pyg4ometry

S. Boogert, A. Abramov, A. Butcher, L. Nevay, W. Shields, S. Walk

Jan 08, 2024

CONTENTS

pyg4ometry is a package to create, load, write and visualise solid geometry for particle tracking simulations.

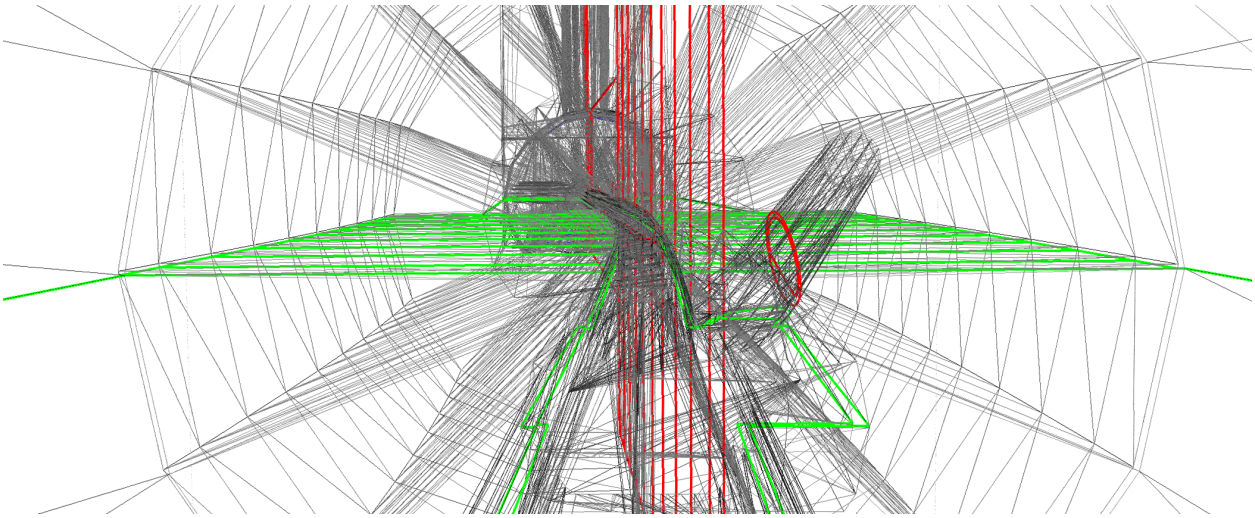


Fig. 1: Example accelerator tunnel complex at CERN.

CHAPTER

ONE

USER MANUAL

1.1 Installation

1.1.1 Requirements

pyg4ometry is developed exclusively for Python 3 (Python2 is deprecated). It is developed on Python 3.9 and 3.10.

- VTK (Visualisation toolkit) (including Python bindings)
- antlr4
- cython
- CGAL
- pybind11

Packages that are required but will be found through PIP automatically:

- matplotlib
- GitPython
- pandas
- pypandoc
- networkx
- numpy
- sympy

Optional

- Freecad for CAD conversion.

Note: A full list can be found in `pyg4ometry/setup.py`.

Note: if you are choosing a python version, it is worth choosing according to which version VTK provides a python build of through PIP if you use that. See https://pypi.org/project/vtk/#files For example, there are limited builds for M1 Mac (ARM64).

1.1.2 Installation

To install pyg4ometry, simply run `make install` from the root pyg4ometry directory:

```
cd /my/path/to/repositories/
git clone http://bitbucket.org/jairhul/pyg4ometry
git checkout develop
cd pyg4ometry

make install
```

Note: To build using the git directory and not installing into /usr/local use `make develop` instead of `make install`

To build pycsg with cpython:

```
make build_ext
```

Or install from pypi:

```
pip install pyg4ometry
```

or alternatively, run `make develop` from the same directory to ensure that any local changes are picked up.

1.1.3 Docker image

1. Download and install Docker desktop
2. open a terminal (linux) or cmd (windows)
3. (windows) Start Xming or Vxsrv
4. Download the pyg4ometry docker file
5. `docker build -t ubuntu-pyg4ometry -f Dockerfile-ubuntu-pyg4ometry .`

If you need to update increment the variable `ARG PYG4OMETRY_VER=1`

To start the container

1. open a terminal (linux/mac) or cmd (windows)
2. get your IP address `ifconfig` (linux/mac) or `ipconfig /all` (windows)
3. Start XQuartz (mac) or Xming/Vxsrv (windows). For Xming/Vxsrv (might need to play with the settings when launching)
4. `docker run -ti -v /tmp/.X11-unix:/tmp/.X11-unix -v YOURWORKDIR:/tmp/Physics -e DISPLAY=YOUR_IP ubuntu-pyg4ometry` (the `-v /tmp/.X11-unix:/tmp/.X11-unix` is only required for mac/linux)

Test the installation

1. `docker> cd pyg4ometry/pyg4ometry/test/pythonGeant4/`
2. `docker> ipython`
3. `python> import pyg4ometry`
4. `python> import T001_Box`

5. python> `T001_Box.Test(True,True)`

1.1.4 Linux installation

There are docker files for Centos 7 and Ubuntu 20. The docker files can be used as list of instructions for installation for each of these OSes.

- Ubuntu 20.02
- Centos 7

1.1.5 FreeCAD support for CAD to GDML conversion

For FreeCAD support and you already have it installed you need to add library to PYTHONPATH, for example

```
export PYTHONPATH=/opt/local/libexec/freecad/lib/
```

Building FreeCAD can be a pain for MAC so

```
mkdir FreeCAD
cd FreeCAD
set FCROOT=$pwd
wget  https://github.com/FreeCAD/FreeCAD/archive/refs/tags/0.19.4.tar.gz
tar zxf 0.19.4.tar.gz
mkdir build
mkdir install
cd build
cmake ../FreeCAD-0.19.4 -DCMAKE_INSTALL_PREFIX=../install \
-DCOIN3D_LIBRARIES=/opt/local/Library/Frameworks/Inventor.framework/Libraries/libCoin.
↪dylib -DBUILD_FEM=0 \
-DBUILD_MATERIAL=0 -DBUILD_SHIP=0 -DBUILD_DRAFT=0 -DBUILD_TUX=0 -DBUILD_ARCH=0 -DBUILD_
↪PLOT=0 \
-DBUILD_OPENSCAD=0
make -jN
make install
export PYTHONPATH=$PYTHONPATH:$FCROOT/install
```

1.1.6 Python 3.9

At the time of writing, there are limited VTK distributions for Python 3.9 on pypi (what PIP uses when finding packages). However, you can have VTK with Python 3.9 through say MacPorts or by compiling it yourself. In this case, you can comment out the VTK requirement from the setup.py around line 86, as long as you know you can `import vtk` ok in your Python installation.

Warning: ANTLR will create an unbelievable amount of warnings when using a different ANRLR version that the one the parser was generated with. It should work though. We are trying to include multiple versions of the ANTLR parser to avoid this in future.

1.2 Introduction

This package started as an internal tool for the BDSIM and machine backgrounds group at Royal Holloway. BDSIM is a tool to rapidly create Geant4 models of accelerator systems. Creation of geometry is a time consuming activity and pyg4ometry hopefully will improve the time taken to create accurate reliable geometry flies.

1.2.1 Need for programmatic geometry generation

- Non-expert user creation and maintenance of geometry
- Reduce time spent creating geometry
- Reproducibility
- Lower number of errors
- Parameterisation of geometry
- Visualisation of geometry
- Overlap checking
- Import from other geometry packages

1.2.2 Geant4 key concepts

- **solid** - describes shape only.
- **logical volume** - a solid (shape) plus a material. Practically, in Geant4 it can include fields, regions, visualisation attributes and user limits.
- **physical volume** - a placement of a logical volume. A 'stamp' out of the logical volume. It is uniquely identified by an associated integer called "copy number".
- **placement** - the term placement is used often to describe a physical volume. They are one and the same.
- **geometry reuse** - individual solids and logical volumes are encouraged to be reused. For example a row of copper boxes all the same would require only 1x solid and 1x logical volume with *N* placements (also known as physical volumes).

1.2.3 Geometry key concepts

- Constructive Solid Geometry (CSG)
- Boolean operations
- Boundary representation (B-REP)
- Boundary mesh

1.2.4 Implementation concepts

Registry

In pyg4ometry and in GDML we must uniquely identify objects by their associated name. However, in Geant4 (in C++) objects are uniquely identified by their memory address (pointer) and even for objects that have a name parameter, there is no requirement for these to be unique.

To resolve this we have the concept of a registry. This is a holder for all the definitions for a given set of geometry. It can be thought of as a namespace. It will protect against duplicate names that would prevent writing the geometry to GDML.

A `pyg4ometry.geant4.Registry` instance holds dictionaries to all solids, logical volumes and other objects, but also a nominated 'top volume' - the world volume. This needn't be the "world" as such, but is identified as the topmost part of the geometry.

When loading geometry, the typical result is a registry that contains all definitions and a top volume.

It is possible to merge two registries and all the name conflicts will be explicitly resolved. See *Combining Geometry*.

- Parameter
- ParameterVector
- Pycsg

1.3 Loading Geometry

Generally, a reader class is provided for each format. The reader is created, then told to load a file, and it creates a Registry object (see *Registry*) containing the model. The registry is the final object from a reader, and its top-most volume can be used for visualisation or other operations.

In directory `pyg4ometry/test/gdmlG4examples/ChargeExchangeMC/`

```
import pyg4ometry

r = pyg4ometry.gdml.Reader("lht.gdml")
l = r.getRegistry().getWorldVolume()
v = pyg4ometry.visualisation.VtkViewerColouredMaterial()
v.addLogicalVolume(l)
v.view()
```

In directory `pyg4ometry/test/flukaCompoundExamples`

Note FLUKA geometry can be loaded but cannot be visualised directly without conversion to a Geant4 model. This is not necessary for loading, but shown here.

```
import pyg4ometry

r = pyg4ometry.fluka.Reader("corrector-dipole2.inp")
flukaRegistry = r.flukaregistry

geantRegistry = pyg4ometry.convert.fluka2Geant4(flukaRegistry)

l = geantRegsitry.getWorldVolume()
v = pyg4ometry.visualisation.VtkViewerColouredMaterial()
```

(continues on next page)

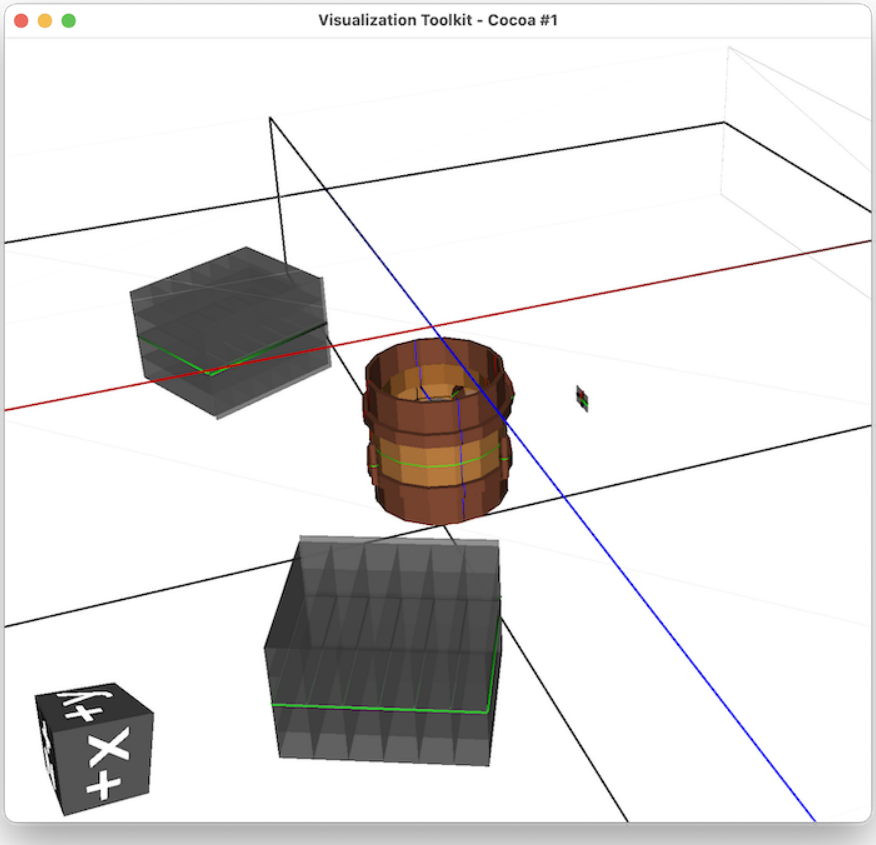


Fig. 1: Geant4 example GDML from ChargeExchangeMC example.

(continued from previous page)

```
v.addLogicalVolume(l)
v.view()
```

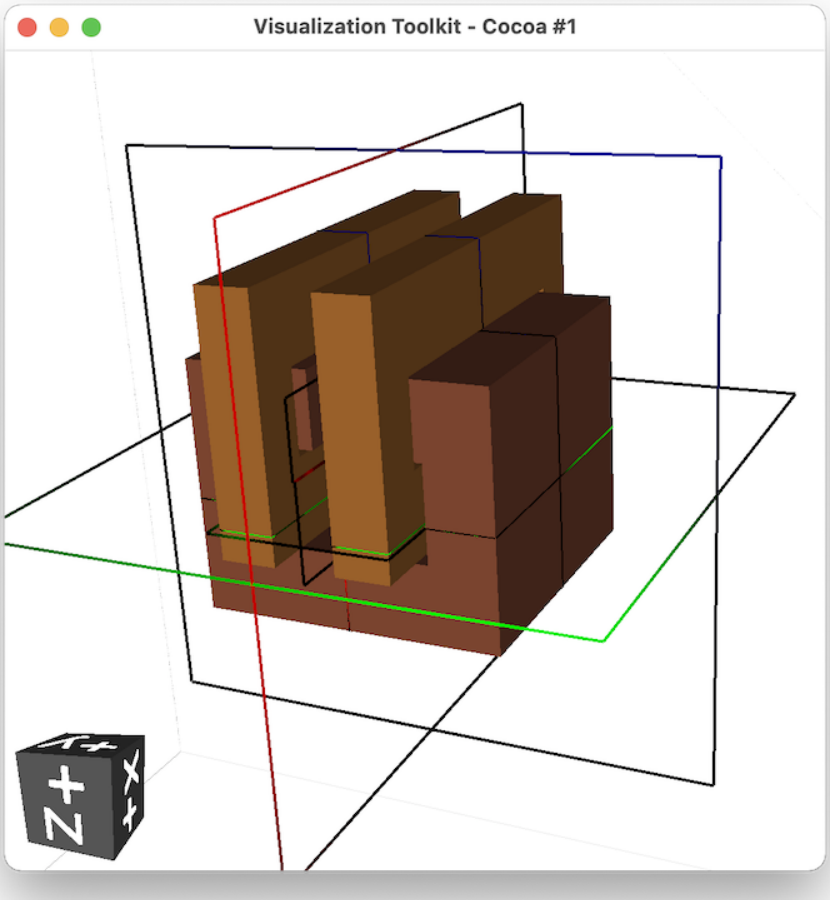


Fig. 2: FLUKA example of a dipole magnet.

In directory `pyg4ometry/test/root2Gdml`

```
import pyg4ometry

r = pyg4ometry.io.ROOTTGeo.Reader("example.root")
l = r.getRegistry().getWorldVolume()
v = pyg4ometry.visualisation.VtkViewerColouredMaterial()
v.addLogicalVolume(l)
v.view()
```

In directory `pyg4ometry/test/stl`

STL files are typically used for a single watertight solid mesh. This mesh is converted to a TesselatedSolid and then a logical volume which can be placed in a geometry. In directory `pyg4ometry/test/stl`.

