs.core.tramat.Mixture class. In the example, a is a mixture of one mole of nuclide n1 and one mole of nuclide n2, and b is a mixture consisting of 3 moles of nuclide n1.
8.2 Mixtures

The `pirs.core.tramat.Mixture` class represents a mixture of nuclides or other mixtures. Its constructor takes arguments that define the mixture recipe – what materials at which amount constitute the mixture. In the most simple case, there can be only one integer argument specifying ZAID, or a string specifying chemical name. In general case, the constructor accepts several tuples each defining material and its amount used to construct the mixture.

```python
from pirs.core.tramat import Mixture

m1 = Mixture(1001)  # mixture contains only H-1
m2 = Mixture('He')  # mixture of He nuclide with nat. abund.
# Mixture does not accept string representation of a nuclide:
# m2 = Mixture('He-4') # will cause an error

m3 = Mixture((m1, 0.1, 1), (m2, 0.9, 1))  # 0.1 mole of m1 and 0.9 moles of m2
m4 = Mixture((m1, 0.1, 2), (m2, 0.9, 2))  # 0.1 g of m1 and 0.9 g of m2
m5 = Mixture((m1, 0.1, 1), (m2, 0.9, 2))  # 0.1 mole of m1 and 0.9 g of m2.

m1.dens = 1.0  # g/cm3
m6 = Mixture((m1, 0.1, 3), (m2, 0.9, 1))  # 0.1 cm3 of m1 and 0.9 moles of m2

for m in [m1, m2, m3, m4, m5, m6]:
    print(m.report())
    print('='*30)
```

In this example, `m1` is a mixture consisting of one nuclide H-1, and `m2` is a mixture of He isotopes with natural abundances. Note that the string representation of a nuclide, in the form `He-4` cannot be used in the `Mixture` constructor, while a string is assumed to define a chemical element name. Mixtures `m3`, `m4` etc. are defined using the general form of constructor arguments. Each tuple defines the material, amount and units in which amount should be understood: 1 for moles, 2 for grams and 3 for cubic cm; the latter is only possible for materials with specified density or concentration.

In the above example, mixture `m3` has two ingredients: 0.1 mole (last tuple element is 1) of `m1` and 0.9 moles of `m2`. Mixture `m4` is defined using grams: it consists of 0.1 g of `m1` and 0.9 g of `m2`. Note that one can mix units, see definition of `m5`.

One can specify ingredient amount in cubic centimeters, as for the mixture `m6` above. For this to have sense, the ingredient material, `m1` in this case, should have the `Mixture.dens` or `Mixture.conc` attribute defined.

Main properties of a mixture can be inquired by the `Mixture.report()` method. It returns a multi-line string where mixture ingredients and mixture nuclide composition is summarized. The above script produces the following output:

```
Mixture H-001
< 1001 0.9992>: 1.0 mol
    total: 1.0 mol or 1.00782474906 g or 1.00782474906 cc
Nuclide composition:
    Nuclide At.frac Wgt.frac
    < 1001 0.9992> 1.00000e+00 1.00000e+00

====================================================================
Mixture He
< 2003 2.9901>: 1.34e-06 mol
    < 2004 3.9682>: 0.99999866 mol
    total: 1.0 mol or 4.0026021646 g
Nuclide composition:
    Nuclide At.frac Wgt.frac
    < 2003 2.9901> 1.34000e-06 1.00971e-06
```
The name of a mixture, printed in the first line returned by the report() method, is given by the `pirs.core.tramat.Mixture.name` attribute. If not specified explicitly by the user (as in the example above), it is generated based on the chemical element names that constitute the mixture.

The first part of report, between the lines `Mixture ...` and `Nuclide composition:`, lists ingredients used to construct the mixture. The second part, after the line `Nuclide composition:`, shows nuclide composition, computed from the mixture recipe and recipes of its ingredients, recursively.

Mixtures can be added to each other and multiplied by a scalar, as shown in the following example.

```python
from pirs.core.tramat import Mixture

h1 = Mixture('H')
he = Mixture('He')
ml = h1 + he
```
m2 = 2*h1
m3 = 2*h1 + 3*he
m4 = 2*m1 + 3*m2

m1.name = 'm1'
for m in m1, m2, m3, m4:
    print m.report()
    print '='*30

Total: 2.0 mol or 5.01054263557 g

Nuclide composition:

Nuclide | At.frac | Wgt.frac
--------|---------|---------
< 1001  | 0.9992  | 0.999885 mol
< 1002  | 1.9968  | 0.000115 mol
< 2003  | 2.9901  | 1.34e-06 mol
< 2004  | 3.9682  | 0.0009866 mol

Total: 2.0 mol or 5.01054263557 g

Nuclide composition:

Nuclide | At.frac | Wgt.frac
--------|---------|---------
< 1001  | 0.9992  | 0.999885 mol
< 1002  | 1.9968  | 0.000115 mol
< 2003  | 2.9901  | 1.34e-06 mol
< 2004  | 3.9682  | 0.0009866 mol

Total: 5.0 mol or 14.0236874357 g

Nuclide composition:

Nuclide | At.frac | Wgt.frac
--------|---------|---------
< 1001  | 0.9992  | 0.999885 mol
< 1002  | 1.9968  | 0.000115 mol
< 2003  | 2.9901  | 1.34e-06 mol
< 2004  | 3.9682  | 0.0009866 mol

Total: 5.0 mol or 8.03436404848 g

Nuclide composition:
Result of addition of two mixtures is a new mixture that is a mix of the operands in equal proportions. Note, however, that the resulting mixture recipe does not refer directly to the operands, they are first expanded to nuclides (see the `pirs.core.tramat.Mixture.expanded()` method). In the above example, the recipe of m1 consists of 4 nuclides, not of two mixtures h1 and he.