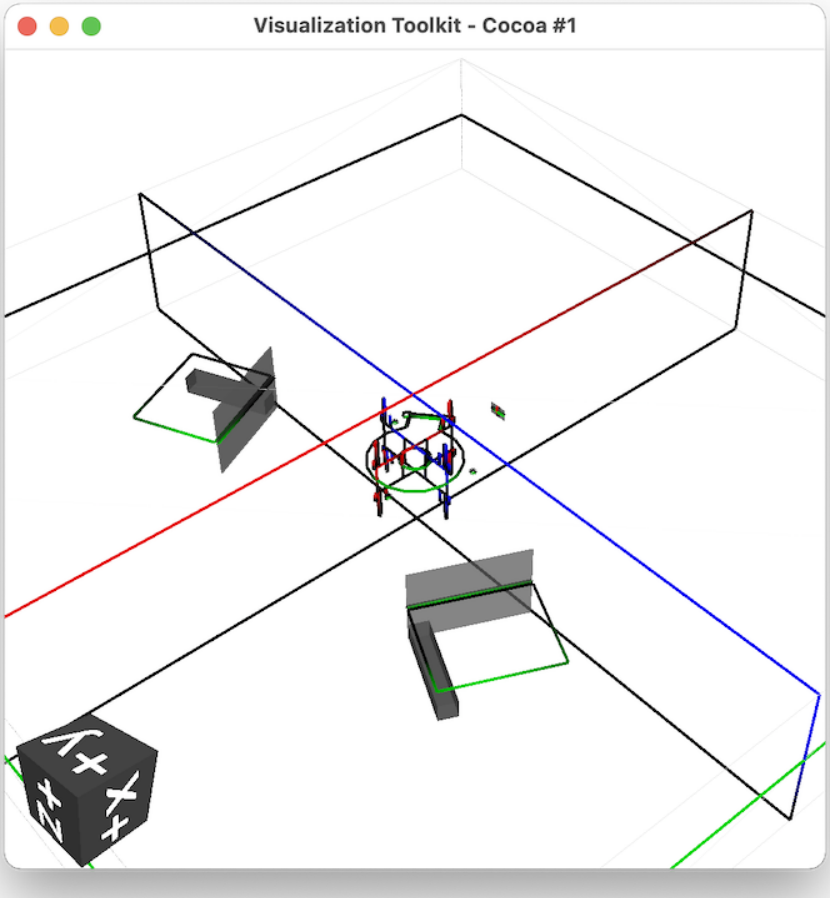
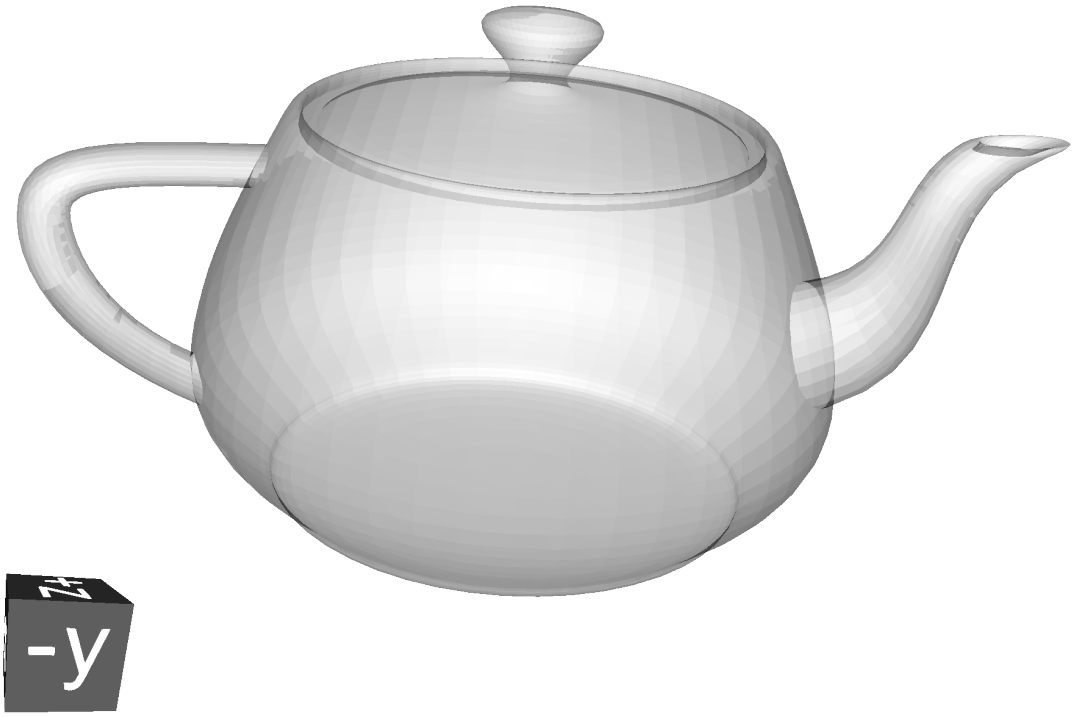


Fig. 3: ROOT example of Geant4's LHT geometry.

```
import pyg4ometry

reg = pyg4ometry.geant4.Registry()
r = pyg4ometry.stl.Reader("utahteapot.stl", reg)
s = r.getSolid()
copper = pyg4ometry.geant4.MaterialPredefined("G4_Cu", reg)
l = pyg4ometry.geant4.LogicalVolume(s, copper, "utahteapot_lv", reg)
v = pyg4ometry.visualisation.VtkViewerColouredMaterial()
v.addLogicalVolume(l)
v.view()
```



alt
Example of STL loading in pyg4ometry

In directory `pyg4ometry/test/freecad`

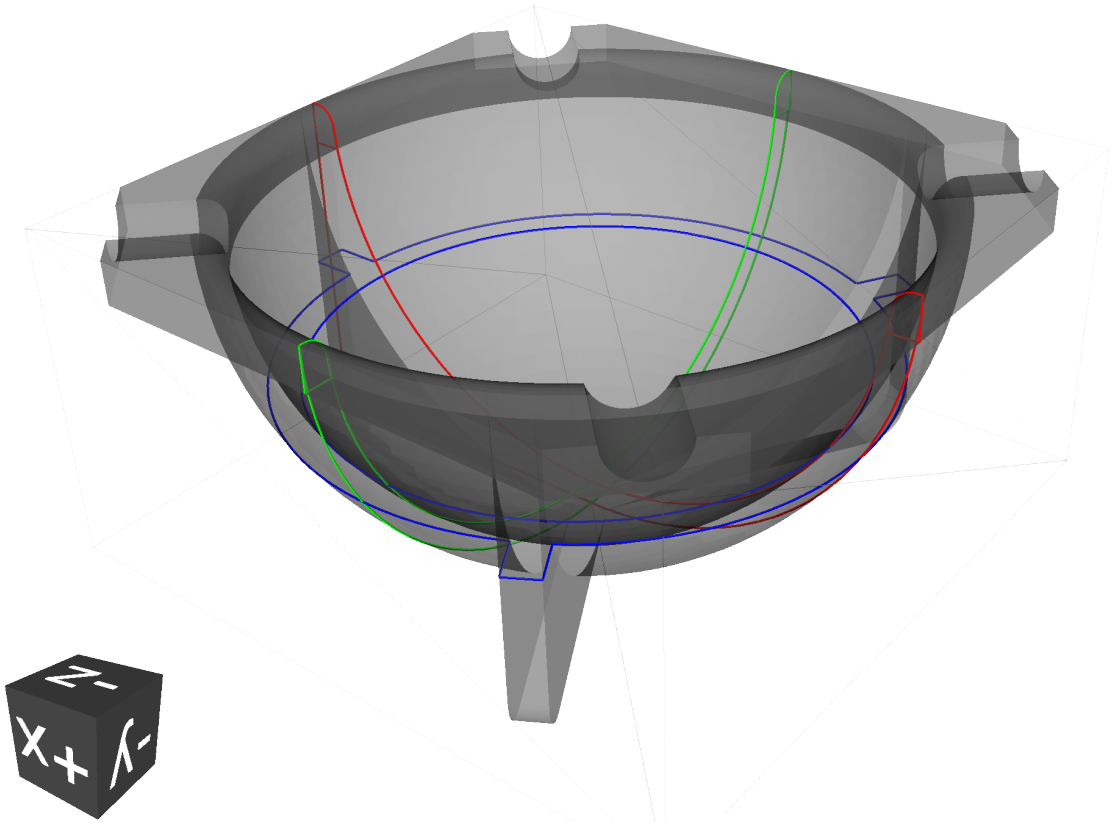
```
import pyg4ometry

r = pyg4ometry.freecad.Reader("08_AshTray.step")
r.relabelModel()
r.convertFlat()
l = r.getRegistry().getWorldVolume()
v = pyg4ometry.visualisation.VtkViewer()
```

(continues on next page)

(continued from previous page)

```
v.addLogicalVolume(l)
v.view()
```



1.4 Viewing Geometry

pyg4ometry uses the Visualisation Toolkit - VTK - to view geometry. This provides a window with a 3D view of a model that can be manipulated with a cursor (rotate, pan, zoom, click). Most manipulation can be done by calling methods on the instance of the visualiser class in the terminal.

Basic loading and viewing is described in *Loading Geometry*. Here, more detailed individual features are described.

1.4.1 Visualiser Classes

Several visualiser classes are provided but ultimately, the only difference is the default colouring of volumes.

- *pyg4ometry.visualisation.VtkViewer* - all in grey
- *pyg4ometry.visualisation.VtkViewerColoured* - user-provided dictionary of materials to colours
- *pyg4ometry.visualisation.VtkViewerColouredMaterial* - default dictionary of material colours included

Both `VtkViewerColoured` and `VtkViewerColouredMaterial` inherit `VtkViewer` and have the same functionality and differ only in colouring of volumes.

Generally:

```
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(lv)
v.view()  # or
v.view(interactive=False)  # to not block the terminal
```

For FLUKA, ROOT, STL, STEP visualisation, see *Loading Geometry* where each example has visualisation.

Exact documentation can be found in module-docs-visualisation.

If not adding a whole geometry tree, then individual solids can be added and overlaid with transparency for comparison purposes.

1.4.2 Using the Visualiser

Action	Outcome
click and drag	rotate the geometry
shift key + click and drag	pan (move) the geometry
right click	'pick' a volume and print out its name if found
scroll	zoom in and out

Rotating

Rotate by clicking and dragging, then release.

Zooming

Scroll in and out on a mouse or trackpad whilst pointing at the visualiser.

Panning And Rotating

Click and drag to rotate. Hold the shift key on the keyboard, then click and drag to pan.

When we rotate the geometry it may twist in multiple angles. To rotate in a specific way we can click and drag and draw it small circles where the geometry will precess.

Picking

If you right click on a volume and look at the terminal, if pyg4ometry can find a volume behind the point clicked it will print out the name.

1.4.3 Solid or Wireframe

When using the visualiser window, the same geometry can be viewed as solid surfaces or as a wireframe by pressing `s` key or the `w` key respectively.

Note, the original visualisation has the outermost volume as wireframe and the contents as solid. Once, the wireframe or solid option has been chosen, all volumes will have the same style.

1.4.4 Logical Volume

A `pyg4ometry.geant4.LogicalVolume` instance can be added to the visualiser. A logical volume has no concept of translation or rotation on its own, so it is placed in the centre of the visualiser coordinate system, i.e. in its own frame.

```
lv  # pyg4ometry.geant4.LogicalVolume instance
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(lv)
v.view()
```

It is possible to view the logical volume with an offset (i.e. translation) and rotation. This is purely for adding the scene of the viewer and does not affect the logical volume itself or anything it is used in. We can see the docstring:

```
>>> v = pyg4ometry.visualisation.VtkViewer()
>>> v.addLogicalVolume?
Signature:
v.addLogicalVolume(
logical,
mtra=matrix([[1, 0, 0],
[0, 1, 0],
[0, 0, 1]]),
tra=array([0, 0, 0]),
recursive=True,
)
```

If we start from a rotation as a series of Tait-Bryan angles, we can turn this into a matrix with:

```
import numpy as np
rotation = [0, np.pi/2, 0] # for example
matrix = np.linalg.inv(pyg4ometry.transformation.tbxyz2matrix(rotation))
l # a pyg4ometry.geant4.LogicalVolume instance
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(l, mtra=rotation, tra=[0,0,500])
```

Note: When directly using rotations and translations, the units are radians and mm.

If overlap checking has been used, this produces overlap meshes (if any) and these will be visualised automatically when visualising a LogicalVolume instance as they are associated with that instance.

1.4.5 Solid

It is possible to view an individual solid, i.e. any instance of a class in `pyg4ometry.geant4.solid` module.

```
s  # e.g. a pyg4ometry.geant4.solid.Box instance
v = pyg4ometry.visualisation.VtkViewer()
v.addSolid(s)
v.view()
```

Similarly to a logical volume, an individual solid has no concept of placement position and will by default be placed at the centre of the scene. It is also possible to add it to the scene with a rotation and translation.

```
>>> v.addSolid?
Signature:
v.addSolid(
solid,
rotation=[0, 0, 0],
position=[0, 0, 0],
representation='surface',
colour=[0.5, 0.5, 0.5],
opacity=0.2,
)
```

This uses Tait-Bryan angles for the rotation.

1.4.6 Boolean Solid

When creating geometry, it is common to use Boolean operations. Sometimes, we make mistakes in these and it is useful to understand the individual constituents even if the result is not a valid solid or mesh (i.e. completely disconnected solids). To do this we can visualise just a Boolean solid on its own.

```
s # e.g. a pyg4ometry.geant4.solid.Subtraction instance
v = pyg4ometry.visualisation.VtkViewer()
v.addBooleanSolidRecursive(s)
v.view()
```

This will work recursively for each solid that makes up the Boolean even if they are Booleans themselves. It will tolerate shapes that cannot form a valid mesh such as the resultant Boolean solid.

1.4.7 Default Colour Coding

With the `VtkViewer` class all volumes are visualised as semi-transparent grey.

1.4.8 Custom Colour Coding

With the `VtkViewerColoured` class, we can provide a default general colour and also a dictionary of specific colours for materials by name.

1.4.9 Random Colours

With the `VtkViewerColoured` class, we can supply the default colour as `"random"`, which will result in every volume being visualised with a random colour to be different.

```
v = pyg4ometry.visualisation.VtkViewerColoured(defaultColour="random")
```

1.4.10 Overlaying Two Geometries

In the visualiser we add "meshes" to the scene that are displayed. We are not restricted to make a physically accurate model and we can draw multiple meshes on top of each other by successively adding them to the scene.

Logical Volume Difference

The function *pyg4ometry.visualisation.viewLogicalVolumeDifference()* is provided that will view two `pyg4ometry.geant4.LogicalVolume` instances. It will also calculate the difference mesh between the two and visualise that also on top of the two with a different colour to highlight it.

1.4.11 Plotting Cutter Outlines From Files

Cutters are a feature in the visualiser to generate the usually red, green, blue lines that intercept the edges of the geometry in a given plane. By default, these are along each axis aligned with 0,0,0. These can be plotted or added to an existing plot as follows:

:: code-block:

```
pyg4ometry.visualisation.Plot.AddCutterDataToPlot("crosssection-ZX.dat",
                                                  "zx",
                                                  unitsFactor=0.001)
```

An optional Matplotlib axes instance can be given if there is one from an existing plot.

:: code-block:

```
f = matplotlib.pyplot.figure()
ax = f.add_subplot(111)
ax.plot([0,1],[0,1])
pyg4ometry.visualisation.Plot.AddCutterDataToPlot("crosssection-ZX.dat",
                                                  "zx", ax,
                                                  unitsFactor=0.001)
```

The full documentation can be found for *pyg4ometry.visualisation.Plot.AddCutterDataToPlot()*.

1.5 Validating Geometry

1.5.1 Overlap Checking

"Overlaps" is a general term used to describe malformed geometry. Such geometry is unphysical and may causing particle tracking problems in simulations such as stuck particles, or particles completely missing certain volumes entirely. Such errors are rarely easy to spot from results or running the simulation.

Given all the PVs (daughters) of a LV (mother) should be bounded by the LV/mother solid. It is possible to check between all daughter solid meshes and between daughters and the mother solid mesh. Given an `pyg4ometry.geant4.LogicalVolume` instance ("lv"), this check can be performed by calling the following code:

```
lv.checkOverlaps()
```

This will check only the immediate daughters of this logical volume. To descend further into a geometry, the recursive flag can be used:

```
lv.checkOverlaps(recursive=True)
```

See g4-module : `LogicalVolume.checkOverlaps()` for full details. A more complete example is:

```
# cd pyg4ometry/test/pythonGeant4
import pyg4ometry
r  = pyg4ometry.freecad.Reader("./T103_overlap_copl.gdml")
l = r.getRegistry().getWorldVolume()
l.checkOverlaps(recursive=False,coplanar=True,debugIO=False)
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(l)
v.view()
```

Text is by default only printed out when an overlap is found. Any overlaps will be prepared for visualisation in a VtkViewer (must be constructed and given the LV after this).

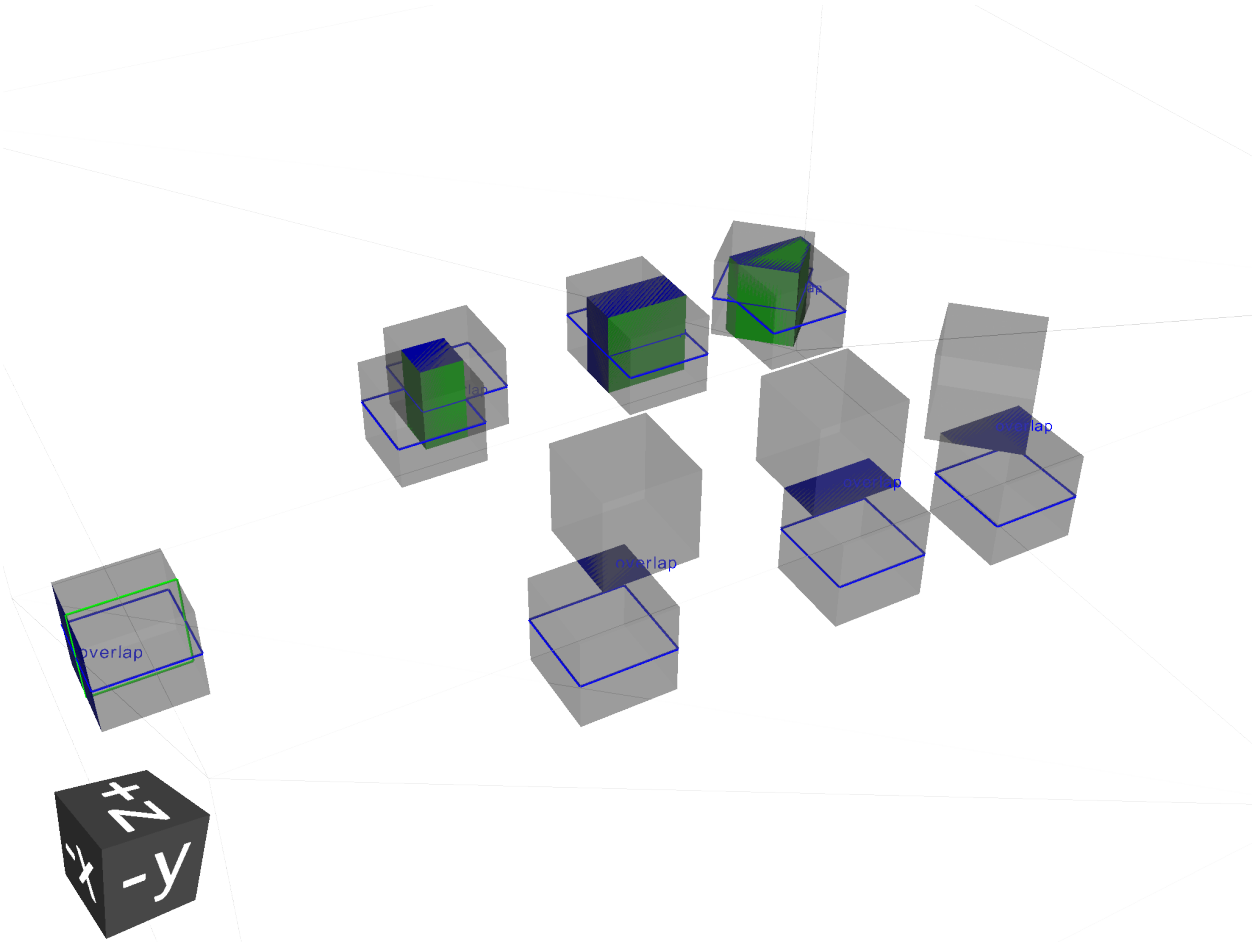
The following overlap checks are performed:

1. daughter with other daughter overlap
2. co-planar daughter with other daughter overlap
3. protrusion of a daughter from the mother volume
4. co-planar daughter with mother volume

Colour Coding

In the visualiser, text will be overlaid saying "overlap" where some kind of overlap is detected. Additionally, the actual overlap itself will be visualised and colour coded according to:

- red: protrusion overlap
- green: daughter-daughter overlap
- blue: co-planar overlap


overlap
overlap
overlap
overlap
overlap
+z
-y

Limitations

1. The overlap detection is performed by checking for overlaps in the visualisation meshes generated for each volume. In the case of curved solids (e.g. a cylinder), the mesh is not truly curved but a polygon. Very closely spaced curved surfaces may produce false overlaps. By default, all curved solids will use the same number of points around a circle, so usually we can "get away" with this if the curved solids aren't rotated about their axis.
2. Currently, division and parameterised volumes are not handled explicitly.