Multiplication of a mixture by a scalar returns a new mixture with only one ingredient – the operand, which amount in moles is given by the scalar, see mixture m2. Note, in this case the operand is not expanded to nuclides.

Note that the total amount of ingredients in the mixture m4 sums up to 5 moles, although it is defined as 2*m1 + 3*m2, where m1 and m2 have itself 2 moles in their recipes. This behaviour is chosen deliberately, the scalar before the mixture is not a coefficient for the amount of its ingredients, it specifies the amount of mixture itself.

### 8.3 Data modules

Mapping between charge numbers and chemical names are defined in the `pirs.core.tramat.data_names` module. This module also contains functions to parse nuclide names and to define isomeric names from special ZAIDS.

Default awr masses are taken from the `pirs.core.tramat.data_masses` module.

Natural abundances of isotopes are taken from the module `pirs.core.tramat.data_natural`.

### 8.4 Docstrings

#### 8.4.1 Nuclide class

```python
class pirs.core.tramat.Nuclide(ID, M=None)

Representation of a nuclide. A nuclide is defined by its mass and charge numbers, A and Z, isomeric state I, and by its molar mass M.

The A, Z and I properties of a nuclide instance can be changed. The nuclide mass, represented by the property M, cannot be changed.

The constructor argument ID specifies Z, A and I and can be given in several forms:
- a tuple (Z, A, I),
- a string of the form ‘cc-AAA[m[I]]’, for example, ‘Pu-241’, ‘O-16’, ‘Te-129m’
- an integer of the form ZZAAA.
- another instance of Nuclide class; in this case, the copy of ID is created.

If M is not specified, values derived from xsdir file will be used (the xsdir file is not read. Mass is actually taken from the data_masses module, where nuclide masses in awr units are stored. These values are originally taken from an xsdir file).

The class supports some arithmetical operations:
- An instance of the Nuclide class can be multiplied by a scalar c. The result is an instance of the Mixture class meaning c moles of the nuclide.
```
Two Nuclide instances can be added, n1 + n2. This creates a Mixture representing one mole of n1 mixed with one mole of n2.

AWR_SET Dictionary with default awr masses.

A

Mass number A of the nuclide (integer). Can be set.

```python
>>> n1 = Nuclide('H-1')
>>> n1.A = 5
>>> n1.ZAID
1005
```

I

Nuclide isomeric state.

M()

Returns the nuclide molar mass. If the molar mass was not specified at the initialization of the instance, the value in AWR will be set depending on the nuclide ID:

```python
>>> n1 = Nuclide(1001, 1.)
>>> n1.M()
1.0
>>> n2 = Nuclide(1001)
>>> n2.M()
0.999167
```

Z

Charge number, Z, of the nuclide. Can be set.

```python
>>> n1 = Nuclide('O-16')
>>> n1.Z = n1.Z/2
>>> n1.name
'Be-016'
```

ZAID

ZAID (integer) of the nuclide. Can be set to any integer of the form AAZZZ.

Support for isomeric states.

check_attributes(**kwargs)

Check conditions specified in kwargs in the form ATTR=VAL. Attr can be usual attribute or a method. In the latter case, the result returned by the method is compared against VAL. Only methods without arguments can be checked.

```python
>>> n1 = Nuclide(1001)
>>> n1.check_attributes(A=1, Z=1)
True
>>> n1.check_attributes(M=0.999167)
True
```

isfuel()

Returns True if one of the fissionable isotopes 92235, 94239 or 94241.

name

The name (string) of the nuclide in the form ‘Cc-AAA[m[I]]’. For example, ‘Pu-241’.

Can be set. in this case, attributes A, Z and I will be changed accordingly.
8.4.2 Mixture class

class pirs.core.tramat.Mixture(*args)
Mixture of nuclides or other mixtures.

A mixture is defined by specifying a recipe – a sequence of ingredients (a nuclide or another mixture) each
supplemented with certain amount.

Since all ingredients in the recipe have particular amount, the total amount of all ingredients can be computed.
However, if a mixture is defined using another, previously defined mixture, the amount of latter is not “propa-
gated”.

The constructor takes arbitrary amount of arguments specifying the mixture ingredients:

\[ m = \text{Mixture}(\text{arg1}, \text{arg2}, \text{arg3}, \ldots \text{argN}) \]

Each argument must be a string, an integer or an instance of Nuclide or Mixture class, or a tuple of the form
(MAT, a) or (MAT, v, t), where MAT can be an integer, a string or an instance of the Nuclide or Mixture
class. The other tuple elements represent the amount: a is an instance of the \text{Amount} \() \text{class}, v, t \text{are value
and unit to define the amount.}

If a mixture used as ingredient in a new mixture has itself only one ingredient, this ingredient will be used in the
new mixture.

If the argument is an integer, it is understood as ZAID of a nuclide. This argument adds one mole of the nuclide
to the recipe.

If the argument is a string, it is considered as the name of a chemical element. This argument defines one mole
of the mixture of naturally occuring isotopes of this chemical element.

When the argument is an instance of the \text{Nuclide} or \text{Mixture} class, it denotes one mole of it.

When a tuple is given, its first element is understood as described above. The second and third elements define
amount and unit in which the amount is expressed, they both have default values 1. The third element is 1 for
moles, 2 for grams and 3 for cubic centimeters.

\[ M() \]
Effective molar mass of the recipe.

\[ \text{amount}(t=1) \]
Returns the total amount of ingredients, specified in the recipe definition. The \( t \) argument defines the units
(moles, g or cc).

Returned value is an instance of the \text{Amount} \() \text{class}.

See also methods \text{cc}, \text{grams} and \text{moles}.

\[ \text{cc}() \]
returns total volume in cubic centimeters of the ingredients, specified in the recipe definition.

Returned value is an instance of the \text{Amount} \() \text{class}.

Some ingredients might have no density/concentration property, in this case the direct evaluation of the
sum of ingredients volumes is not possible. In this case, the value

\[ \text{self.grams}() / \text{self.dens} \]

will be returned. If the density/concentration of self is not defined, None is returned.

\[ \text{conc} \]
Concentration of the mixture, 1/cc.

Concentration can be computed from the material ingredients, or set explicitly. The computed value ( see
derived_conc method) is used only if the concentration not set explicitly.
Use the `dens` property to set concentration in terms of density [g/cc].

**dens**
Density of the mixture, g/cc.

Density can be computed from the recipe ingredients, or set explicitly to the mixture. The computed value is returned only if the density is not set explicitly.

Internally, density is saved as concentration. Thus, if a density was set to a recipe and the molar mass of recipe (or its ingredients) was changed, the density will be changed also.

**derived_conc()**
Returns computed concentration of the recipe, 1/cc.

This can be done only if all recipe ingredients have their concentration/density attributes either set explicitly or can be computed.

**derived_name()**
The derived name is composed from the element names entering the mixture. The general form of the derived name is

‘E1-E2-E3-’

where E1, E2, E3 are the chemical element names of the mixture ingredients. 3 element names accounted at maximum. If there are other elements, the name ends with ‘-’, as shown above.

The element names are listed in the decreasing order of its molar amount in the mixture.

If the mixture consists of only one nuclide, the nuclide name will be used.

**expanded()**
Returns a new mixture instance with all ingredients resolved down to nuclides with amounts expressed in moles.

The sum of moles of each ingredient nuclide is equal to the amount of originally defined recipe.

```python
>>> r1 = Mixture( ('H', 2), ('O', 1) )
>>> r2 = r1.expanded()
>>> r1.amount() == r2.amount()
True
>>> for (m,a) in r2.recipe():
...   print m, a
...
< 1001 0.9992> 1.99977 mol
< 1002 1.9968> 0.00023 mol
< 8016 15.8575> 0.99757 mol
< 8017 16.8531> 0.00038 mol
< 8018 17.8445> 0.00205 mol
```

**grams()**
returns total mass in grams of the ingredients, specified in the recipe definition.

Returned value is an instance of the `Amount` class.

**how_much()**
Computes amount of specified ingredients or nuclides entering the mixture.

Computes the amount in units specified by the argument `t` of the mixture ingredients specified by `*args`, or of nuclides specified by `**kwargs`.

Only `t=1` (moles) or `t=2` (grams) can be used.

Returned value is an instance of the `Amount` class.
Arguments *args and **kwargs cannot be used simultaneously. If *args are specified, the **kwargs have no effect.

When *args are specified, they must be nuclides, mixtures or integers (meaning ZAID) entering to the mixture recipe. The returned value is the total amount of all specified in *args materials. The materials are searched only in the recipe, i.e. a nuclide will be accounted only if it was specified directly in the recipe. If it enters the mixture only indirectly, it is not accounted for:

```python
>>> r = Mixture( 'Fe', 8016 )
>>> print r.how_much(1, 8016)
1.0 mol
>>> print r.how_much(1, 'Fe')
1.0 mol
>>> print r.how_much(1, 'Fe', 8016)
2.0 mol
```

The optional keyword arguments, **kwargs, specify properties of nuclides that should be accounted for. In this case, the mixture first expanded so that nuclides of the mixture ingredients are also taken into account.

```python
>>> r = Mixture( 'Fe', 8016 )
>>> print r.how_much(1, ZAID=8016)
1.0 mol
>>> print r.how_much(1, Z=26)
1.0 mol
```

If neither *args nor **kwargs are specified, how_much(t) is equal to amount(t).

**index(i)**

Returns index of the ingredient i in the mixture’s recipe. Similar to the index method of a list.

Argument i can be an instance of the Nuclide or Mixture class, in which case the index of this ingredient is returned (if any).

If i is an integer, it is internally converted into Nuclide(i).

```python
>>> m = Mixture('Fe')
>>> m.index(26054)
0
>>> m.index(26056)
1
>>> m.index(26058)
3
```

**isfuel()**

Returns True if the mixture contains one of the following nuclides: 92235, 94239, 94241.

**moles()**

Returns total amount of moles of the ingredients, specified in the recipe definition.

Returned value is an instance of the Amount class.

**name**

Name of the mixture, a string.

The mixture name can be specified explicitly. If not specified explicitly, it will be derived from the recipe definition, see the derived_name() method.

```python
>>> r = Mixture(('O', 2), ('U', 1))
>>> r.name
'O-U'
>>> r.name = 'mox'
```
```python
>>> r.name
'mox'
>>> steel = Mixture('Fe', 'Cr', 'Mo', 'Ni') # more than 4 elements.
>>> steel.name
'Fe-Cr-Mo-'

normalize (a, t=1)
Change the amounts of ingredients in the recipe proportionally, so that the resulting amount is a (if a is a float, one has to specify also units by setting t=1, 2 or 3).