Assemblies

In the case of assembly volumes, and if an overlap is detected, a unique name is built up based on the parent PhysicalVolume, the assembly and the PhysicalVolume inside it. Furthermore, this is done recursively is assemblies of assemblies (etc) are used. The name is built up with an underscore "_" for padding and the user should decode this from their input.

As there is no 'mother' of an assembly, there is no mother protrusion directly. The contents of an assembly are compared to all other daughters and the mother at the higher level in which they are placed.

1.6 Exporting Geometry

1.6.1 Registry and GDML Output

Strictly speaking a registry class to store all of the GDML is not required. As with normal Geant4 given a `lv` pointer it should possible to form an aggregation hierarchy that contains all necessary objects. Now GDML breaks this as the structure is built up using `name` tags. For example a placement requires a position. In Geant4 this would just be a pointer to an transformation object, but GDML has two mechanisms to represent this, firstly child nodes of a PhysicalVolume tag or secondly a position define, see below

The registry class is a storage class for a complete GDML file. At the construction stage of almost all objects a registry is required. If the object is added to the registry then it will appear explicitly in the GDML output

1.6.2 GDML Output

To write an GDML file, a `pyg4ometry.geant4.registry` instance (`reg` here), must be supplied.

```
import pyg4ometry
w = pyg4ometry.gdml.Writer()
w.addDetector(reg)
w.write('file.gdml')
# make a quick bdsim job for the one component in a beam line
w.writeGmadTester('file.gmad', 'file.gdml')
```

1.6.3 Export scene to paraview/vtk

```
import pyg4ometry
```

1.6.4 Export scene to unity/unreal

The quickest way to get geometry to Unity/Unreal is to use a standard asset format. This takes a vtkRenderer and creates a OBJ file. The vtkRenderer managed within pyg4ometry from the VtkViewer class, once a geometry is created (either from any source) then an OBJ file can be created. Taking the example in `pyg4ometry/test/pythonCompoundExamples/`

```
import pyg4ometry
r = pyg4ometry.gdml.Reader("./Chamber.gdml")
l = r.getRegistry().getWorldVolume()
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(l)
v.exportOBJScene("Chamber")
```

`obj` files are written `Chamber.obj` and `Chamber.mtl`.

For a FLUKA file, first it must be converted to Geant4 and then the same process should be followed.

```
import pyg4ometry
r = pyg4ometry.fluka.Reader("./Chamber.inp")
greg = pyg4ometry.convert.fluka2Geant4(r.getRegistry())
l = greg.getWorldVolume()
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(l)
v.exportOBJScene("Chamber")
```

As the meshing might need to changed for the visualisation application, the parameters for the meshing for each solid might need to changed.

An `obj` file for an entire experiment does not help with work flows where meshes have to be UV-ed and textured. Tools like Blender and Gaffer can be used for this workload but require meshes for each object and their placement. To enable there is a special writer

```
import pyg4ometry

r = pyg4ometry.gdml.Reader("./Chamber.gdml")
l = r.getRegistry().getWorldVolume()
w = pyg4ometry.visualisation.RenderWriter()
w.addLogicalVolumeRecursive(l)
w.write("./SphericalChamber")
```

The directory `SphericalChamber` contains all the meshes in OBJ format along with an instance file `0_instances.dat` which contains a row for each instance of a mesh.

1.7 Creating Geometry

1.7.1 Precepts

- Units may be specified but default to Geant4 ones (e.g. mm, rad).
- Rotations are made using Tait-Bryan angles (rotation about reference x,y,z axes).
- A `Registry` object should be used to hold all things related in a model and passed into the constructors of most objects.
- GDML-like **full lengths** are used instead of typically half lengths

1.7.2 Units

- The default units are mm and rad for length and angle.
- Most constructors will take units as an optional key word argument ('kwarg').
- The kwargs are typically `lunit` for length unit and `aunit` for angle unit.
- Units are defined in `pyg4ometry.gdml.Units.py`.

The following units (as strings) are accepted:

Unit	Value
nm	1e-6
um	1e-3
mm	1
cm	10
m	1e3
km	1e6
deg	$\pi/180$
degree	$\pi/180$
rad	1
radian	1
mrad	1e-3
urad	1e-6
eV	1e-3
keV	1
MeV	1e+3
none	1
ns	1e-9
us	1e-6
ms	1e-3
s	1

Examples:

```
reg = pyg4ometry.geant4.Registry()
boxSolid = pyg4ometry.genat4.solid.Box("aBox", 10, 20, 30, reg)
```

This defines a box with the default units (none specified), so mm. We can specify them:

```
boxSolid = pyg4ometry.genat4.solid.Box("aBox", 10, 20, 30, reg, "cm")
```

1.7.3 Geant4 Python Scripting

Making use of pyg4ometry requires the following modules

```
import pyg4ometry
```

To make a simple geometry of a box located at the origin

```
# load pyg4ometry
import pyg4ometry

# registry to store gdml data
reg  = pyg4ometry.geant4.Registry()

# world solid and logical
ws   = pyg4ometry.geant4.solid.Box("ws",50,50,50,reg)
wl   = pyg4ometry.geant4.LogicalVolume(ws,"G4_Galactic","wl",reg)
reg.setWorld(wl.name)

# box placed at origin
b1   = pyg4ometry.geant4.solid.Box("b1",10,10,10,reg)
b1_l = pyg4ometry.geant4.LogicalVolume(b1,"G4_Fe","b1_l",reg)
b1_p = pyg4ometry.geant4.PhysicalVolume([0,0,0],[0,0,0],b1_l,"b1_p",wl,reg)

# visualise geometry
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(wl)
v.addAxes(20)
v.view()
```

Here is the vtk visualiser output of the above example

1.7.4 GDML Defines

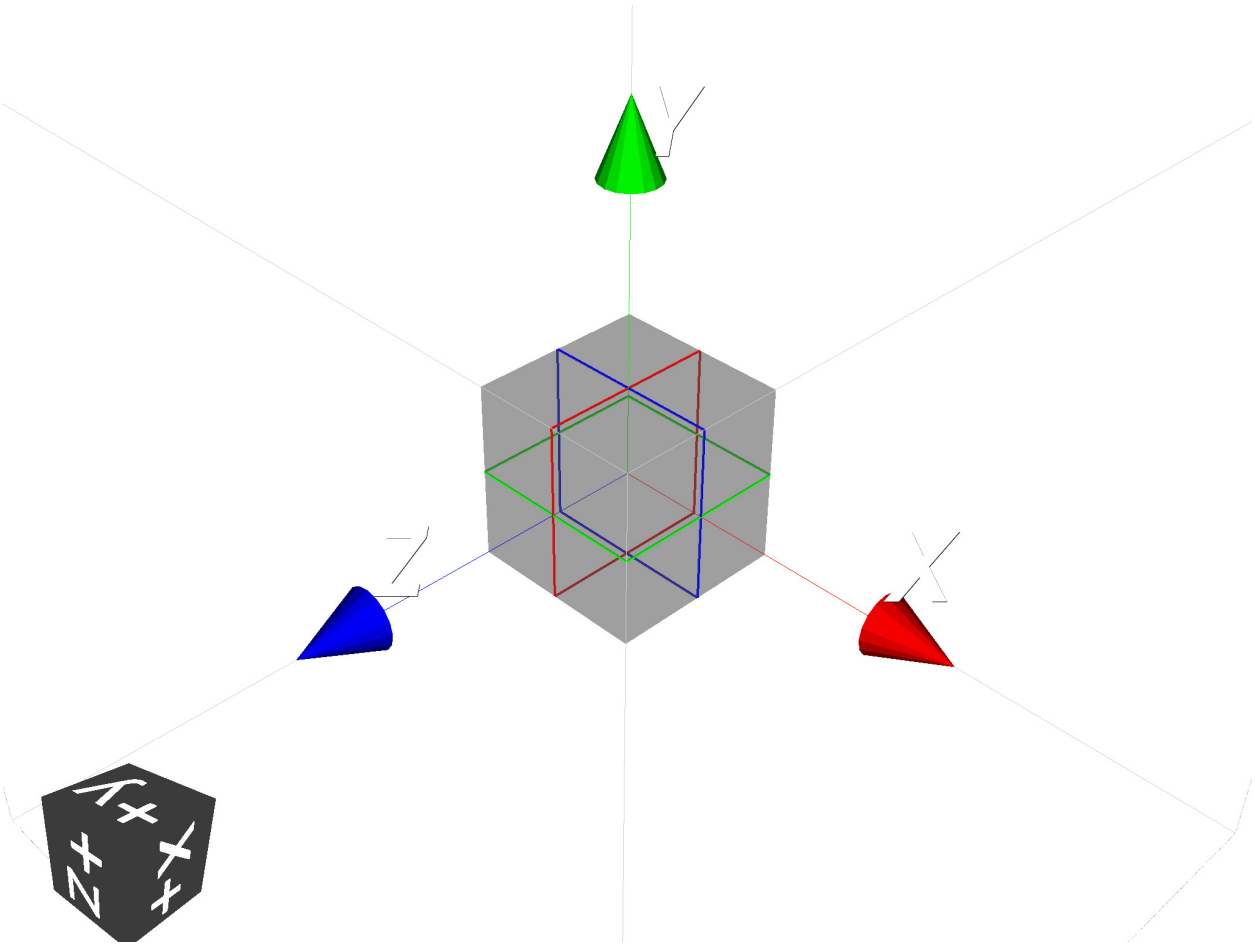
In GDML there are multiple `define` objects that can be used parameterise geometry, materials etc. These can be used as variables or definitions and mean that any equations used will be retained in GDML output. For example a GDML constant can be created in the following way

```
# registry to store gdml data
reg = pyg4ometry.geant4.Registry()

# constant called x
x = pyg4ometry.gdml.Constant("x",10,reg)
```

The normal set of mathematical operations in python can be performed and evaluated

```
y = 2*x + 10
y.eval()
```


Y+
Z-
X+
Z+
X-
Y-

```
>> 30
```

The constant `x` can of course be changed and `y` re-evaluated

```
x.setExpression(5)
y.eval()
```

```
>> 20
```

Note: Standard mathematical functions can be used with GDML defines (Constant, Variable, etc). So sin, cos, tan, exp and so on, but pyg4ometry functions have to be used

```
x  = pyg4ometry.gdml.Constant("x",10,reg)
cx = pyg4ometry.gdml.cos(x)
```

So the box example above can be rewritten using constants

```
# load pyg4ometry
import pyg4ometry

# registry to store gdml data
reg  = pyg4ometry.geant4.Registry()

bx = pyg4ometry.gdml.Constant("bx","10",reg,True)
by = pyg4ometry.gdml.Constant("by",2*bx,reg,True)
bz = pyg4ometry.gdml.Constant("bz",2*by,reg,True)

# world solid and logical
ws   = pyg4ometry.geant4.solid.Box("ws",50,50,50,reg)
wl   = pyg4ometry.geant4.LogicalVolume(ws,"G4_Galactic","wl",reg)

# box placed at origin
b1   = pyg4ometry.geant4.solid.Box("b1",bx,by,bz,reg)
b1_l = pyg4ometry.geant4.LogicalVolume(b1,"G4_Fe","b1_l",reg)
b1_p = pyg4ometry.geant4.PhysicalVolume([0,0,0],[0,0,0],b1_l,"b1_p",wl,reg)

# visualise geometry
v = pyg4ometry.visualisation.VtkViewer()
v.addLogicalVolume(wl)
v.addAxes(20)
v.view()
```

Note: All GDML defines (Constant, Variable, etc) can be used in the construction of other pyg4ometry classes interchangeably instead of floats or strings (where strings are either numbers or a GDML expression)

Warning: Avoid reassigning variables used as defines, this can have unexpected consequences so for example

```
b1 = pyg4ometry.geant4.solid.Box("b1", bx, by, bz, reg)
b1.pX = 20  # do not do this
```

```
b1.pX.setExpression(20)  # rather do this
```

1.7.5 Solids

The python geant4 solids match the Geant4 constructors as much possible (different constructor signatures are not supported in python). For example looking at the G4Box class

```
pyg4ometry.geant4.solid.Box(name, pX, pY, pZ, registry, lunit)
```

```
G4Box(const G4String& pName, G4double  pX, G4double  pY, G4double pZ)
```

A full list of solids can be found in all-solids.

Warning: The parameters stick to the GDML convention of **full** lengths opposed to half lengths.

1.7.6 Materials

As with solids materials are defined in a similar way to Geant4 C++. Python does not have overloaded constructors, so unique signatures are needed, in contrast to Geant4.

To define a material from the Geant4 predefined (e.g. NIST) materials

```
import pyg4ometry.geant4 as _g4
wm = _g4.MaterialPredefined("G4_Galactic")
bm = _g4.MaterialPredefined("G4_Fe")
```

To define a single element in terms of atomic number, atomic mass and density.

```
import pyg4ometry.geant4 as _g4
wm = _g4.MaterialSingleElement("galactic",1,1.008,1e-25,reg)    # low density hydrogen
bm = _g4.MaterialSingleElement("iron",26,55.8452,7.874,reg)     # iron at near room temp
```

To define a compound two elements using the mass fraction

```
import pyg4ometry.geant4 as _g4
wm = _g4.MaterialCompound("air",1.290e-3,2,reg)
ne = _g4.ElementSimple("nitrogen","N",7,14.01)
oe = _g4.ElementSimple("oxygen","O",8,16.0)
wm.add_element_massfraction(ne,0.7)
wm.add_element_massfraction(oe,0.3)
bm = _g4.MaterialSingleElement("iron",26,55.8452,7.874,reg)     # iron at near room temp
```

To define a compound using number of atoms

```
import pyg4ometry.geant4 as _g4
bm = _g4.MaterialCompound("plastic",1.38,3,reg)     # Generic PET C_10 H_8 O_4
he = _g4.ElementSimple("hydrogen","H",1,1.008)
ce = _g4.ElementSimple("carbon","C",6,12.0096)
oe = _g4.ElementSimple("oxygen","O",8,16.0)
```

(continues on next page)

(continued from previous page)

```
bm.add_element_natoms(he,8)
bm.add_element_natoms(ce,10)
bm.add_element_natoms(oe,4)
```

Material as a mixture of materials

```
import pyg4ometry.geant4 as _g4
bm     = _g4.MaterialCompound("YellowBrass_C26800", 8.14, 2, reg)
copper = _g4.MaterialPredefined("G4_Cu")
zinc   = _g4.MaterialPredefined("G4_Zn")
bm.add_material(copper, 0.67)
bm.add_material(zinc, 0.33)
```

Example of elements formed by isotopes

```
import pyg4ometry.geant4 as _g4
u235 = _g4.Isotope("U235", 92, 235, 235.044)
u238 = _g4.Isotope("U238", 92, 238, 238.051)
uranium = _g4.ElementIsotopeMixture("uranium", "U", 2)
uranium.add_isotope(u235, 0.00716)
uranium.add_isotope(u238, 0.99284)
bm = _g4.MaterialCompound("natural_uranium", 19.1, 1, reg)
bm.add_element_massfraction(uranium, 1)
```

NIST Materials

Geant4 has many predefined materials according to the NIST database. Their name typically starts with `G4_`. These typically can be used with `MaterialPredefined` and we **do not need** to specify the full composition - Geant4 will find them at run time.

However, in the case of conversion to FLUKA, these are fully expanded according to their definition in Geant4 based on a cache in pyg4ometry of the material compositions generated using BDSIM from Geant4 (10.7.p01 as of writing). Should the user wish to use these, they can be accessed from the functions in the geant4 module.

```
import pyg4ometry
nistHydrogenElement = pyg4ometry.geant4.nist_element_2geant4Element('G4_H')
```

Note, an 'element' cannot be used as a 'material' in a logical volume. We must upgrade it to a material for that. The NIST elements contain the appropriate mixture of natural isotopes and can be used in `MaterialCompound` as demonstrated above.

Alternatively, we can access the NIST materials and materials of elements.

```
import pyg4ometry
nistHydrogenMaterial = pyg4ometry.geant4.nist_material_2geant4Material('G4_H')
nistConcreteMaterial = pyg4ometry.geant4.nist_material_2geant4Material('G4_CONCRETE')
```

1.7.7 Detector Construction

This largely proceeds in exactly the same way as in G4 or GDML. Hierarchy of solids, booleans, logical, physical (replica, division, param) volumes.