```python
>>> r = Mixture('O', 'U', 'O') # r has 3 moles
>>> r.dens = 10. # set density so the cc amount has sense
>>> r.normalize(1, 1)
>>> print r.amount(1), r.amount(2), r.amount(3)
1.0 mol 90.0092422778 g 9.00092422778 cc
>>> r.normalize(1, 2)
>>> print r.amount(1), r.amount(2), r.amount(3)
0.0111099702063 mol 1.0 g 0.1 cc
>>> r.normalize(1, 3)
>>> print r.amount(1), r.amount(2), r.amount(3)
0.111099702063 mol 10.0 g 1.0 cc
```

recipe()
Returns a list of tuples, representing the recipe of the mixture.

The resulting list can be passed to the mixture constructor to create another mixture:

```python
>>> r1 = Mixture('O')
>>> r2 = Mixture(*r1.recipe())
>>> r1.recipe() == r2.recipe()
True
```

remove_duplicates()
Check the mixture and join equal materials. Changes are made in-place.

Only the materials specified directly in the recipe are compared.

If a material is mentioned in the recipe several times, only one entry with the total amount will remain. The units are defined by the first mentioning of the material in the recipe.

```python
>>> r = Mixture('O', 'U', 'O')
>>> r.remove_duplicates()
>>> for (m, v) in r.recipe():
... print m, v
... '<O 15.8620> 2.0 mol '<U 235.9841> 1.0 mol
```

report()
Returns a multi-line string, describing the mixture.

tune (objective, var, err=1e-05)
Changes the mixture recipe to satisfy objective function.

The objective argument is a function taking as argument an instance of the Mixture class.

The var argument is a list of two elements each of them can be a nuclide, mixture or an integer (meaning ZAID), that were used in the definition of the mixture.

The method changes the amount of materials var[0] and var[1] in the mixture recipe, so that the objective function returns value with the module less than the err argument.
8.4.3 data_masses module

A place where sets of isotopic masses are defined.

A set of masses (the term “set” here means a collection of isotopic masses consistent to each other, not the python type) must be represented by python dictionary of the form

d = { ZAID1: mass1,
       ZAID2: mass2,
       ...
       'avogadro': Na,
       'reference': 'the data origin'}

where keys are integer ZAIDs and values are isotope masses. Additionally, each dictionary must have the key ‘avogadro’, which value is a float specifying the Avogadro Number correspondent to the mass units, and the ‘reference’ key, which is a string specifying the origin of data.

pirs.core.tramat.data_masses.xsdirl
Dictionary ZAID: awr with awr masses taken from the xsdir file distributed with MCNP5. See also the ‘reference’ key in this dictionary.

8.4.4 data_names module

Module contains mapping from Z number to chemical element names. For the chemical element names, the following rules apply:

- each name is a string of length 2. If chemical name is of length one, it is preceded by space.
- The first letter is capital. The second (if any) – small.

There are two dictionaries defined in the module: name and charge defining mappings Z: name and name: Z.

pirs.core.tramat.data_names.name Dictionary Z: name

pirs.core.tramat.data_names.charge Dictionary name: Z

pirs.core.tramat.data_names.ZAI2ZAI(z, a, i)
Inverse function to ZAID2ZAI

pirs.core.tramat.data_names.ZAID2ZAI(ZAID)
Returns (Z, A, I) tuple for the nuclide specified by its ZAID.

Rules to construct ZAIDs for isomeric states are from https://www.oecd-nea.org/dbprog/RPSD2008-endf70-paper2.pdf
\[
\begin{align*}
\text{if } I < 1: & \\
\quad \text{ZAID} = Z \times 1000 + A \\
\text{else:} & \\
\quad \text{ZAID} = Z \times 1000 + (A + 300) + I \times 100
\end{align*}
\]

\text{pirs.core.tramat.data_names.str2ZAI(name)}
\quad \text{Returns (Z, A, I) tuple for the nuclide name specified in the form } %s-%i[m[I]].

\text{pirs.core.tramat.data_names.zai(ID)}
\quad \text{Returns a tuple (Z, A, I), where I is the isomeric state.}

### 8.4.5 data_natural module

Similar to the module \text{data_masses}, there can be several data sets of the natural isotopic compositions of chemical elements, originated from different sources.

Each data set is a dictionary of the form \text{ZAID}_i: \text{af}_i. The key \text{ZAID}_i is the integer of the form \text{ZZZA}_\text{AA}_\text{A} of isotope (Z and A are charge and mass numbers of the isotope), correspondent value, \text{af}_i, is the naturally occurring atomic fraction of this isotope in the chemical element with charge number Z.

Additionally to the \text{ZAID} keys, the \text{reference} key contains a string referencing the origin of the natural abundancies.

\text{pirs.core.tramat.data_natural.d1 dictionary ZAID: af. For reference see the “reference“ key.}
PIRS.CORE.TRAGEOM SUBPACKAGE

This subpackage defines only one class, `pirs.core.trageom.Vector3` representing a vector (coordinate) in three-dimensional space. The main functionality of this class is to perform conversion between cartesian, cylindrical and spherical coordinate systems (CS). A user can work (set and get) attributes representing coordinates in these coordinate systems and the class ensures that all coordinates are at each time consistent.