0. Create registry to hold everything
1. Create solids
2. Create logical volumes
3. Place logical volumes (construct physical volumes)
4. Visualise
5. Check
6. Export

1.7.8 Transformations & Physical Volumes

Transformations in 3D are essential for the easy placement of solids in a CSG tree or LV placement. There is not a specific transformation class like in Geant4. The matrices and vectors used for placements are here typically Numpy arrays or matrices.

Geant4 has two possible constructors for a physical volume. These provide active and passive transformations. In pyg4ometry, only one is provided.

- The transform in a physical volume first translates the placed logical volume with respect to the mother logical, then rotates it.

The physical volume class is documented here: g4-module, but an example is shown here.

```
import pyg4ometry

r = pyg4ometry.geant4.Registry()
vacuum = _g4.MaterialPredefined("G4_Galactic")
water = _g4.MaterialPredefined("G4_WATER")
worldSolid = pyg4ometry.geant4.solid.Box("world_solid", 100, 100, 100, reg)
boxSolid = pyg4ometry.geant4.solid.Box("box_solid", 10, 20, 40, reg)
worldLV = pyg4ometry.geant4.LogicalVolume(worldSolid, vacuum, "world_lv", reg)
boxLV = pyg4ometry.geant4.LogicalVolume(boxSolid, water, "box_lv", reg)

pyg4ometry.geant4.PhysicalVolume([0, 0, 0], [0, 0, 0], boxLV, "box_pv", worldLV, reg)
```

This creates a box of water inside a box of vacuum. The box of water is 10 x 20 x 50 mm long (note mm are the default length units), and it is placed with no offset and no rotation (i.e. at the centre) of the world volume. Alternatively:

```
import numpy as np

pyg4ometry.geant4.PhysicalVolume(
    [0, np.pi / 3.0, 0], [0, 0, 0], boxLV, "box_pv", worldLV, reg
)
```

In this case, the box is placed with no offset but with a rotation of $\pi/3$ radians about the y axis of the world box.

Note: The rotations are Tait-Bryan angles, which are rotations about the reference x,y,z axes. i.e. if there is a rotation about both x and y, these are independent and it is **not** a compound frame that is rotated. These are commonly thought

of like an aircraft and called pitch, yaw and tilt.

There are utility functions for translation between different transformations in `pyg4ometry.transformation`. See transformation-module.

1.7.9 Optical Surfaces

Optical surfaces can be created in a similar way as in Geant4 C++. A *pyg4ometry.geant4.solid.OpticalSurface* instance holds all the needed properties of the surface (including extra properties, e.g. for optical processes). This is then assigned to the surface between either

- two physical volumes: *pyg4ometry.geant4.BorderSurface*, or
- a logical volume and all its neighbouring volumes: *pyg4ometry.geant4.SkinSurface*.

```
opa = _g4.solid.OpticalSurface(
    "AirSurface",
    finish="polished",
    model="glisur",
    surf_type="dielectric_dielectric",
    value="1",
    registry=reg,
)
opw = _g4.solid.OpticalSurface(
    "WaterSurface",
    finish="ground",
    model="unified",
    surf_type="dielectric_dielectric",
    value="0",
    registry=reg,
)

_g4.SkinSurface("AirSurface", air_lv, opa, reg)
_g4.BorderSurface("WaterSurface", water_phys, world_phys, opw, reg)
```

1.7.10 Properties of Materials and Optical Surfaces

Materials and optical surfaces support adding properties that can be used by Geant4 to influence processes, e.g. for scintillation, refraction or other optical processes.

In the GDML, a matrix is used to hold the value(s) of the property.

- `addProperty(name, matrix)` - Add a property based on an existing *pyg4ometry.gdml.Matrix* object.
- `addVecProperty(name, e, v, eunit='eV', vunit='')` - Add a property based on a energy vector and a value vector.
- `addConstProperty(name, value, vunit='')`- Add a property that has only one constant value.

Units can be specified by setting the parameters `eunit` for the energy vector and `vunit` for the values. The given vectors are expected to be homogeneous in their units.

Note: Optical properties can only use units (or combinations of units) that are also defined in pyg4ometry. If needed, additional units can be added: `pyg4ometry.gdml.Units.units['ps'] = 1e-12`.

```
scint = _g4.Material(...)
scint.addConstProperty("SCINTILLATIONTIMECONSTANT1", 2.5, vunit="ns")
scint.addConstProperty("SCINTILLATIONYIELD", 8000, vunit="/MeV")
scint.addVecProperty("RINDEX", [1, 10], [1.3, 1.05])
```

1.7.11 FLUKA Geometry Creation

In a very similar way to geant4 geometry authoring it is possible to use pyg4ometry to create fluka output. To create a simple region consisting of a single body

```
import pyg4ometry.convert as convert
import pyg4ometry.visualisation as vi
from pyg4ometry.fluka import RPP, Region, Zone, FlukaRegistry

freg = FlukaRegistry()

rpp = RPP("RPP_BODY", 0, 10, 0, 10, 0, 10, flukaregistry=freg)
z = Zone()
z.addIntersection(rpp)
region = Region("RPP_REG", material="COPPER")
region.addZone(z)
freg.addRegion(region)

greg = convert.fluka2Geant4(freg)
greg.getWorldVolume().clipSolid()

v = vi.VtkViewer()
v.addAxes(length=20)
v.addLogicalVolume(greg.getWorldVolume())
v.view()
```

1.8 Converting Geometry

1.8.1 Assembly Conversion

It is possible transform a logical volume into an assembly volume. This is useful in the case of two sources of geometry: placement of top level world logical volume solids will likely result in an overlap. This effectively removes the outermost logical volume but keeps the daughters. The produced assembly volume can then be placed or imprinted somewhere in the geometry.

An assembly **cannot** be used as an outermost 'world' volume for a geometry hierarchy.

Assuming `lv` is a `pyg4ometry.geant4.LogicalVolume` instance:

```
av = lv.assemblyVolume()
```

1.8.2 GDML to FLUKA

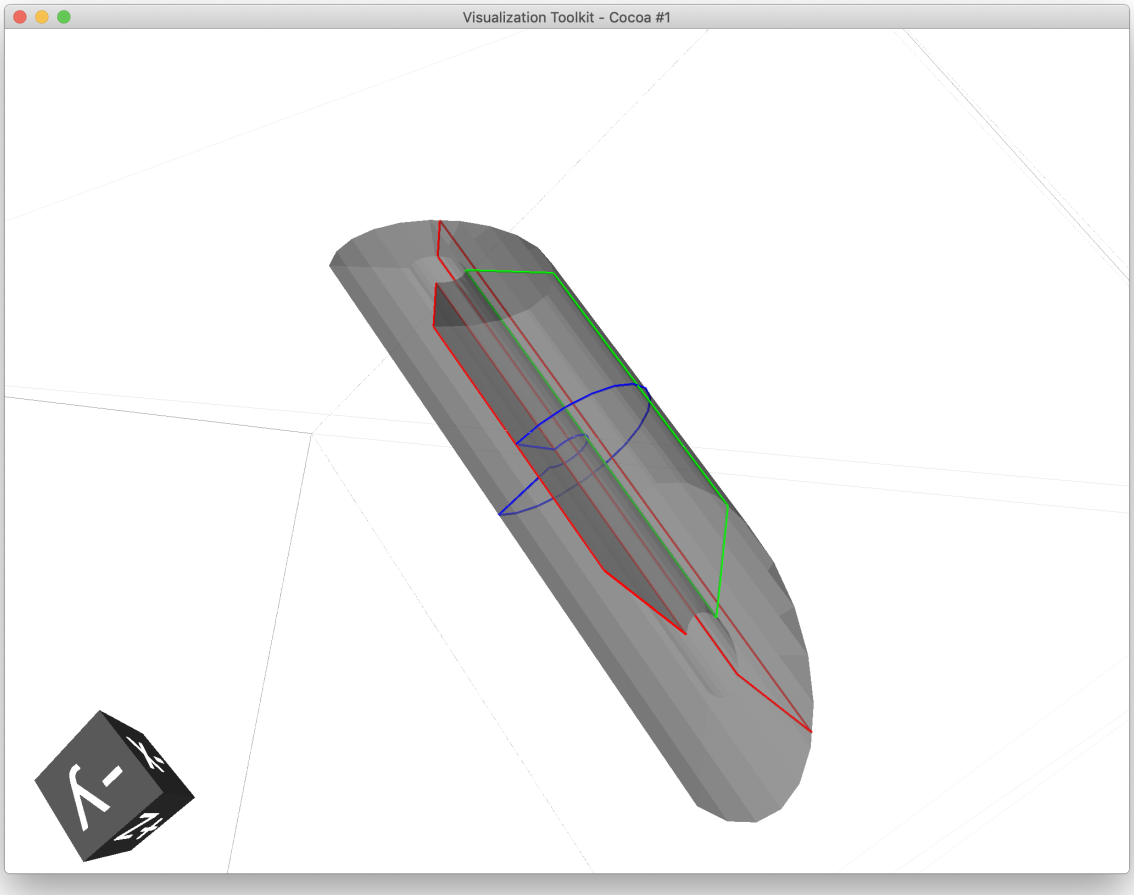
It is possible convert a pyg4ometry geometry to FLUKA. This is currently a work in progress and not all Geant4-GDML constructions are implemented, although they can be quickly added. Given a LV variable named `logical`

```
import pyg4ometry
reader = pyg4ometry.gdml.Reader("input.gdml")
logical = reader.getRegistry().getWorldVolume()
freg = pyg4ometry.convert.geant4Logical2Fluka(logical)
w = pyg4ometry.fluka.Writer()
w.addDetector(freg)
w.write("FileName.inp")
```

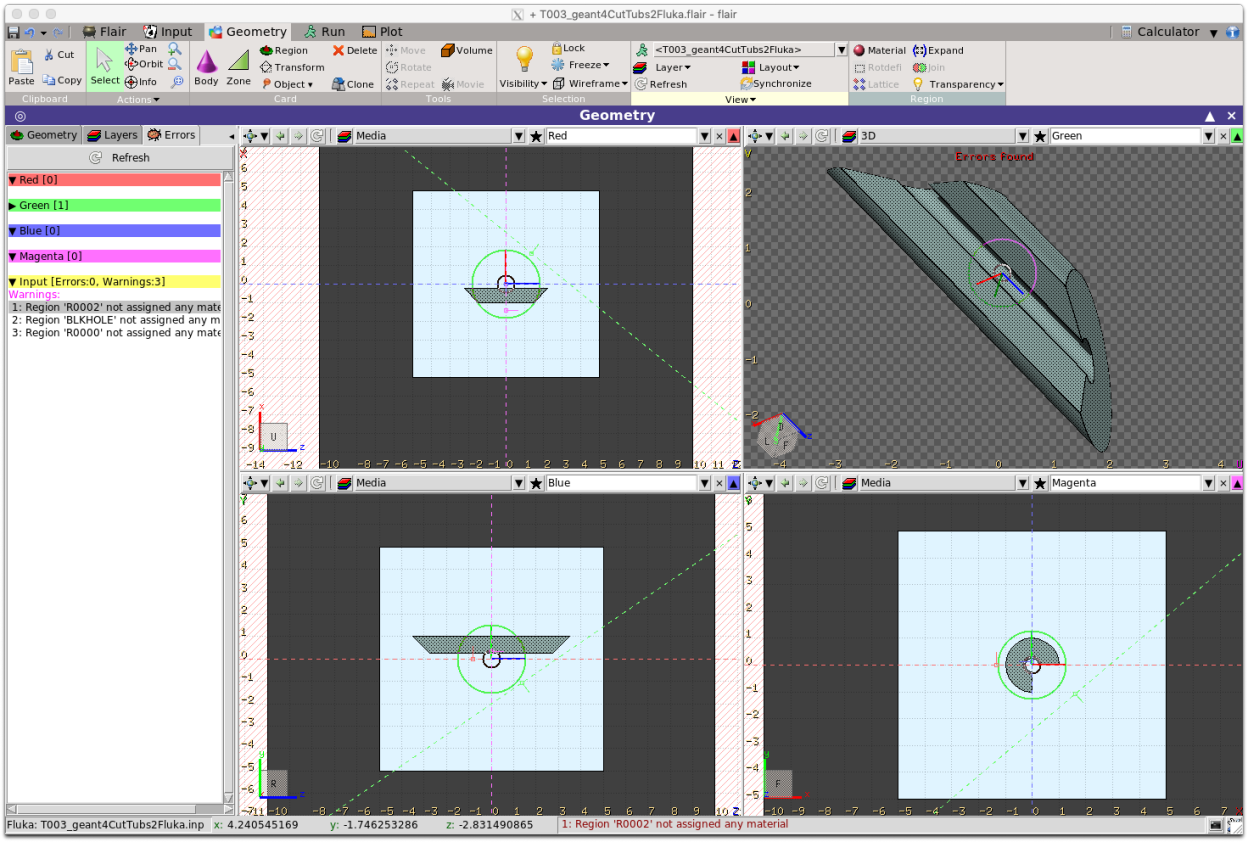
If you want to load a file into Flair then a flair file can be written based on `FileName.inp` using the following

```
extent = logical.extent(includeBoundingSolid=True)
f = pyg4ometry.fluka.Flair("FileName.inp",extent)
f.write("FileName.flair")
```

Here is an example (viewed in Flair) of a simple Geant G4 solid that has been converted to FLUKA using this method



Note: All GDML placements are respected in the conversion from GDML to FLUKA, for both Placements and



Boolean Solids. So for example a tree of LV-PV placements are reduced into a single transformation of a LV into a global coordinate space for FLUKA. A similar process is used for a tree of CSG operations.

Warning: Currently there are some things which are not implemented in the conversion. 1) Materials, 2) Scaled solids, 3) Reflections in placements, 4) Division, replica and parameterised placements. Some of these are straight forward to implement, like Materials and the non-Placement physical volumes can be done quickly if a user requires it.

1.8.3 FLUKA To GDML

FLUKA geometry can be converted to GDML using `pyg4ometry.convert.fluka2geant4`. The conversion process is robust and supports all FLUKA geometry constructs. Given a FLUKA file *model.inp*, the following code can be used to translate it to a GDML file.

```
import pyg4ometry.fluka as fluka
import pyg4ometry.gdml as gdml
from pyg4ometry.convert import fluka2Geant4

# Read the FLUKA file, get the FlukaRegistry, convert the registry to a
# Geant4 Registry
reader = fluka.Reader("model.inp")
flukaregistry = reader.flukaregistry
```

(continues on next page)

(continued from previous page)

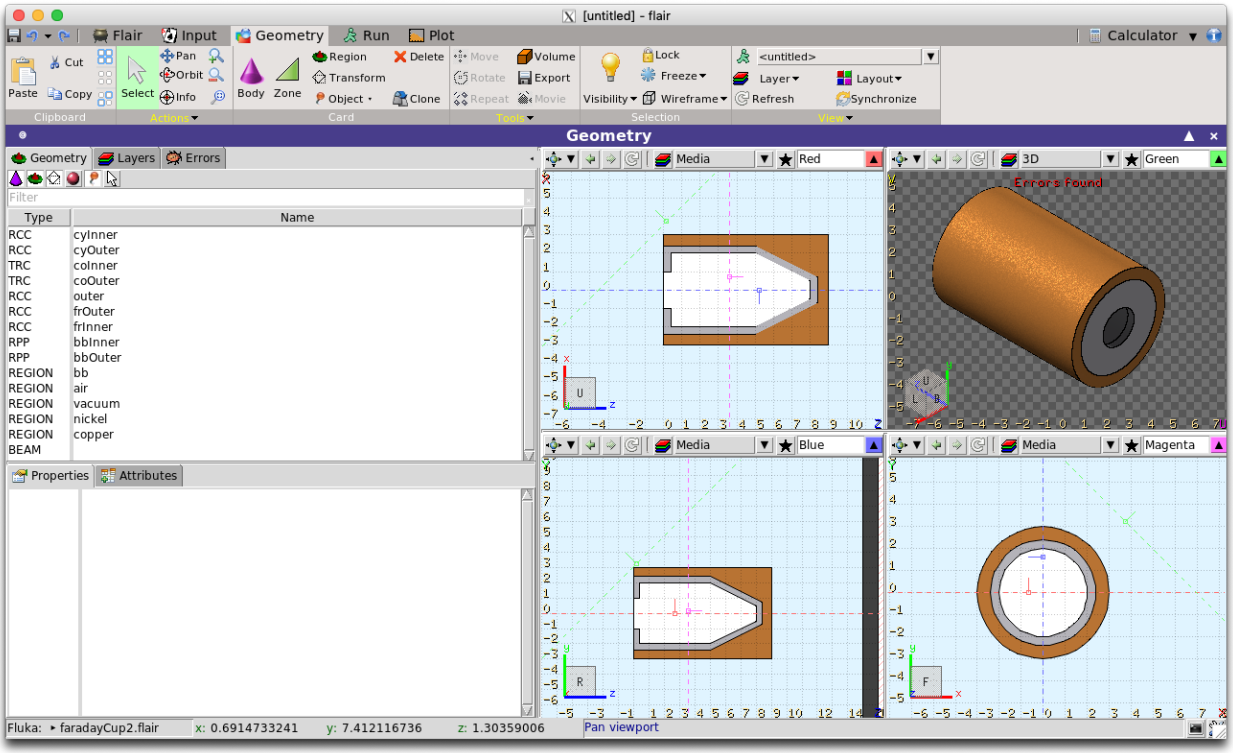
```
geant4Registry = fluka2Geant4(flukaRegistry)

worldLogicalVolume = geant4Registry.getWorldVolume()
worldLogicalVolume.clipSolid()

writer = gdml.Writer()
writer.addDetector(geant4Registry)
writer.write("model.gdml")
```