```python
from pirs.core.trageom import Vector3, pi2, pi

v1 = Vector3(car=(1, 0, 0))  # x, y, z
v2 = Vector3(cyl=(1, 0, 1))  # r, theta, z
v3 = Vector3(sph=(1, 0, 0))  # R, theta, phi

print 'rotate v1:
print v1.car
v1.t += pi2
print v1.car

print 'stretch v2 2 times:
print v2.car
v2.R *= 2.
print v2.car

print 'flip v3:
print v3.car
v3.p = pi
print v3.car

rotate v1:
(1.0, 0.0, 0.0)
(0.0, 1.0, 0.0)
stretch v2 2 times:
(1.0, 0.0, 1.0)
(2.0, 0.0, 2.0000000000000004)
flip v3:
(0.0, 0.0, 1.0)
(-0.0, -0.0, -1.0)
```

A vector instance is initialized by passing a coordinate 3-tuple to the constructor. Read-only attributes `car`, `cyl` and `sph` return coordinates in the correspondent CS. Properties `x`, `y`, `z`, `r`, `t`, `z`, `R`, `t`, `p` can be set. In the above, the `v1` vector is first specified using the cartesian CS. The coordinates and the CS type is stored internally and can be used later to compute coordinates in the other CS. When `t` is changed, first, coordinates in the cylindrical CS are computed from the previously defined cartesian coordinates. Then, the theta cylindrical variable is updated and the internal CS type is set to the cylindrical CS. When we call the `Vector3.car` for the second time, i.e. after `Vector3.t` was set, the cartesian coordinates are computed from internally stored cylindrical coordinates and returned as a 3-tuple.
class pirs.core.trageom.Vector3(car=None, cyl=None, sph=None)
Vector in three-dimensional space.

Coordinate conversion between cartesian (car), cylindrical (cyl) and spherical (sph) coordinate systems is performed "on demand", i.e. when values of these coordinates are inquired by user.

Vectors are created by specifying a 3-tuple containing coordinates in cartesian, cylinder, or spherical coordinate system (CS):

```python
>>> v1 = Vector3( car=(1, 0, 0) )  # cartesian coordinates
>>> v2 = Vector3( cyl=(1, 0, 0) )  # cylinder coordinates
>>> v3 = Vector3( sph=(1, 0, pi2))  # spherical coordinates (pi2 is defined in the module as pi/2)
```

The order of values in tuples is the following:

- `(x, y, z)`  # for cartesian CS
- `(r, theta, z)`  # for cylinder CS
- `(R, theta, phi)`  # for spherical CS

Coordinates R and r used in the spherical and cylinder CS, have different meaning: R is the vector’s length, r is the length of vector’s projection onto xy plane.

If the type of coordinate system is not given explicitly, the cartesian is assumed. The following two definitions are equal:

```python
>>> v4 = Vector3( (1, 0, 0))
>>> v5 = Vector3( car=(1, 0, 0))
>>> v4 == v5
True
```

If incomplete tuples are specified, they are augmented by zeroes:

```python
>>> v1 = Vector3((1,))
>>> v2 = Vector3((1, 0, 0))
>>> v1 == v2
True
```

If the ‘car’ argument is another Vector3 object, its copy is returned. This is to make Vector3() method a type convertor.

```python
>>> v1 = Vector3( (1, 2, 3) )
>>> v2 = Vector3( v1 )
>>> v1 == v2
True
>>> v1 is v2
False
```

If no arguments are specified in the constructor, the argument car=(0,0,0) is assumed:

```python
>>> print Vector3()
car (x=0, y=0, z=0)
```

After a vector instance is created, their coordinates can be accessed by attributes `x, y, z, r, t, R, p`, which mean one of the coordinate in cartesian (x,y,z), cylindrical (r, t[theta], z), or spherical (R[ho], t[heta], p[phi]) systems.

```python
>>> v = Vector3( (1,1,1) )
>>> v.x, v.y, v.z  # cartesian coordinates
(1.0, 1.0, 1.0)
>>> v.r, v.t, v.z  # cylinder coordinates
(1.414..., 0.785..., 1.0)
>>> v.R, v.t, v.p  # spherical coordinates
(1.732..., 0.785..., 0.955...)
```
After a vector instance is created, it has at least one set of coordinates. Thus, they always can be used to get the
cordinates in another system.

The transition from one coordinate system to another is performed when a coordinate is set. For example,

```python
>>> v = Vector3( (0,0,0) )
>>> v.r
0.0
>>> v.r = 3
```

In the first line, a vector instance is created. Since the cartesian coordinates are given (by default), the internal
representation of the vector v uses cartesian system. In the second line, the radius in the cylindrical CS is
requested. This results internally in calculation of the coordinates in the cylindrical system, but the internal
representation is still uses cartesian coordinates. In the third line, the cylindrical radius is set. Now, the internal
representation is changed from cartesian to cylindrical.

Each time a coordinate is read, it is recalculated from the internal representation. Each time coordinate of a new
system is set, first the new system coordinates are updated using the current coordinate system and then the new
coordinate is set and the internal system is changed.

Sets coordinates of the vector.

Arguments car, cyl or sph must be a tuple specifying coordinates in the cartesian, cylinder or spherical coordinate
systems, respectively.

If the passed tuple contains less than 3 elements, it is augmented with zeroes.

R

R coordinate in spherical CS.

When R is set, the vector internal representation is changed to spherical (thus R, t and p are computed from
cartesian or cylinder coordinates), and then new value is set to R.