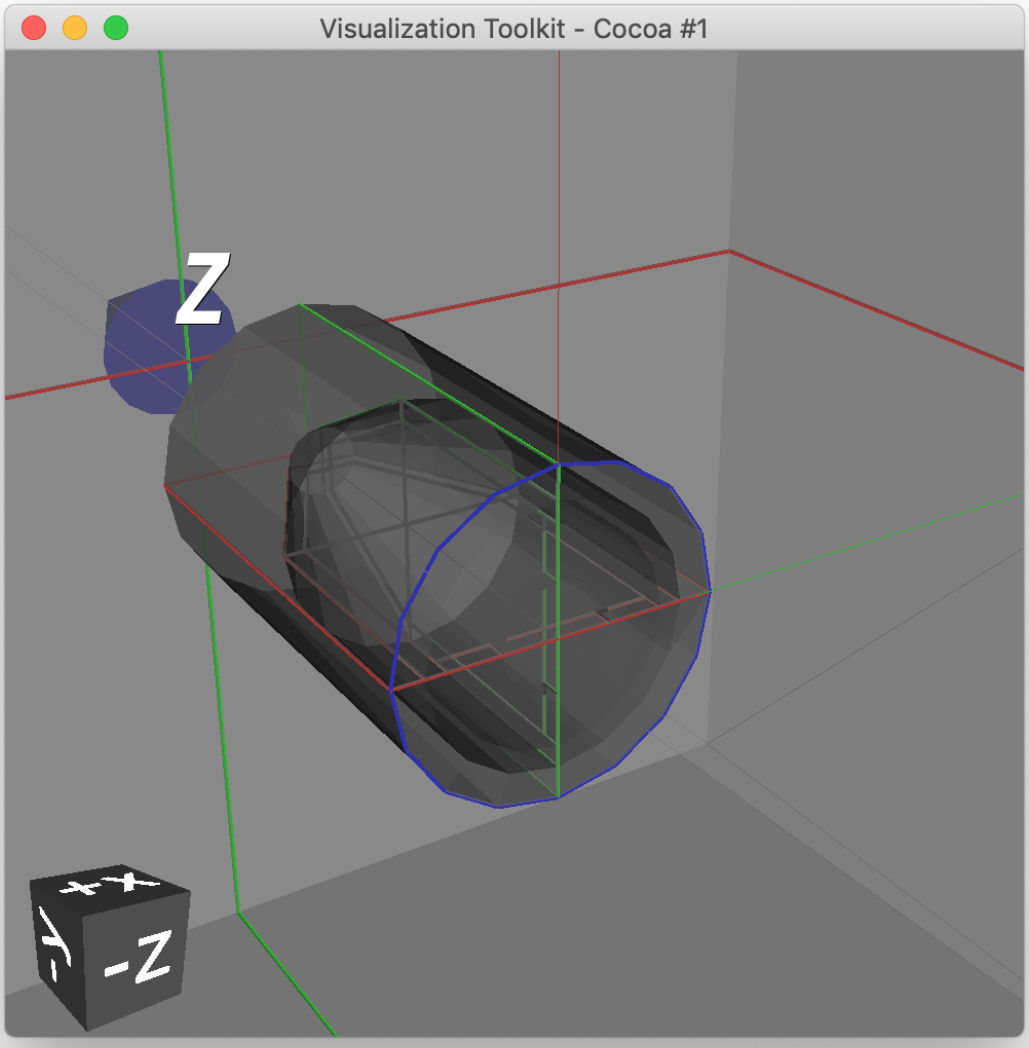
The core of this functionality is the translation of the *FlukaRegistry* instance into the equivalent *Registry* (i.e. Geant4) instance.

Here is an example of a model viewed in flair and the resulting visualisation in VTK of the Geant4 model



A number of keyword arguments are available to further modify the conversion. The *fluka2Geant4* keyword arguments *region* and *omitRegions* allow the user to select a subset of the named regions to be translated.

The conversion of QUA bodies (fluka2geant4 kwarg *quadricRegionAABBs*) is complex and requires further explanation. In Pyg4ometry the mesh and GDML representations of FLUKA infinite circular cylinders, elliptical cylinders and half-spaces are all finite (but very large) cylinders, elliptical cylinders and boxes. This is robust as increasing the length of cylinders and depth/breadth of boxes does not increase the number of polygons used in the underlying mesh representation for that solid. However, this is not true of the quadric surface. A quadric surface cannot simply be generated to be "very large", as the number of polygons will grow quickly, along with the memory consumption and facets in the resulting GDML TesselatedSolid, which will also slowing down tracking time in Geant4. For this reason the user must provide axis-aligned bounding boxes of the regions where any QUA bodies are present. It is recommended that these boxes be a centimetre larger than formally necessary to ensure a correct conversion. Providing the bounding box ensures that an efficient and accurate mesh of the QUA bodies can be generated meaning that the conversion to be performed in a tractable amount of time as well giving more performant tracking in Geant4.


Visualization Toolkit - Cocoa #1
Z
+X
-Z

1.9 Combining Geometry

There are ways to incorporate geometry from multiple sources in GDML. This has potentially lots of problems as each file needs to be a well formed GDML file and care has to be taken with degenerate names from the different sources. For example a volume can be extracted from GdmlFile1 and added to GdmlFile2, clearly all solids, materials and variables need to also transferred. For this example we need two GDML files, so `pyg4ometry/test/pythonGeant4/T001_Box.py` and `pyg4ometry/test/pythonGeant4/T002_Tubs.py`, so run them

```
import T001_Box
T001_Box.Test(True,True)

import T002_Tubs
T002_Tubs(True,True)
```

This will create two GDML files `T001_Box.gdml` and `T002_Tubs.gdml`. It is possible to find the volumes contained in each file (using the tubs gdml file as the example) by performing the following

```
import pyg4ometry
r = pyg4ometry.gdml.Reader("T002_Tubs.gdml")
reg = r.getRegistry()

# printing the names of the logical volumes
print(reg.logicalVolumeDict.keys())

# printing the names of the physical volumes
print(reg.physicalVolumeDict.keys())

lv = reg.logicalVolume["tl"]
```

Now merging the `tl` logicalVolume (which is a simple tubs) with the box gdml file

```
import pyg4ometry
r1 = pyg4ometry.gdml.Reader("T001_Box.gdml")
reg1 = r1.getRegistry()

r2 = pyg4ometry.gdml.Reader("T002_Tubs.gdml")
reg2 = r2.getRegistry()

lv = reg2.logicalVolumeDict["tl"]

# create physical volume with placement
pv = pyg4ometry.geant4.PhysicalVolume([0,0,0],[50,0,0], lv, "tl_pv", reg1.
↪getWorldVolume(), reg1)

reg1.addVolumeRecursive(pv)

# gdml output
w = pyg4ometry.gdml.Writer()
w.addDetector(reg1)
w.write("MergeRegistry.gdml")
```

Note: In the example two registry objects are created and objects from reg2 are merged into reg1. Of course one

registry might be formed by pyg4ometry commands opposed created from a file.

Warning: The pv needs to added with addVolumeRecursive otherwise it is possible that GDML definitions which lv depends on are not transferred over.

1.10 Comparing Geometry

When converting or preparing geometry, it is useful to compare to other geometry to validate work. Examples of use include:

- validating changes to existing pyg4ometry scripts
- validating geometry created in pyg4ometry versus another source
- comparing different representations
- validating conversion between formats

pyg4ometry provides a set of tests for comparing two geometry "trees" that are loaded into memory. Each hierarchy of geometry is navigated and a variety of user-selectable tests are carried out. A result class with details of tests carried out and their outcome is produced and can be easily printed or inpsected programmatically.

When we compare 2 things, we use the terminology **reference** and **other** for each. When we need to identify a pair, we use the reference object name. Reference is the object that is the control, or assumed to be the 'right' one.

1.10.1 Tests

Which tests to conduct are defined by an instance of `pyg4ometry.compare.Tests`, which has a set of Boolean options for each test.

The available tests are described below.

Test Name	Description
names	Test whether objects have the exact same name.
namesIgnorePointers	Test whether objects have the exact same name, but whilst ignorign pointer suffixes such as 0x1234567. Note, the geometry will load ok, the name stripping is only for comparison.
nDaughters	Test for a matching number of daughter volumes in a LogicalVolume or AssemblyVolume.
solidExact	Compare the class type of every solid.
shapeExtent	Numerically compare the size of the bounding box (i.e. extent) of each solid.
shapeVolume	Compare the volume as per the visualisation mesh of each shape.
shapeArea	Compare the surface area as per the visualisation mesh of each shape.
placement	Numerically compare the rotation and translations in each placement.
scale	Compare the 'scale' for a physical volume that can be used for a reflection. No scale is equal to unit scale [1,1,1].
copyNumber	Compare the copy number for a physical volume.
materials	Whether to do a series of tests of materials in a LogicalVolume.
materialClassType	If doing materials tests, compare the class name of each material.
materialCompositionType	If doing materials tests, compare the class name of each component of a material.
testDaughtersByName	Whether to check daughter volumes by name or by index. If true, then a set of overlapping names is used and the matching ones that exist in both are compared. If false, the daughters are strictly compared in numerical order.

Note: Volumes and areas are based on the polygonal meshes generated for visualisation purposes. These may not be the exact area of a perfect shape, such as a sphere, but an approximation. They will however, be consistently generated. Consider the tolerances also.

Tolerances

When making numerical comparisons (e.g. position of a volume), we must consider the finite numerical precision in a computer. Also, this may be useful as geometry from another source may be practically equivalent, but not exactly at the last decimal place.

Therefore, we include a set of tolerances for numerical comparisons that are included in the Tests class. These are typically fractional.

Note: The fraction is calculated with respect to the reference value. So $(value_{other} - value_{reference})/value_{reference}$.

Variable Name	Default Value	Description
toleranceSolidParameterFraction	1e-3	Maximum fractional difference for any solid parameter.
toleranceSolidExtentFraction	1e-6	Maximum fractional difference for the calculated extent of any solid.
toleranceVolumeFraction	1e-2	Maximum fractional difference in volume.
toleranceAreaFraction	1e-2	Maximum fractional difference in area.
toleranceTranslationFraction	1e-6	Maximum fractional difference for translations in placements.
toleranceScaleFraction	1e-3	Maximum fractional difference in scale parameters in placements.
toleranceRotationFraction	1e-6	Maximum fractional difference in rotation angles in placements.
toleranceMaterialDensityFraction	1e-4	Maximum fractional difference in the density of a material.
toleranceMaterialMassFraction	1e-4	Maximum fractional difference in the mass fraction of a material component.

1.10.2 Running a Comparison

See also geometry-compare-module for all functions.

A comparison is run by using a function in `pyg4ometry.compare`, giving it two objects and a Tests instance. The comparison result returned is a class with a list of test results that can be inspected and also printed.

GDML Files

```
>>> comparison = pyg4ometry.compare.gdmlFiles("file_reference.gdml", "file_other.gdml")
>>> comparison.print()
```

Logical Volumes

Compare two logical volumes, such as the top volume of a registry from a loaded file.

```
>>> comparison = pyg4ometry.compare.geometry(refLV, otherLV)
>>> comparison.print()
```

There are functions to compare individual objects too. These follow the same pattern. A few are:

- `pyg4ometry.compare.physicalVolumes`
- `pyg4ometry.compare.assemblyVolumes`
- `pyg4ometry.compare.replicaVolumes`
- `pyg4ometry.compare.divisionVolumes` - not implemented yet
- `pyg4ometry.compare.parameterisedVolumes` - not implemented yet
- `pyg4ometry.compare.materials`
- `pyg4ometry.compare.solids`

Example Output

For a test that failed the following is an example of output.

```
Overall result>  TestResult.Failed
Test>  position
(av): a_assembly: (pv): a_a_pv1: a_a_pv1_pos: TestResult.Failed: z: (reference): 100.0,␣
↪(other): -100.0

Test>  shapeExtentBoundingBoxMin
a_assembly_a_a_pv1: TestResult.Failed: axis-aligned bounding box lower edge: dimension:␣
↪z, (reference): 85.0, (other): -115.0

Test>  shapeExtentBoundingBoxMax
a_assembly_a_a_pv1: TestResult.Failed: axis-aligned bounding box upper edge: dimension:␣
↪z, (reference): 115.0, (other): -85.0
```

1.10.3 Seeing the Results

The return of a comparison is a `pyg4ometry.compare.ComparisonResult` instance. The most useful function here is the print function, which will print all results but also you can print select sets of tests or only certain results.

- See geometry-compare-module and `ComparisonResult` for details.

1.10.4 Examples

See `pyg4ometry/test/pythonGeant4/T7*.py` for examples that form the tests of this code.

Only Volume

Here, two ways are shown for creating the set of tests.

```
>>> import pyg4ometry
>>> t = pyg4ometry.compare.Tests()
>>> pyg4ometry.compare.Tests.printAllTestNames()
    "names"
    "namesIgnorePointer"
    "nDaughters"
    "solidExact"
    "solidExtent"
    "shapeExtent"
    "shapeVolume"
    "shapeArea"
    "placement"
    "scale"
    "copyNumber"
    "materials"
    "materialClassType"
    "materialCompositionType"
    "testDaughtersByName"
>>> t.setAllFalse()
```

(continues on next page)

(continued from previous page)

```
>>> t.shapeVolume = True
>>> comparison = pyg4ometry.compare.gdmlFiles("file1.gdml", "file2.gdml", t)
>>> comparison.print()
```

or

```
>>> import pyg4ometry
>>> t2 = pyg4ometry.compare.Tests("shapeVolume")
>>> comparison = pyg4ometry.compare.gdmlFiles("file1.gdml", "file2.gdml", t2)
>>> comparison.print()
```

Removing a Test

A test can be turned off by name:

```
>>> import pyg4ometry
>>> t = pyg4ometry.compare.Tests()
>>> t.setFalse("names")
```

1.11 Analysing Geometry

1.11.1 Finding volumes

Before editing geometry it is useful to find a logical volume. The registry contains all the logical volumes using the GDML in `pyg4ometry/test/gdmlG4examples/ChargeExchangeMC/`

```
1 import pyg4ometry
2 r = pyg4ometry.gdml.Reader("lht.gdml")
3 reg = r.getRegistry()
```

The `registry` instance `reg` has a member variable called `logicalVolumeDict` so calling

```
reg.logicalVolumeDict.keys()
```

should print

```
In [4]: reg.logicalVolumeDict.keys()
Out[4]: odict_keys(['vMonitor', 'vMonitorBack', 'vTarget', 'vTargetInnerCover',
↪'vTargetColumn', 'vTargetInnerColumn',
                    'vTargetVacuumSpace', 'vTargetOuterCover', 'vCrystal', 'vCrystalRow',
↪ 'vCalorimeter', 'vVetoCounter',
                    'vOuterFerrumRing', 'vInnerFerrumRing', 'vInnerCuprumRing',
↪'vTargetWindow', 'vTargetWindowCap',
                    'vTargetWindowMylarCover', 'vTargetWindowAluminiumCover',
↪'vWorldVisible', 'World'])
```

then the LogicalVolume can be obtained simply from the dictionary

```
lv = reg.logicalVolumeDict['vTargetInnerColumn']
```

This `lv` can be used for manipulating geometry, passing to visualisers etc.

1.11.2 Navigating the LV-PV hierarchy

There is a hierarchy of LV-PVs to describe a GDML/Geant4 geometry. An LV in terms of geometry consists of an outer solid `lv.solid` and `lv.daughterVolumes`. `lv.solid` is one of the `pyg4ometry.geant4.solid` types which match the GDML/Geant4 solids. `lv.daughterVolumes` is a list of `pyg4ometry.geant4.PhysicalVolumes`.

The best way to explore the methods and data members of `pyg4ometry.geant4.LogicalVolume` and `pyg4ometry.geant4.PhysicalVolume` is to explore in iPython. See cute video based on the `lht.gdml` example above.

1.11.3 Geometry Complexity Analysis

For a given logical volume we can get some statistics on the complexity of the geometry. A simple class called *GeometryComplexityInformation* is returned that has a serious of dictionaries with information.

```
cd pyg4ometry/test/gdmlCompoundExamples/bdsim_2
ipython
>>> import pyg4ometry
>>> r = pyg4ometry.gdml.Reader("22-size-variation-facetcrop-quad.gdml")
>>> info = pyg4ometry.geant4.AnalyseGeometryComplexity(r.getRegistry().getWorldVolume())
>>> info.printSummary()
Types of solids
ExtrudedSolid        : 96
Tubs                 : 51
Intersection         : 24
Polyhedra            : 12
Subtraction          : 6
Box                  : 1

# of daughters       count
0                    : 152
2                    : 19
4                    : 12
13                   : 6
25                   : 1

Depth of booleans    count
1                    : 30

Booleans width depth over  3
Solid name                                : n Booleans

>>> info. <tab>
comp.booleanDepth      comp.nDaughtersPerLV
comp.booleanDepthCount comp.printSummary
comp.nDaughters        comp.solids
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

REFERENCING & CITATION

Important: Citation information can be obtained on GitHub by pointing your browser to github.com/g4edge/pyg4ometry and selecting "Cite this repository" in the sidebar on the right. pyg4ometry uses a CITATION.cff file.

Citations allow the authors to demonstrably show the code usage and therefore to continue to support the development and maintenance of pyg4ometry.

Any publications including simulations made using pyg4ometry **must** cite:

> S.D. Walker, A. Abramov, L.J. Nevay, W. Shields, S.T. Boogert, *"pyg4ometry: A Python library for the creation of Monte Carlo radiation transport physical geometries"*, Computer Physics Communications **272** 108228 (2022). DOI: 10.1016/j.cpc.2021.108228

Additionally, we automatically generate a DOI for each pyg4ometry code release through Zenodo at this page, if you wish to cite a specific pyg4ometry version.

Other publications on pyg4ometry:

- S.T. Boogert, A. Abramov, J. Albrecht, G. D'Alessandro, L.J. Nevay, W. Shields, S.D. Walker, *Pyg4ometry : A Tool To Create Geometries For Geant4, Bdsim, G4Beamline and Fluka For Particle Loss and Energy Deposit Studies*, IPAC2019, Melbourne, Australia, 2019. DOI: 10.18429/JACoW-IPAC2019-WEPTS054

CHAPTER

FOUR

CHANGELOG (LEGACY)

V1.0.3 - 2022 / 08 / XX

New features

- Feature extraction from tessellated meshes (with some accelerator applications)
- New opencascade based step/iges loading
- Command line interface
- Clipping of geometry when top level solid is moved (rotated/translated) or changed
- New more efficient visualisation

4.1 V1.0.2 - 2022 / 04 / 11

4.1.1 New Features

- Function to view the difference between two logical volumes.
- Comparison test for names that ignores pointers as part of the name.
- New `addSolid` function for visualiser to view a solid in the current scene without having a logical volume.
- Every solid now has the function `convert2Tessellated` to allow it to be easily converted to a tessellated solid.
- Ability to 'collapse' assemblies (T. Latham)

4.1.2 General

- All solids now have a list of units to accompany their list of variables for inspection.
- Checking angles for solids (e.g. start and sweep angle) are within 2π now includes a numerical precision tolerance that is by default floating point precision. See `pyg4ometry.config.twoPiComparisonTolerance`.

4.1.3 Bug Fixes

- Fix deployment of nist_materials.txt and nist_elements.txt package files.
- Fix loading of ROOT geometry for material and various solids. Put tests in for parameters that ROOT allows to be 0 but ultimately mean we cannot construct a valid shape from.
- Reduce length of names in Geant4 to FLUKA conversion to fix rejected names by FLUKA.
- Fix import of MutableMappings for Python 3.10
- Fix use of units throughout all solids for the geometry comparison tests.
- Fix units for define vectors.
- Fix copy number of physical volumes sometimes not being an integer for loaded GDML geometry.
- Fix reading and writing of abundance if it is an expression in a GDML material.
- Fix missing length unit for GDML writing of Generic Trap.
- Fix units in GenericTrap and Extruded solid classes.
- Fix renaming of materials (in a recursion chain) when transferring from one registry to another (T. Latham).
- Reduce verbosity of ROOT tests.
- Reduce verbostiy of comparison tests.
- Fix zero division errors in various comparison tests.

4.2 v1.0.1 - 2022 / 02 / 10

4.2.1 New Features

- Geometry comparison tests for comparing two different geometry trees.
- Copy number for physical volumes.
- Overlap checking for assembly and replica volumes.
- Ability to analyse a registry structure and usage of objects throughout.
- ROOT geometry loader.

4.2.2 General

- Improved documentation and docstrings.
- Refactored registry for transfer and object versus simply adding an object.
- Optimised imports.
- Cleaned up private imports throughout code to make tab complete cleaner.
- Simplified code in solids by using base class.

4.2.3 Bug Fixes

- Improved search paths for libraries in setup.py.
- Fix merging of registries - object names won't be altered if they don't have to be. Fixed a clash with scales.
- Fixed conversion of FLUKA materials to Geant4 - many fixes.
- Fixed some material issues when converting Geant4 to FLUKA.

4.3 v0.9.0 - 2021 / 07 / 01

- Working version regularly used, submitted to CPC Journal for review.
- Based on CGAL for Boolean mesh operations, using pybind11, whereas previously was based on pycgal.
- FLUKA conversion to pyg4ometry and GDML has been reimplemented from the pyfluka package.
- Extensive code testing has been introduced and basic functionality documented.
- Given the strictness of CGAL, many bugs in meshing algorithms were fixed for all solids in *pyg4ometry.geant4.solid*.

4.4 Pre-History

- v0.2.0 - 2018 / 06 / 23
- v0.1.4 - 2018 / 06 / 04
- v0.1.2 - 2018 / 06 / 03
- v0.1.1 - 2018 / 06 / 03
- v0.1.0 - 2017 / 06 / 05
- v0.4.0 - 2017 / 10 / 17
- v0.3.0 - 2017 / 07 / 06

CHAPTER

FIVE

API REFERENCE

This page contains auto-generated API reference documentation[1].

5.1 pyg4ometry

5.1.1 Subpackages

pyg4ometry.analysis

Submodules

pyg4ometry.analysis.Data

Module Contents

Classes

TH1
TH2
TH3

class pyg4ometry.analysis.Data.**TH1**(*data*, *nPrimary=1*, *flukaBdxVariable='energy'*)

__getitem__(*slice*)

plot(*ms='.'*)

class pyg4ometry.analysis.Data.**TH2**(*data*, *nPrimary=1*)

__getitem__(*slice*)

plot()

[1] Created with sphinx-autoapi

class pyg4ometry.analysis.Data.**TH3**(*data*, *nPrimary=1*)

__getitem__(*slice*)

plot()

plot3Projections()

pyg4ometry.analysis.Plot

Module Contents

Functions

plotEventSections()	
plotEventSection(geometryFile[, flukaFile, bdsimFile])	
plotScoringMeshSection(geometryFile[, planeAxis, ...])	
flukaEventDisplay(gdmlFile, dumpFile)	

pyg4ometry.analysis.Plot.**plotEventSections**()

pyg4ometry.analysis.Plot.**plotEventSection**(*geometryFile*, *flukaFile=None*, *bdsimFile=None*)

pyg4ometry.analysis.Plot.**plotScoringMeshSection**(*geometryFile*, *planeAxis=1*, *flukaFile=None*, *flukaBin=None*, *bdsimFile=None*, *bdsimHisto=None*)

pyg4ometry.analysis.Plot.**flukaEventDisplay**(*gdmlFile*, *dumpFile*)

pyg4ometry.analysis.bdsimData

pyg4ometry.analysis.flukaData

Module Contents

Classes

_FlukaDataFile
FlukaBinData
FlukaBdxData
Usrbin
Usrbdx
Usrdump

Functions

fortran_skip(f)
fortran_read(f)
debugDumpFile(fd[, limit])

pyg4ometry.analysis.flukaData.**fortran_skip**(*f*)

pyg4ometry.analysis.flukaData.**fortran_read**(*f*)

pyg4ometry.analysis.flukaData.**debugDumpFile**(*fd*, *limit=10000000*)

class pyg4ometry.analysis.flukaData.**_FlukaDataFile**(*fd*)

read_header(*fd*)

read_data(*fd*)

print_header()

class pyg4ometry.analysis.flukaData.**FlukaBinData**(*index*, *name*, *type*)

binToVtkGrid()

class pyg4ometry.analysis.flukaData.**FlukaBdxData**(*index*, *name*, *type*)

class pyg4ometry.analysis.flukaData.**Usrbin**(*fd*, *read_data=False*)

Bases: *_FlukaDataFile*

read_file(*fd*)

read_data(*fd*)

read_stats(*fd*)

read_header(*fd*)

print_header()

class pyg4ometry.analysis.flukaData.**Usrbdx**(*file*)

Bases: *_FlukaDataFile*

read_file(*fd*)

read_data(*fd*)

read_stats(*fd*)

read_header(*fd*)

class pyg4ometry.analysis.flukaData.**Usrdump**(*fd*, *iEventHeaderToRead=10000*)

Bases: *_FlukaDataFile*

read_structure(*fd*, *iEventHeaderToRead=10000*)

read_event(*fd*, *ievent=0*)

trackDataToPolydata()

pyg4ometry.bdsim

Submodules

pyg4ometry.bdsim.beam

Module Contents

Classes

Beam

class pyg4ometry.bdsim.beam.**Beam**(*particleName='e-'*, *energy='16.5*GeV'*, *pos=[0, 0, 0]*, *dir=[0, 0, 1]*)

write(*fd*)

pyg4ometry.bdsim.beamline

Module Contents

Classes

Beamline

class pyg4ometry.bdsim.beamline.**Beamline**(*name*)

addElement(*element*)

write(*fd*)

pyg4ometry.bdsim.element

Module Contents

Classes

Element

class pyg4ometry.bdsim.element.**Element**(*name*, *type*, ***kwargs*)

write(*fd*)

pyg4ometry.bdsim.gmad

Module Contents

Classes

Gmad

class pyg4ometry.bdsim.gmad.**Gmad**(*beamline=None*, *beam=None*, *options=None*, *samplers=None*, *scorers=[]*, *scorer_meshes=[]*)

write(*fd='test.gmad'*)

pyg4ometry.bdsim.options

Module Contents

Classes

Options

class pyg4ometry.bdsim.options.**Options**

addOption(*key*, *value*)

write(*fd*)

pyg4ometry.bdsim.sampler

Module Contents

Classes

Sampler

class pyg4ometry.bdsim.sampler.**Sampler**(*range=None*, *type=None*)

write(*fd*)

pyg4ometry.bdsim.scoring

Module Contents

Classes

Scorer
ScorerMesh

class pyg4ometry.bdsim.scoring.**Scorer**(*name='score1'*, *type='depositedenergy'*, *parameters=None*, *conversionFactorFile=None*, *conversionFactorPath=None*)

write(*fd*)

class pyg4ometry.bdsim.scoring.**ScorerMesh**(*name*, *scorer*, *mesh='box'*, *position=[0, 0, 0]*, *rotation=[0, 0, 0]*, *nbins=[10, 10, 10]*, *size=[100, 100, 100]*)

write(*fd*)

Package Contents

Classes

Sampler
Scorer
ScorerMesh
Beamline
Options
Gmad
Element
Beam

class pyg4ometry.bdsim.**Sampler**(*range=None*, *type=None*)

write(*fd*)

class pyg4ometry.bdsim.**Scorer**(*name='score1'*, *type='depositedenergy'*, *parameters=None*, *conversionFactorFile=None*, *conversionFactorPath=None*)

write(*fd*)

class pyg4ometry.bdsim.**ScorerMesh**(*name*, *scorer*, *mesh='box'*, *position=[0, 0, 0]*, *rotation=[0, 0, 0]*, *nbins=[10, 10, 10]*, *size=[100, 100, 100]*)

write(*fd*)

class pyg4ometry.bdsim.**Beamline**(*name*)

addElement(*element*)

write(*fd*)

class pyg4ometry.bdsim.**Options**

addOption(*key*, *value*)

write(*fd*)

class pyg4ometry.bdsim.**Gmad**(*beamline=None*, *beam=None*, *options=None*, *samplers=None*, *scorers=[]*, *scorer_meshes=[]*)

write(*fd='test.gmad'*)

class pyg4ometry.bdsim.**Element**(*name*, *type*, ***kwargs*)

write(*fd*)

class pyg4ometry.bdsim.**Beam**(*particleName='e-'*, *energy='16.5*GeV'*, *pos=[0, 0, 0]*, *dir=[0, 0, 1]*)

write(*fd*)

pyg4ometry.compare

Submodules

pyg4ometry.compare._Compare

Module Contents

Classes

Tests	Set of options of which tests to perform and potentially with what tolerance.
TestResult	A test result can be either pass, fail or not conducted.
TestResultNamed	
ComparisonResult	Holder for a test result. Roughly a dict[testname] = list(TestResultNamed)

Functions

gdmlFiles(referenceFile, otherFile[, tests, ...])	**param referenceFile** GDML file to use as a reference.
geometry(referenceLV, otherLV[, tests, ...])	**param referenceLV** LogicalVolume instance to compare against.
logicalVolumes(referenceLV, otherLV, tests[, ...])	Compare two LogicalVolume instances with a set of tests.
physicalVolumes(referencePV, otherPV, tests[, ...])	lvName is an optional parent object name to help in print out details decode where the placement is.
assemblyVolumes(referenceAV, otherAV, tests[, ...])	
_checkPVLikeDaughters(referencePVLikeObject, ...[, ...])	
_testDaughterNameSets(referenceDaughterNameSet, ...)	
replicaVolumes(referenceRV, otherRV, tests[, ...])	
divisionVolumes(referenceRV, otherRV, tests[, ...])	Compare two DivisionVolume instances with a set of tests.
parameterisedVolumes(referenceRV, otherRV, tests[, ...])	Compare two ParameterisedVolume instances with a set of tests.
materials(referenceMaterial, otherMaterial, tests[, ...])	Compare two materials with a set of tests.
_elements(referenceElement, otherElement, tests[, ...])	
solids(referenceSolid, otherSolid, tests[, lvName, ...])	Compare any two solids with a set of tests.
_names(testName, str1, str2[, parentName, ...])	
_namesIgnorePointer(testName, str1, str2[, ...])	
_vector(vectortype, r1, r2, tests[, parentName, ...])	
_copyNumber(pvname, c1, c2, tests[, includeAllTestResults])	
_getBoundingBox(obj)	
_getRawMesh(obj)	
_meshes(lvname, referenceMesh, otherMesh, tests[, ...])	

class pyg4ometry.compare._Compare.**Tests**(**testsByNameToTurnOn*)

Set of options of which tests to perform and potentially with what tolerance.

Parameters

testsByNameToTurnOn (*str, list of*) – optional strings of tests to turn on. If any, all will be turned off first.

```
>>> t = Tests()
>>> t = Tests("nDaughters") # only nDaughters will be tested
>>> t = Tests("nDaughters", "shapeArea") # only nDaughters and shapeArea will be␣
↪tested
```

_testNames = ['names', 'namesIgnorePointer', 'nDaughters', 'solidExact', 'solidExtent', 'shapeExtent',...

setAllFalse()

Utility to turn off all tests at once. Then can just turn on the one we want.

setFalse(*testName*)

Safely set a test by name to False.

__len__()

classmethod printAllTestNames()

Print all tests names - the exact strings that can be used to turn them off or on.

__repr__()

Return repr(self).

class pyg4ometry.compare._Compare.**TestResult**

Bases: enum.Enum

A test result can be either pass, fail or not conducted.

Use 0,1 so we can also implicitly construct this with a Boolean.

Use the bitwise or operator | and not the keyword *or*. This bitwise or operator returns Failed if either have failed. Only returns NotTested if both are not tested. Cannot use bitwise |= as we cannot update an Enum internally.

Use TestResult.All() for a list of all possible results - useful as an argument for printing.

Failed = 0

Passed = 1

NotTested = 2

__or__(*other*)

__ior__(*other*)

static All()

Utility function to get a list of all test types in one.

class pyg4ometry.compare._Compare.**TestResultNamed**(*nameIn*, *testResultIn=TestResult.Failed*, *detailsIn=''*)

__str__()

Return str(self).

class pyg4ometry.compare._Compare.**ComparisonResult**

Holder for a test result. Roughly a dict[testname] = list(TestResultNamed)

Use + and += to append to this object. Uses a default dictionary so no need to initialise any key names. Should always append a list even if only 1 item.

```
>>> cr = ComparisonResult()
>>> cr['nDaughtersTest'] += [TestResultNamed('volume_1', TestResult.Failed,
↪'different number')]
>>> cr.print()
```

print() can take a list of test result outcomes to print. e.g. TestResult.All()

__getitem__(*key*)

__setitem__(*key*, *value*)

__add__(*other*)

__iadd__(*other*)

__len__()

testNames()

print(*testName=None*, *testResultsToPrint=[TestResult.Failed, TestResult.NotTested]*, *allTests=False*)

Parameters

- **testName** (*str*) – (optional) name of specific single test to print - see testNames()
- **testResultsToPrint** (*list(TestResult)*) – (optional) list of result outcomes to print
- **allTests** (*bool*) – (optional) print all tests irrespective of the result

Print failed tests and ones relevant but not implemented yet by default. Control level of print out with optional argument of list of test outcomes to print, or allTests=True.