```python
classmethod UnitX()
    Returns unit vector along X axis
classmethod UnitY()
    Returns unit vector along Y axis
classmethod UnitZ()
    Returns unit vector along Z axis
classmethod Zero()
    Returns zero vector with cartesian internal coordinate system.
```

```python
>>> v1 = Vector3.Zero()
>>> v2 = Vector3()
>>> v3 = Vector3( car=(0,0,0) )
>>> v1 == v2
True
>>> v2 == v3
True
```

all

Returns a 7-tuple of coordinates in all systems, (x, y, z, r, t, R, p)

alld

Returns a dictionary with coordinates in all systems:

```python
{'x':x, 'y':y, 'z':z, 'r':r, 't':t, 'R':R, 'p':p}
```

car

Returns a 3-tuple with cartesian coordinates, (x, y, z).
card
Returns a dictionary with cartesian coordinates, {'x':x, 'y':y, 'z':z}.

copy()
Returns a new instance with the same coordinates

```python
>>> v1 = Vector3((1,2,3))
>>> v2 = v1.copy()
>>> v1 is v2, v1 == v2
(False, True)
```

cross(othr)
Vector product
cyld
Returns a dictionary with cylinder coordinates, {'r':r, 't':t, 'z':z}.

cyl
Returns a 3-tuple with cylinder coordinates, (r, t, z).

dot(othr)
scalar product

```plaintext
is_on_axis(axis='x')
test if self is on axis
```

```plaintext
is_parallel(othr)
check if self and othr are parallel taking into account machine epsilon.
```

```plaintext
is_perpendicular(othr)
check that two vectors are perpendicular taking into account the machine epsilon.
```

```plaintext
is_zero()
return true if length of self is zero
```

own
Returns a 3-tuple with coordinates in the internal CS.

```plaintext
P
p (phi) coordinate in spherical CS.
When p is set, the vector internal representation is changed to spherical (thus R, t and p are computed from cartesian or cylinder coordinates), and then new value is set to p.
```

```plaintext
r
r coordinate in cylinder CS.
When r is set, the vector internal representation is changed to cylinder (thus r, t and z are computed from cartesian or spherical coordinates), and then new value is set to r.
```

```plaintext
sph
Returns a 3-tuple with spherical coordinates, (R, t, p).
```

```plaintext
sphd
Returns a dictionary with spherical coordinates, {'R':R, 't':t, 'p':p}.
```

```plaintext
t
t (theta) coordinate in cylinder CS.
When t is set to a vector with cylinder or spherical coordinates, the internal CS is not changed. If t is set to a vector with cartesian coordinates, the spherical coordinates are computed from cartesian, and than the new value is set to t.
```
**x**

Returns x coordinate in cartesian CS.

When x is set, the vector internal representation is changed to cartesian (thus x, y and z are computed from cyl. or sph coordinates), and then new value is set to x.

```python
>>> Vector3( (1,1,1) ).x
1.0
>>> Vector3(cyl=(2**0.5, pi/4, 1.)).x
1.00...
>>> Vector3(sph=(2**0.5, pi/4, pi/2)).x
1.00...
```

```python
>>> v = Vector3( sph=(1,0,0) )
>>> v.x = 1
>>> print v
car (x=1, y=0, z=1)
```

```python
>>> v = Vector3( cyl=(2**0.5, pi/4, 1))
>>> v.x = 4
>>> print v
car (x=4, y=1, z=1)
```

**y**

Returns y coordinate in cartesian CS.

When y is set, the vector internal representation is changed to cartesian (thus x, y and z are computed from cyl. or sph coordinates), and then new value is set to y.

```python
>>> Vector3( (1,1,1) ).y
1.0
>>> Vector3(cyl=(2**0.5, pi/4, 1.)).y
1.0
>>> Vector3(sph=(2**0.5, pi/4, pi/2)).y
1.0
```

```python
>>> v = Vector3( sph=(1,0,0) )
>>> v.y = 1
>>> print v
car (x=0, y=1, z=1)
```

```python
>>> v = Vector3( cyl=(2**0.5, pi/4, 1))
>>> v.y = 4
>>> print v
car (x=1, y=4, z=1)
```

**z**

Returns z coordinate in cartesian or cylinder CS.

When z is set in a vector with spherical internal representation, the new CS will be cartesian. In other cases, i.e. when z is set to a cartesian or cylinder vector, its type is not changed.

```python
>>> Vector3( (1,1,1) ).z
1.0
>>> Vector3(cyl=(2**0.5, pi/4, 1.)).z
1.0
>>> Vector3(sph=(2**0.5, pi/4, pi/2)).z
0.0
```
>>> v = Vector3( sph=(1,0,pi/2) )
>>> v.z = 1
>>> print v
car (x=1, y=0, z=1)

>>> v = Vector3( cyl=(2**0.5, pi/4, 1))
>>> v.z = 4
>>> print v
cyl (r=1.41..., t=0.785..., z=4)
• genindex
• modindex
• search
p
pirs.core.tramat.data_masses, 73
pirs.core.tramat.data_names, 73
pirs.core.tramat.data_natural, 74
pirs.mcnp.auxiliary.xs_interpolation, 46
Symbols

_under() (pirs.solids.Box method), 8
_under() (pirs.solids.Cylinder method), 14
_under() (pirs.solids.Sphere method), 19

A

A (pirs.core.tramat.Nuclide attribute), 68
a (pirs.mcnp.Volume attribute), 39
a1 (pirs.mcnp.Volume attribute), 40
a2 (pirs.mcnp.Volume attribute), 40
abspos() (pirs.solids.Box method), 8
abspos() (pirs.solids.Cylinder method), 14
abspos() (pirs.solids.Sphere method), 19
acc (pirs.mcnp.Model attribute), 44
adc (pirs.mcnp.Model attribute), 44
add() (pirs.core.scheduler.Scheduler method), 60
adjust_grid() (pirs.solids.zmesh method), 26
all (pirs.core.trageom.Vector3 attribute), 77
alld (pirs.core.trageom.Vector3 attribute), 77
amc (pirs.mcnp.Model attribute), 44
amount() (pirs.core.tramat.Mixture method), 69
asc (pirs.mcnp.Model attribute), 44
awr (pirs.mcnp.Xsdir attribute), 36
AWR_SET (pirs.core.tramat.Nuclide attribute), 68

B

basename (pirs.core.scheduler.InputFile attribute), 59
bc (pirs.hli.mcnp.interface.McnpInterface attribute), 52
boundaries() (pirs.solids.positions.RGrid method), 25
boundary_coords() (pirs.solids.zmesh method), 26
Box (class in pirs.solids), 8

C

car (pirs.core.trageom.Vector3 attribute), 77
card (pirs.core.trageom.Vector3 attribute), 78
card() (pirs.mcnp.Cell method), 41
card() (pirs.mcnp.Material method), 37
card() (pirs.mcnp.Surface method), 38
cards() (pirs.mcnp.MaterialCollection method), 43
cards() (pirs.mcnp.Model method), 44
cards() (pirs.mcnp.SurfaceCollection method), 43
cc() (pirs.core.tramat.Mixture method), 69
Cell (class in pirs.mcnp), 41
cellCounter (pirs.mcnp.Model attribute), 45
CellOpts (class in pirs.mcnp.cells), 42
cells (pirs.mcnp.Model attribute), 45
center() (pirs.solids.positions.RGrid method), 42
check_attributes() (pirs.core.tramat.Nuclide method), 68
children (pirs.solids.Box attribute), 8
children (pirs.solids.Cylinder attribute), 14
children (pirs.solids.Sphere attribute), 19
circumscribe() (pirs.solids.Box method), 8
circumscribe() (pirs.solids.Cylinder method), 14
circumscribe() (pirs.solids.Sphere method), 20
circumscribed() (pirs.solids.Box method), 8
circumscribed() (pirs.solids.Cylinder method), 14
circumscribed() (pirs.solids.Sphere method), 20
clear() (pirs.mcnp.Model method), 45
clear() (pirs.mcnp.Xsdir method), 36
clear() (pirs.scf2.Input method), 47
clear() (pirs.solids.zmesh method), 26
cmd (pirs.core.scheduler.Job attribute), 61
cnt (pirs.mcnp.Cell attribute), 42
colormap() (in module pirs.tools.plots), 29
column() (pirs.scf2.OutputTable method), 48
column_names (pirs.scf2.OutputTable attribute), 48
columns (pirs.scf2.OutputTable attribute), 48
copy() (pirs.core.trageom.Vector3 method), 78
copy() (pirs.mcnp.Volume method), 40
copy() (pirs.solids.zmesh method), 26
corefering_sibling() (pirs.solids.Box method), 8
corefering_sibling() (pirs.solids.Cylinder method), 14
corefering_sibling() (pirs.solids.Sphere method), 20
coref() (pirs.solids.zmesh method), 26
cross() (pirs.core.trageom.Vector3 method), 78
cyl (pirs.core.trageom.Vector3 attribute), 78
cylid (pirs.core.trageom.Vector3 attribute), 78
Cylinder (class in pirs.solids), 13

datapath (pirs.mcnp.Xsdir attribute), 36
default() (pirs.mcnp.Xsdir class method), 36
defined (pirs.core.scheduler.InputFile attribute), 59
dens (pirs.core.tramat.Mixture attribute), 70
dens (pirs.solids.Box attribute), 8
dens (pirs.solids.Cylinder attribute), 14
dens (pirs.solids.Sphere attribute), 20
dens() (pirs.solids.Box method), 8
dens() (pirs.solids.Cylinder method), 14
dens() (pirs.solids.Sphere method), 20
derived_cone() (pirs.core.tramat.Mixture method), 70
derived_name() (pirs.core.tramat.Mixture method), 70
dir (pirs.core.scheduler.Job attribute), 61
dir (pirs.mcnp.Xsdir attribute), 36
dot() (pirs.core.trageom.Vector3 method), 78

element_coord() (pirs.solids.zmesh method), 26
element_coords() (pirs.solids.zmesh method), 27
elements() (pirs.solids.positions.RGrid method), 25
executable (pirs.core.scheduler.InputFile attribute), 59
exfile (pirs.core.scheduler.InputFile attribute), 59
expanded() (pirs.core.tramat.Mixture method), 70
extension() (pirs.solids.Box method), 8
extension() (pirs.solids.Cylinder method), 14
extension() (pirs.solids.positions.RGrid method), 25
extension() (pirs.solids.Sphere method), 20

facets() (pirs.mcnp.Surface method), 39
filename (pirs.mcnp.Xsdir attribute), 36
files (pirs.core.scheduler.WorkPlace attribute), 58
find() (pirs.scf2.Input method), 47
find_thermal() (pirs.mcnp.Xsdir method), 36
fmt (pirs.mcnp.Material attribute), 37
fmt (pirs.scf2.OutputTable attribute), 48

get_child() (pirs.solids.Box method), 9
get_child() (pirs.solids.Cylinder method), 14