```
>>> cr.print()
>>> cr.print('solidName')
>>> cr.print(testResultsToPrint=TestResult.All())
>>> cr.print(allTests=True)
```

pyg4ometry.compare._Compare.**gdmlFiles**(*referenceFile*, *otherFile*, *tests=Tests()*, *includeAllTestResults=False*)

Parameters

- **referenceFile** (*str.*) – GDML file to use as a reference.
- **otherFile** (*str.*) – GDML file to compare.
- **tests** (*pyg4ometry.compare._Compare.Tests.*) – Tests instance to use.
- **includeAllTestResults** (*bool.*) – document all tests attempted in result.

pyg4ometry.compare._Compare.**geometry**(*referenceLV*, *otherLV*, *tests=Tests()*, *includeAllTestResults=False*)

Parameters

- **referenceLV** (LogicalVolume) – LogicalVolume instance to compare against.
- **otherLV** (LogicalVolume) – LogicalVolume instance to compare with referenceLV.
- **tests** (*pyg4ometry.compare._Compare.Tests.*) – Tests instance to use.
- **includeAllTestResults** (*bool.*) – document all tests attempted in result.

pyg4ometry.compare._Compare.**logicalVolumes**(*referenceLV*, *otherLV*, *tests*, *recursive=False*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two LogicalVolume instances with a set of tests.

testsAlreadyDone should be purposively given as an empty list if using more than once. This is an artefact of the only way we can nicely pass by reference a set of tests while we use these functions recursively. However, this is defined on import and must be updated if reused.

pyg4ometry.compare._Compare.**physicalVolumes**(*referencePV*, *otherPV*, *tests*, *recursive=False*, *lvName=''*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

lvName is an optional parent object name to help in print out details decode where the placement is.

pyg4ometry.compare._Compare.**assemblyVolumes**(*referenceAV*, *otherAV*, *tests*, *recursive=False*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

pyg4ometry.compare._Compare.**_checkPVLikeDaughters**(*referencePVLikeObject*, *otherPVLikeObject*, *tests*, *parentName*, *testName*, *result*, *recursive=True*, *includeAllTestResults=True*, *testsAlreadyDone=[]*)

pyg4ometry.compare._Compare.**_testDaughterNameSets**(*referenceDaughterNameSet*, *otherDaughterNameSet*, *result*, *testName*, *includeAllTestResults*)

pyg4ometry.compare._Compare.**replicaVolumes**(*referenceRV*, *otherRV*, *tests*, *recursive=True*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

pyg4ometry.compare._Compare.**divisionVolumes**(*referenceRV*, *otherRV*, *tests*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two DivisionVolume instances with a set of tests.

pyg4ometry.compare._Compare.**parameterisedVolumes**(*referenceRV*, *otherRV*, *tests*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two ParameterisedVolume instances with a set of tests.

pyg4ometry.compare._Compare.**materials**(*referenceMaterial*, *otherMaterial*, *tests*, *lvName=''*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two materials with a set of tests.

This tests assumes both referenceMaterial and otherMaterial are derived from the type pyg4ometry.geant4._Material.Material.

Compares, name, classname, density, n components

pyg4ometry.compare._Compare.**_elements**(*referenceElement*, *otherElement*, *tests*, *lvName=''*, *includeAllTestResults=False*)

pyg4ometry.compare._Compare.**solids**(*referenceSolid*, *otherSolid*, *tests*, *lvName=''*, *includeAllTestResults=False*)

Compare any two solids with a set of tests.

pyg4ometry.compare._Compare.**_names**(*testName*, *str1*, *str2*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare._Compare.**_namesIgnorePointer**(*testName*, *str1*, *str2*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare._Compare.**_vector**(*vectortype*, *r1*, *r2*, *tests*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare._Compare.**_copyNumber**(*pvname*, *c1*, *c2*, *tests*, *includeAllTestResults=False*)

pyg4ometry.compare._Compare.**_getBoundingBox**(*obj*)

pyg4ometry.compare._Compare.**_getRawMesh**(*obj*)

pyg4ometry.compare._Compare.**_meshes**(*lvname*, *referenceMesh*, *otherMesh*, *tests*, *includeAllTestResults=False*)

Package Contents

Classes

_Material	This class provides an interface to GDML material definitions.
_Element	This class provides an interface to GDML material definitions. Because of the different options
Tests	Set of options of which tests to perform and potentially with what tolerance.
TestResult	A test result can be either pass, fail or not conducted.
TestResultNamed	
ComparisonResult	Holder for a test result. Roughly a dict[testname] = list(TestResultNamed)

Functions

_evaluateToFloat(reg, obj)	
gdmlFiles(referenceFile, otherFile[, tests, ...])	**param referenceFile** GDML file to use as a reference.
geometry(referenceLV, otherLV[, tests, ...])	**param referenceLV** LogicalVolume instance to compare against.
logicalVolumes(referenceLV, otherLV, tests[, ...])	Compare two LogicalVolume instances with a set of tests.
physicalVolumes(referencePV, otherPV, tests[, ...])	lvName is an optional parent object name to help in print out details decode where the placement is.
assemblyVolumes(referenceAV, otherAV, tests[, ...])	
_checkPVLikeDaughters(referencePVLikeObject, ...[, ...])	
_testDaughterNameSets(referenceDaughterNameSet, ...)	
replicaVolumes(referenceRV, otherRV, tests[, ...])	
divisionVolumes(referenceRV, otherRV, tests[, ...])	Compare two DivisionVolume instances with a set of tests.
parameterisedVolumes(referenceRV, otherRV, tests[, ...])	Compare two ParameterisedVolume instances with a set of tests.
materials(referenceMaterial, otherMaterial, tests[, ...])	Compare two materials with a set of tests.
_elements(referenceElement, otherElement, tests[, ...])	
solids(referenceSolid, otherSolid, tests[, lvName, ...])	Compare any two solids with a set of tests.
_names(testName, str1, str2[, parentName, ...])	
_namesIgnorePointer(testName, str1, str2[, ...])	
_vector(vectortype, r1, r2, tests[, parentName, ...])	
_copyNumber(pvname, c1, c2, tests[, includeAllTestResults])	
_getBoundingBox(obj)	
_getRawMesh(obj)	
_meshes(lvname, referenceMesh, otherMesh, tests[, ...])	

pyg4ometry.compare.**_evaluateToFloat**(*reg*, *obj*)

class pyg4ometry.compare.**_Material**(***kwargs*)

Bases: MaterialBase

This class provides an interface to GDML material definitions.

Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

MaterialSingleElement MaterialCompound MaterialPredefined

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string density - float atomic_number - int atomic_weight - float number_of_components - int state - string pressure - float pressure_unit - string temperature - float temperature_unit - string

property state_variables

add_element_massfraction(*element*, *massfraction*)

Add an element as a component to a material as a fraction of the material mass. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance massfraction - float, 0.0 < massfraction <= 1.0

add_element_natoms(*element*, *natoms*)

Add an element as a component to a material as a number of atoms in the material molecule. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance natoms - int, number of atoms in the compound molecule

add_material(*material*, *fractionmass*)

Add a material as a component to another material (mixture) as a fraction of the mixture mass. Can only add new materials to materials defined as composite.

Inputs:

material - pyg4ometry.geant4.Material.Material instance massfraction - float, 0.0 < massfraction <= 1.0

set_pressure(*value*, *unit='pascal'*)

set_temperature(*value*, *unit='K'*)

__str__()

Return str(self).

addProperty(*name*, *matrix*)

Add a material property from a matrix.

Parameters

- **name** (*str*) – key of the material property
- **matrix** (*Matrix*) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit

- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str,float,int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.compare.**_Element**(***kwargs*)

Bases: MaterialBase

This class provides an interface to GDML material definitions. Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

ElementSimple ElementIsotopeMixture

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string symbol - string Z - int A - int n_comp - int

add_isotope(*isotope*, *abundance*)

Add an isotope as a component to an element as an abundance fraction in the element.

Inputs:
element - pyg4ometry.geant4.Material.Isotope instance abundance - float, 0.0 < abundance <= 1.0

class pyg4ometry.compare.**Tests**(**testsByNameToTurnOn*)

Set of options of which tests to perform and potentially with what tolerance.

Parameters
testsByNameToTurnOn (*str, list of*) – optional strings of tests to turn on. If any, all will be turned off first.

```
>>> t = Tests()
>>> t = Tests("nDaughters") # only nDaughters will be tested
>>> t = Tests("nDaughters", "shapeArea") # only nDaughters and shapeArea will be␣
↪tested
```

_testNames = ['names', 'namesIgnorePointer', 'nDaughters', 'solidExact', 'solidExtent', 'shapeExtent',...

setAllFalse()

Utility to turn off all tests at once. Then can just turn on the one we want.

setFalse(*testName*)

Safely set a test by name to False.

__len__()

classmethod printAllTestNames()

Print all tests names - the exact strings that can be used to turn them off or on.

__repr__()

Return repr(self).

class pyg4ometry.compare.**TestResult**

Bases: enum.Enum

A test result can be either pass, fail or not conducted.

Use 0,1 so we can also implicitly construct this with a Boolean.

Use the bitwise or operator | and not the keyword *or*. This bitwise or operator returns Failed if either have failed. Only returns NotTested if both are not tested. Cannot use bitwise |= as we cannot update an Enum internally.

Use TestResult.All() for a list of all possible results - useful as an argument for printing.

Failed = 0

Passed = 1

NotTested = 2

__or__(*other*)

__ior__(*other*)

static All()

Utility function to get a list of all test types in one.

class pyg4ometry.compare.**TestResultNamed**(*nameIn*, *testResultIn=TestResult.Failed*, *detailsIn=''*)

__str__()

Return str(self).

class pyg4ometry.compare.**ComparisonResult**

Holder for a test result. Roughly a dict[testname] = list(TestResultNamed)

Use + and += to append to this object. Uses a default dictionary so no need to initialise any key names. Should always append a list even if only 1 item.

```
>>> cr = ComparisonResult()
>>> cr['nDaughtersTest'] += [TestResultNamed('volume_1', TestResult.Failed,
↪'different number')]
>>> cr.print()
```

print() can take a list of test result outcomes to print. e.g. TestResult.All()

__getitem__(*key*)

__setitem__(*key*, *value*)

__add__(*other*)

__iadd__(*other*)

__len__()

testNames()

print(*testName=None*, *testResultsToPrint=[TestResult.Failed, TestResult.NotTested]*, *allTests=False*)

Parameters

- **testName** (*str*) – (optional) name of specific single test to print - see testNames()
- **testResultsToPrint** (*list(TestResult)*) – (optional) list of result outcomes to print
- **allTests** (*bool*) – (optional) print all tests irrespective of the result

Print failed tests and ones relevant but not implemented yet by default. Control level of print out with optional argument of list of test outcomes to print, or allTests=True.