get_child() (pirs.solids.Sphere method), 20
get_grid() (pirs.solids.zmesh method), 27
get_job() (pirs.core.scheduler.Scheduler method), 60
get_key() (pirs.core.tramat.Mixture method), 9
get_key() (pirs.solids.Box method), 9
get_key() (pirs.solids.Cylinder method), 14
get_key() (pirs.solids.Sphere method), 20
get_max() (pirs.solids.zmesh method), 27
get_parents() (pirs.solids.Box method), 9
get_parents() (pirs.solids.Cylinder method), 15
get_parents() (pirs.solids.Sphere method), 20
get_radius() (pirs.solids.Box method), 9
get_radius() (pirs.solids.Cylinder method), 15
get_radius() (pirs.solids.Sphere method), 21
get_siblings() (pirs.solids.Box method), 9
get_siblings() (pirs.solids.Cylinder method), 15
get_siblings() (pirs.solids.Sphere method), 21
get_value_by_coord() (pirs.solids.Box method), 10
get_value_by_coord() (pirs.solids.Cylinder method), 15
get_value_by_coord() (pirs.solids.Sphere method), 21
get_value_by_index() (pirs.solids.Box method), 10
get_value_by_index() (pirs.solids.Cylinder method), 15
get_value_by_index() (pirs.solids.Sphere method), 21
gm (pirs.hli.mcnp.interface.McnpInterface attribute), 42
grams() (pirs.core.tramat.Mixture attribute), 70
grid (pirs.solids.Box attribute), 10
grid (pirs.solids.Cylinder attribute), 15
grid (pirs.solids.Sphere attribute), 21

has_var() (pirs.solids.Box method), 10
has_var() (pirs.solids.Cylinder method), 15
has_var() (pirs.solids.Sphere method), 21
has_zeroes() (pirs.solids.zmesh method), 27
heat (pirs.solids.Box attribute), 10
heat (pirs.solids.Cylinder attribute), 16
heat (pirs.solids.Sphere attribute), 21
heats() (pirs.solids.Box method), 10
heats() (pirs.solids.Cylinder method), 16
heats() (pirs.solids.Sphere method), 21
hideing_parent() (pirs.solids.Box method), 10
hideing_parent() (pirs.solids.Cylinder method), 16
hideing_parent() (pirs.solids.Sphere method), 21
how_much() (pirs.core.tramat.Mixture method), 70

I
I (pirs.core.tramat.Nuclide attribute), 68
i (pirs.solids.Box attribute), 10
i (pirs.solids.Cylinder attribute), 16
i (pirs.solids.Sphere attribute), 21
ID (pirs.mcnp.Cell attribute), 41
id() (pirs.solids.Box method), 10
id() (pirs.solids.Cylinder method), 16
id() (pirs.solids.Sphere method), 21
ijk (pirs.solids.Box attribute), 10
ijk (pirs.solids.Cylinder attribute), 16
ijk (pirs.solids.Sphere attribute), 21
index() (pirs.core.tramat.Mixture method), 71
index() (pirs.mcnp.MaterialCollection method), 43
index() (pirs.mcnp.SurfaceCollection method), 43

86 Index
index() (pirs.solids.positions.RGrid method), 25
indexed() (pirs.solids.Box method), 10
indexed() (pirs.solids.Cylinder method), 16
indexed() (pirs.solids.Sphere method), 21
Input (class in pirs.scf2), 47
InputFile (class in pirs.core.scheduler), 59
insert() (pirs.solids.Box method), 10
insert() (pirs.solids.Cylinder method), 16
insert() (pirs.solids.positions.RGrid method), 25
insert() (pirs.solids.Sphere method), 22
integral() (pirs.solids.zmesh method), 27
interpolate() (pirs.solids.zmesh method), 28
intersect() (pirs.solids.Box method), 10
intersect() (pirs.solids.Cylinder method), 16
intersect() (pirs.solids.Sphere method), 22
intersection_operands() (pirs.mcnp.Volume method), 40
is_constant() (pirs.solids.Box method), 10
is_constant() (pirs.solids.Cylinder method), 16
is_constant() (pirs.solids.Sphere method), 22
is_constant() (pirs.solids.zmesh method), 28
is_empty() (pirs.mcnp.Volume method), 40
is_intersection() (pirs.mcnp.Volume method), 40
is_macrobody() (pirs.mcnp.Surface method), 39
is_on_axis() (pirs.core.trageom.Vector3 method), 78
is_parallel() (pirs.core.trageom.Vector3 method), 78
is_perpendicular() (pirs.core.trageom.Vector3 method), 78
is_simple() (pirs.mcnp.Volume method), 40
is_special() (pirs.mcnp.Volume method), 40
is_union() (pirs.mcnp.Volume method), 40
is_universal() (pirs.mcnp.Volume method), 41
is_visible() (pirs.solids.Box method), 10
is_visible() (pirs.solids.Cylinder method), 16
is_visible() (pirs.solids.Sphere method), 22
is_zero() (pirs.core.trageom.Vector3 method), 78
isfuel() (pirs.core.tramat.Mixture method), 69
isfuel() (pirs.core.tramat.Nuclide method), 68
mat (pirs.mcnp.Cell attribute), 42
Material (class in pirs.mcnp), 36
material (pirs.solids.Box attribute), 11
material (pirs.solids.Cylinder attribute), 17
material (pirs.solids.Sphere attribute), 23
MaterialCollection (class in pirs.mcnp), 43
materials (pirs.hli.mcnp.interface.McnpInterface attribute), 53
max() (pirs.solids.Box method), 11
max() (pirs.solids.Cylinder method), 17
max() (pirs.solids.Sphere method), 23
MBTYPES (pirs.mcnp.Surface attribute), 38
McpnInterface (class in pirs.hli.mcnp.interface), 49
Mixture (class in pirs.core.tramat), 69
mode (pirs.core.scheduler.InputFile attribute), 59
Model (class in pirs.mcnp), 44
moles() (pirs.core.tramat.Mixture method), 71
name (pirs.core.tramat.Mixture attribute), 71
name (pirs.core.tramat.Nuclide attribute), 68
nextID (pirs.core.scheduler.WorkPlace attribute), 58
normalize() (pirs.core.tramat.Mixture method), 72
Nuclide (class in pirs.core.tramat), 67
op (pirs.mcnp.Volume attribute), 41
opt (pirs.mcnp.Cell attribute), 42