```
>>> cr.print()
>>> cr.print('solidName')
>>> cr.print(testResultsToPrint=TestResult.All())
>>> cr.print(allTests=True)
```

pyg4ometry.compare.**gdmlFiles**(*referenceFile*, *otherFile*, *tests=Tests()*, *includeAllTestResults=False*)

Parameters

- **referenceFile** (*str.*) – GDML file to use as a reference.
- **otherFile** (*str.*) – GDML file to compare.
- **tests** (*pyg4ometry.compare._Compare.Tests.*) – Tests instance to use.
- **includeAllTestResults** (*bool.*) – document all tests attempted in result.

pyg4ometry.compare.**geometry**(*referenceLV*, *otherLV*, *tests=Tests()*, *includeAllTestResults=False*)

Parameters

- **referenceLV** (LogicalVolume) – LogicalVolume instance to compare against.
- **otherLV** (LogicalVolume) – LogicalVolume instance to compare with referenceLV.
- **tests** (*pyg4ometry.compare._Compare.Tests.*) – Tests instance to use.
- **includeAllTestResults** (*bool.*) – document all tests attempted in result.

pyg4ometry.compare.**logicalVolumes**(*referenceLV*, *otherLV*, *tests*, *recursive=False*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two LogicalVolume instances with a set of tests.

testsAlreadyDone should be purposively given as an empty list if using more than once. This is an artefact of the only way we can nicely pass by reference a set of tests while we use these functions recursively. However, this is defined on import and must be updated if reused.

pyg4ometry.compare.**physicalVolumes**(*referencePV*, *otherPV*, *tests*, *recursive=False*, *lvName=''*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

lvName is an optional parent object name to help in print out details decode where the placement is.

pyg4ometry.compare.**assemblyVolumes**(*referenceAV*, *otherAV*, *tests*, *recursive=False*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

pyg4ometry.compare.**_checkPVLikeDaughters**(*referencePVLikeObject*, *otherPVLikeObject*, *tests*, *parentName*, *testName*, *result*, *recursive=True*, *includeAllTestResults=True*, *testsAlreadyDone=[]*)

pyg4ometry.compare.**_testDaughterNameSets**(*referenceDaughterNameSet*, *otherDaughterNameSet*, *result*, *testName*, *includeAllTestResults*)

pyg4ometry.compare.**replicaVolumes**(*referenceRV*, *otherRV*, *tests*, *recursive=True*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

pyg4ometry.compare.**divisionVolumes**(*referenceRV*, *otherRV*, *tests*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two DivisionVolume instances with a set of tests.

pyg4ometry.compare.**parameterisedVolumes**(*referenceRV*, *otherRV*, *tests*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two ParameterisedVolume instances with a set of tests.

pyg4ometry.compare.**materials**(*referenceMaterial*, *otherMaterial*, *tests*, *lvName=''*, *includeAllTestResults=False*, *testsAlreadyDone=[]*)

Compare two materials with a set of tests.

This tests assumes both referenceMaterial and otherMaterial are derived from the type pyg4ometry.geant4._Material.Material.

Compares, name, classname, density, n components

pyg4ometry.compare.**_elements**(*referenceElement*, *otherElement*, *tests*, *lvName=''*, *includeAllTestResults=False*)

pyg4ometry.compare.**solids**(*referenceSolid*, *otherSolid*, *tests*, *lvName=''*, *includeAllTestResults=False*)

Compare any two solids with a set of tests.

pyg4ometry.compare.**_names**(*testName*, *str1*, *str2*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare.**_namesIgnorePointer**(*testName*, *str1*, *str2*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare.**_vector**(*vectortype*, *r1*, *r2*, *tests*, *parentName=''*, *includeAllTestResults=False*)

pyg4ometry.compare.**_copyNumber**(*pvname*, *c1*, *c2*, *tests*, *includeAllTestResults=False*)

pyg4ometry.compare.**_getBoundingBox**(*obj*)

pyg4ometry.compare.**_getRawMesh**(*obj*)

pyg4ometry.compare.**_meshes**(*lvname*, *referenceMesh*, *otherMesh*, *tests*, *includeAllTestResults=False*)

pyg4ometry.convert

Submodules

pyg4ometry.convert.fluka2Geant4

Module Contents

Functions

fluka2Geant4(flukareg[, regions, omitRegions, ...])	Convert a FLUKA registry to a Geant4 Registry.
_flukaRegistryToG4Registry(flukareg, regions, ...)	Convert a transformed fluka registry to a geant4 registry.
_filteredRegions(flukareg, regions)	
_makeWorldVolume(dimensions, material, g4registry)	Make a world solid and logical volume with the given dimensions,
_makeLengthSafetyRegistry(flukareg, regions)	Make a new registry from a registry with length safety applied
_getRegionZoneAABBs(flukareg, regions, quadricRegionAABBs)	Loop over the regions, and for each region, get all the aabbs
_filterRegistryNullZones(flukareg, regionZoneAABBs)	
_filterRegionNullZones(region, aabbs)	
_filterNullAABBs(regionZoneAABBs)	
_makeBodyMinimumAABBMap(flukareg, regionZoneAABBs, regions)	
_getMaximalOfTwoAABBs(aabb1, aabb2)	Given two aabbs, returns the total aabb that tightly bounds
_filterBlackHoleRegions(flukareg, regions)	Returns a new FlukaRegistry instance with all regions with
_getOverlappingAABBs(aabb, aabbs)	
_getContentsOfLatticeCells(flukaregistry, regionAABBs)	
_getTransformedCellRegionAABB(lattice)	
_isTransformedCellRegionIntersectingWithRegi ...)	
_checkQuadricRegionAABBs(flukareg, quadricRegionAABBs)	Loop over the regions looking for quadrics and for any quadrics we
_getWorldDimensions(worldDimensions)	Get world dimensinos and if None then return the global constant
_getSelectedRegions(flukareg, regions, omitRegions)	
_filterHalfSpaces(flukareg, regionZoneAABBs)	Filter redundant half spaces from the regions of the
_distanceFromPointToPlane(normal, pointOnPlane, point)	
_convertLatticeCells(greg, flukareg, wlv, ...)	
_makeUniqueQuadricRegions(flukareg, quadricRegionAABBs)	
_makeQuadricRegionBodyAABBMap(flukareg, quadricRegionAABBs)	Given a map of regions featuring quadrics to their aabbs, we
_getMaximalQuadricRegionAABBs(freg, quadricRegionAABBs)	
_regionZoneAABBsToRegionAABBs(regionZoneAABB	Given a map of region names to zone aabbs, return a map of
_copyStructureToNewFlukaRegistry(freg, fregtarget)	

Attributes

logger
WORLD_DIMENSIONS

pyg4ometry.convert.fluka2Geant4.**logger**

pyg4ometry.convert.fluka2Geant4.**WORLD_DIMENSIONS = [10000, 10000, 10000]**

exception pyg4ometry.convert.fluka2Geant4.**NullModel**

Bases: Exception

Common base class for all non-exit exceptions.

pyg4ometry.convert.fluka2Geant4.**fluka2Geant4**(*flukareg*, *regions=None*, *omitRegions=None*, *worldMaterial='G4_Galactic'*, *worldDimensions=None*, *omitBlackholeRegions=True*, *quadricRegionAABBs=None*, ***kwargs*)

Convert a FLUKA registry to a Geant4 Registry.

Parameters

- **flukareg** (FlukaRegistry) – FlukaRegistry instance to be converted.
- **regions** (*list*) – Names of regions to be converted, by default all are converted. Mutually exclusive with omitRegions.
- **omitRegions** (*list*) – Names of regions to be omitted from the conversion. This option is mutually exclusive with the kwarg regions.
- **worldMaterial** (*string*) – name of world material to be used.
- **worldDimensions** (*list*) – dimensions of world logical volume in converted Geant4. By default this is equal to WORLD_DIMENSIONS.
- **omitBlackholeRegions** (*bool*) – whether or not to omit regions with the FLUKA material BLCKHOLE from the conversion. By default, true.
- **quadricRegionAABBs** (*dict*) – The axis-aligned aabbs of any regions featuring QUA bodies, mapping region names to fluka.AABB instances.

Developer options (to kwargs) withLengthSafety: Whether or not to apply automatic length safety.

minimiseSolids: Whether or not to minimise the boxes and tubes of Geant4 used to represent infinite solids in FLUKA.

pyg4ometry.convert.fluka2Geant4.**_flukaRegistryToG4Registry**(*flukareg*, *regions*, *worldinfo*, *aabbinfo*)

Convert a transformed fluka registry to a geant4 registry.

pyg4ometry.convert.fluka2Geant4.**_filteredRegions**(*flukareg*, *regions*)

pyg4ometry.convert.fluka2Geant4.**_makeWorldVolume**(*dimensions*, *material*, *g4registry*)

Make a world solid and logical volume with the given dimensions, material, and size, and add it to the geant4 Registry provided

Parameters

dimensions – list of 3 elements providing the dimensions in [x, y, z]

of the world box. :type dimension: list :param material: The name of the material to be used for the world volume. :type material: str :param g4registry: The geant4 Registry instance this world solid and logical volume is to be added to.

pyg4ometry.convert.fluka2Geant4.**_makeLengthSafetyRegistry**(*flukareg*, *regions*)

Make a new registry from a registry with length safety applied to the zones and regions within.

Parameters
flukareg – The FlukaRegistry from which the new registry

with length safety applied should be built. :type flukareg: FlukaRegistry :param regions: The names of the regions that are to be converted.

pyg4ometry.convert.fluka2Geant4.**_getRegionZoneAABBs**(*flukareg*, *regions*, *quadricRegionAABBs*)

Loop over the regions, and for each region, get all the aabbs of the zones belonging to that region. Don't do this for quadricRegionAABBs, instead, just continue to use the aabb provided by the user.

pyg4ometry.convert.fluka2Geant4.**_filterRegistryNullZones**(*flukareg*, *regionZoneAABBs*)

pyg4ometry.convert.fluka2Geant4.**_filterRegionNullZones**(*region*, *aabbs*)

pyg4ometry.convert.fluka2Geant4.**_filterNullAABBs**(*regionZoneAABBs*)

pyg4ometry.convert.fluka2Geant4.**_makeBodyMinimumAABBMap**(*flukareg*, *regionZoneAABBs*, *regions*)

pyg4ometry.convert.fluka2Geant4.**_getMaximalOfTwoAABBs**(*aabb1*, *aabb2*)

Given two aabbs, returns the total aabb that tightly bounds the two given aabbs.

Parameters

- **aabb1** (AABB) – The first aabb.
- **aabb2** (AABB) – The second aabb

pyg4ometry.convert.fluka2Geant4.**_filterBlackHoleRegions**(*flukareg*, *regions*)

Returns a new FlukaRegistry instance with all regions with BLKCHOLE material removed.

Parameters
flukareg – The FlukaRegistry instance from which the new

FlukaRegistry instance sans BLCKHOLE regions is built. :type flukareg: FlukaRegistry :param regions: The names of the regions to be copied to the new instance. :type regions: list

pyg4ometry.convert.fluka2Geant4.**_getOverlappingAABBs**(*aabb*, *aabbs*)

pyg4ometry.convert.fluka2Geant4.**_getContentsOfLatticeCells**(*flukaregistry*, *regionAABBs*)

pyg4ometry.convert.fluka2Geant4.**_getTransformedCellRegionAABB**(*lattice*)

pyg4ometry.convert.fluka2Geant4.**_isTransformedCellRegionIntersectingWithRegion**(*region*, *lattice*)

pyg4ometry.convert.fluka2Geant4.**_checkQuadricRegionAABBs**(*flukareg*, *quadricRegionAABBs*)

Loop over the regions looking for quadrics and for any quadrics we find, make sure that the region has a defined region aabb in quadricRegionAABBs.

pyg4ometry.convert.fluka2Geant4.**_getWorldDimensions**(*worldDimensions*)

Get world dimensinos and if None then return the global constant WORLD_DIMENSIONS.

pyg4ometry.convert.fluka2Geant4.**_getSelectedRegions**(*flukareg*, *regions*, *omitRegions*)

pyg4ometry.convert.fluka2Geant4.**_filterHalfSpaces**(*flukareg*, *regionZoneAABBs*)

Filter redundant half spaces from the regions of the FlukaRegistry instance. AABBs is a dictionary of region names to region aabbs.

pyg4ometry.convert.fluka2Geant4.**_distanceFromPointToPlane**(*normal*, *pointOnPlane*, *point*)

pyg4ometry.convert.fluka2Geant4.**_convertLatticeCells**(*greg*, *flukareg*, *wlv*, *regionZoneAABBs*, *regionNamesToLVs*)

pyg4ometry.convert.fluka2Geant4.**_makeUniqueQuadricRegions**(*flukareg*, *quadricRegionAABBs*)

pyg4ometry.convert.fluka2Geant4.**_makeQuadricRegionBodyAABBMap**(*flukareg*, *quadricRegionAABBs*)

Given a map of regions featuring quadrics to their aabbs, we loop over the bodies of the region and set their aabbs equal to the region aabb.

pyg4ometry.convert.fluka2Geant4.**_getMaximalQuadricRegionAABBs**(*freg*, *quadricRegionAABBs*)

pyg4ometry.convert.fluka2Geant4.**_regionZoneAABBsToRegionAABBs**(*regionZoneAABBs*)

Given a map of region names to zone aabbs, return a map of region names to a total region aabb.

pyg4ometry.convert.fluka2Geant4.**_copyStructureToNewFlukaRegistry**(*freg*, *fregtarget*)

pyg4ometry.convert.fluka2g4materials

Module Contents

Classes

_FlukaToG4MaterialConverter	
_PeriodicTable	

Functions

_getPeriodicTable()	
makeFlukaToG4MaterialsMap(freg, greg)	Convert the materials defined in a FlukaRegistry, and populate

Attributes

_periodicTable
FLUKA_BUILTIN_TO_G4_MATERIAL_MAP
_FLUKA_ELEMENT_SYMBOLS

pyg4ometry.convert.fluka2g4materials.**_periodicTable**

pyg4ometry.convert.fluka2g4materials.**FLUKA_BUILTIN_TO_G4_MATERIAL_MAP**

pyg4ometry.convert.fluka2g4materials.**_FLUKA_ELEMENT_SYMBOLS**

pyg4ometry.convert.fluka2g4materials.**_getPeriodicTable**()

class pyg4ometry.convert.fluka2g4materials.**_FlukaToG4MaterialConverter**(*freg*, *greg*)

addPredefinedElements()

convertAll()

convertBuiltin(*name*, *flukaMaterial*)

convertElement(*name*, *flukaElement*)

convertCompound(*flukaCompound*)

_makeBaseCompoundMaterial(*name*, *flukaCompound*)

convertMassFractionCompound(*name*, *flukaCompound*)

convertVolumeFractionCompound(*name*, *flukaCompound*)

convertAtomicFractionCompound(*name*, *flukaCompound*)

_totalMassWeightedFractionOfCompound(*compound*)

static _mangleElementName(*name*)

pyg4ometry.convert.fluka2g4materials.**makeFlukaToG4MaterialsMap**(*freg*, *greg*)

Convert the materials defined in a FlukaRegistry, and populate the provided geant4 registry with said materials.

class pyg4ometry.convert.fluka2g4materials.**_PeriodicTable**

massNumberFromZ(*z*)

atomicMassFromZ(*z*)

atomicNumberAndMassFromSymbol(*symbol*)

pyg4ometry.convert.freecad2Fluka

Module Contents

Functions

freecadDoc2Fluka(fcd)	
part2Region(obj, trfm, fgreg[, meshDeviation])	

pyg4ometry.convert.freecad2Fluka.**freecadDoc2Fluka**(*fcd*)

pyg4ometry.convert.freecad2Fluka.**part2Region**(*obj*, *trfm*, *fgreg*, *meshDeviation=0.05*)

pyg4ometry.convert.gdml2stl

Module Contents

Functions

gdml2stl(gdmlFileName[, stlFileName, solidName])	

pyg4ometry.convert.gdml2stl.**gdml2stl**(*gdmlFileName*, *stlFileName='output.gdml'*, *solidName='ws'*)

pyg4ometry.convert.geant42Fluka

Module Contents

Functions

geant4Reg2FlukaReg(greg[, logicalVolumeName])	Convert a Geant4 model to a FLUKA one. This is done by handing over a complete
geant4Logical2Fluka(logicalVolume[, flukaRegistry])	Convert a single logical volume - not the main entry point for the conversion.
geant4PhysicalVolume2Fluka(physicalVolume[, mtra, ...])	
geant4Solid2FlukaRegion(flukaNameCount, solid[, mtra, ...])	
geant4MaterialDict2Fluka(matr, freg)	
geant4Material2Fluka(material, freg[, ...])	
pycsgmesh2FlukaRegion(mesh, name, transform, ...)	
makeStripName(mn)	
makeShortName(mn)	

pyg4ometry.convert.geant42Fluka.**geant4Reg2FlukaReg**(*greg*, *logicalVolumeName=''*)

Convert a Geant4 model to a FLUKA one. This is done by handing over a complete pyg4ometry.geant4.Registry instance.

Parameters
greg (pyg4ometry.geant4.Registry) – geant4 registry

returns: pyg4ometry.fluka.FlukaRegistry

pyg4ometry.convert.geant42Fluka.**geant4Logical2Fluka**(*logicalVolume*, *flukaRegistry=None*)

Convert a single logical volume - not the main entry point for the conversion.

pyg4ometry.convert.geant42Fluka.**geant4PhysicalVolume2Fluka**(*physicalVolume*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *flukaNameCount=0*)

pyg4ometry.convert.geant42Fluka.**geant4Solid2FlukaRegion**(*flukaNameCount*, *solid*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *addRegistry=True*, *commentName=''*)

pyg4ometry.convert.geant42Fluka.**geant4MaterialDict2Fluka**(*matr*, *freg*)

pyg4ometry.convert.geant42Fluka.**geant4Material2Fluka**(*material*, *freg*, *suggestedDensity=None*, *elementSuffix=False*, *materialNameShort=None*)

pyg4ometry.convert.geant42Fluka.**pycsgmesh2FlukaRegion**(*mesh*, *name*, *transform*, *flukaRegistry*, *commentName*)

pyg4ometry.convert.geant42Fluka.**makeStripName**(*mn*)

pyg4ometry.convert.geant42Fluka.**makeShortName**(*mn*)

pyg4ometry.convert.geant42FlukaBake

Module Contents

Functions

geant4Reg2FlukaReg(greg[, logicalVolumeName])	Convert a Geant4 model to a FLUKA one. This is done by handing over a complete
geant4Logical2Fluka(logicalVolume[, flukaRegistry])	Convert a single logical volume - not the main entry point for the conversion.
geant4PhysicalVolume2Fluka(physicalVolume[, mtra, ...])	
geant4Solid2FlukaRegion(flukaNameCount, solid[, mtra, ...])	
geant4MaterialDict2Fluka(matr, freg)	
geant4Material2Fluka(material, freg[, ...])	
pycsgmesh2FlukaRegion(mesh, name, transform, ...)	
makeStripName(mn)	
makeShortName(mn)	

pyg4ometry.convert.geant42FlukaBake.**geant4Reg2FlukaReg**(*greg*, *logicalVolumeName=''*)

Convert a Geant4 model to a FLUKA one. This is done by handing over a complete pyg4ometry.geant4.Registry instance.

Parameters

greg (pyg4ometry.geant4.Registry) – geant4 registry

returns: pyg4ometry.fluka.FlukaRegistry

pyg4ometry.convert.geant42FlukaBake.**geant4Logical2Fluka**(*logicalVolume*, *flukaRegistry=None*)

Convert a single logical volume - not the main entry point for the conversion.

pyg4ometry.convert.geant42FlukaBake.**geant4PhysicalVolume2Fluka**(*physicalVolume*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *flukaNameCount=0*)

pyg4ometry.convert.geant42FlukaBake.**geant4Solid2FlukaRegion**(*flukaNameCount*, *solid*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *addRegistry=True*, *commentName=''*)

pyg4ometry.convert.geant42FlukaBake.**geant4MaterialDict2Fluka**(*matr*, *freg*)

pyg4ometry.convert.geant42FlukaBake.**geant4Material2Fluka**(*material*, *freg*, *suggestedDensity=None*, *elementSuffix=False*, *materialNameShort=None*)

pyg4ometry.convert.geant42FlukaBake.**pycsgmesh2FlukaRegion**(*mesh*, *name*, *transform*, *flukaRegistry*, *commentName*)

pyg4ometry.convert.geant42FlukaBake.**makeStripName**(*mn*)

pyg4ometry.convert.geant42FlukaBake.**makeShortName**(*mn*)

pyg4ometry.convert.geant42Geant4

Module Contents

Functions

geant4Solid2Geant4Tessellated(solid)	
geant4Solid2Geant4Tessellated_NoVTK(solid)	

pyg4ometry.convert.geant42Geant4.**geant4Solid2Geant4Tessellated**(*solid*)

pyg4ometry.convert.geant42Geant4.**geant4Solid2Geant4Tessellated_NoVTK**(*solid*)

pyg4ometry.convert.geant42Vtk

pyg4ometry.convert.oce2Geant4

Module Contents

Functions

oceShape_Geant4_LogicalVolume(name, solid, material, greg)	Make a logical volume from input or get from registry
oceShape_Geant4_Assembly(name, greg)	Make a assembly volume from input or get from registry
oceShape_Geant4_Tessellated(name, shape, greg[, ...])	Make a tessellated solid from a OpenCascade shape
_oce2Geant4_traverse(shapeTool, label, greg, ...[, ...])	
oce2Geant4(shapeTool, shapeName[, materialMap, ...])	Convert CAD geometry starting from shapeName

Attributes

defaultLinDef	
deftaulAngDef	

pyg4ometry.convert.oce2Geant4.**defaultLinDef = 0.5**

pyg4ometry.convert.oce2Geant4.**deftaulAngDef = 0.5**

pyg4ometry.convert.oce2Geant4.**oceShape_Geant4_LogicalVolume**(*name*, *solid*, *material*, *greg*)

Make a logical volume from input or get from registry

Parameters

- **name** (*str*) – Name of logical volume
- **solid** (SolidBase) – Geant4 solid
- **material** (*str or* pyg4ometry.geant4.Material) – Material for logical volume
- **greg** (geant4.Registry) – Geant4 registry

pyg4ometry.convert.oce2Geant4.**oceShape_Geant4_Assembly**(*name*, *greg*)

Make a assembly volume from input or get from registry

Parameters

- **name** (*str*) – Name of logical volume
- **greg** (geant4.Registry) – Geant4 registry

pyg4ometry.convert.oce2Geant4.**oceShape_Geant4_Tessellated**(*name*, *shape*, *greg*, *linDef=0.5*, *angDef=0.5*)

Make a tessellated solid from a OpenCascade shape

Parameters

- **name** (*str*) – Name of logical volume
- **shape** (*TopoDS_Shape*) – OpenCascade shape
- **greg** (geant4.Registry) – Geant4 registry

pyg4ometry.convert.oce2Geant4.**_oce2Geant4_traverse**(*shapeTool*, *label*, *greg*, *materialMap*, *labelToSkipList*, *meshQualityMap*, *badCADLabels*, *addBoundingSolids=False*, *oceName=False*)

pyg4ometry.convert.oce2Geant4.**oce2Geant4**(*shapeTool*, *shapeName*, *materialMap={}*, *labelToSkipList=[]*, *meshQualityMap={}*, *oceName=False*)

Convert CAD geometry starting from shapeName

Parameters

- **shapeTool** (*pyoce.TopoDS_Shape*) – OpenCascade TopoDS_Shape
- **shapeName** (*str*) – Name of the shape in the CAD file
- **materialMap** (*dict*) – dictionary to map shape name to material shapeName:materialName or shapeName:Material
- **meshQualityMap** (*dict*) – dictionary to map shape name to meshing quality str:[LinDef,AngDef]

pyg4ometry.convert.stl2gdml

Module Contents

Functions

stl2gdml(stlFileName[, gdmlFileName, worldMaterial, ...])	

pyg4ometry.convert.stl2gdml.**stl2gdml**(*stlFileName*, *gdmlFileName='output.gdml'*, *worldMaterial='G4_Galactic'*, *solidMaterial='G4_Au'*)

pyg4ometry.convert.vis2oce

Module Contents

Functions

convertMeshToPolyTriangulation(m)	
convertMeshToShapeUsingMakeShapeOnMesh(m)	
convertMeshToShape(m)	
vis2oce(vis[, stepFileName])	

pyg4ometry.convert.vis2oce.**convertMeshToPolyTriangulation**(*m*)

pyg4ometry.convert.vis2oce.**convertMeshToShapeUsingMakeShapeOnMesh**(*m*)

pyg4ometry.convert.vis2oce.**convertMeshToShape**(*m*)

pyg4ometry.convert.vis2oce.**vis2oce**(*vis*, *stepFileName='output.step'*)

Package Contents

Classes

_PolygonProcessing	
RPP	Rectangular Parallelepiped
BOX	General Rectangular Parallelepiped
SPH	Sphere
RCC	Right Circular Cylinder
REC	Right Elliptical Cylinder
TRC	Truncated Right-angled Cone
ELL	Ellipsoid of Revolution
WED	Right Angle Wedge
RAW	Base class representing a body as defined in FLUKA
ARB	Arbitrary Convex Polyhedron
XYP	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)
XZP	Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)
YZP	Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)
PLA	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.
XCC	Infinite Circular Cylinder parallel to the x-axis
YCC	Infinite Circular Cylinder parallel to the y-axis
ZCC	Infinite Circular Cylinder parallel to the z-axis
XEC	Infinite Elliptical Cylinder parallel to the x-axis
YEC	Infinite Elliptical Cylinder parallel to the y-axis
ZEC	Infinite Elliptical Cylinder parallel to the z-axis
QUA	Generic quadric

Functions

fluka2Geant4(flukareg[, regions, omitRegions, ...])	Convert a FLUKA registry to a Geant4 Registry.
_rotoTranslationFromTra2(name, tra2[, flukareg-istry])	
geant4Reg2FlukaReg(greg[, logicalVolumeName])	Convert a Geant4 model to a FLUKA one. This is done by handing over a complete
geant4Logical2Fluka(logicalVolume[, flukaReg-istry])	Convert a single logical volume - not the main entry point for the conversion.

continues on next page

Table 1 – continued from previous page

geant4PhysicalVolume2Fluka(physicalVolume[, mtra, ...])	
geant4Solid2FlukaRegion(flukaNameCount, solid[, mtra, ...])	
geant4MaterialDict2Fluka(matr, freg)	
geant4Material2Fluka(material, freg[, ...])	
pycsgmesh2FlukaRegion(mesh, name, transform, ...)	
makeStripName(mn)	
makeShortName(mn)	
freecadDoc2Fluka(fcd)	
part2Region(obj, trfm, fgreg[, meshDeviation])	
stl2gdml(stlFileName[, gdmlFileName, worldMaterial, ...])	
geant4Solid2Geant4Tessellated(solid)	
geant4Solid2Geant4Tessellated_NoVTK(solid)	
gdml2stl(gdmlFileName[, stlFileName, solidName])	
mkVtkIdList(it)	
pycsgMeshToVtkPolyData(mesh)	
vtkPolyDataToNumpy(data)	
pycsgMeshToObj(mesh, fileName)	
pyg42VtkTransformation(mtra, tra)	
vtkTransformation2PyG4(vt)	
pycsgMeshToStl(mesh, fileName)	
oceShape_Geant4_LogicalVolume(name, solid, material, greg)	Make a logical volume from input or get from registry
oceShape_Geant4_Assembly(name, greg)	Make a assembly volume from input or get from registry
oceShape_Geant4_Tessellated(name, shape, greg[, ...])	Make a tessellated solid from a OpenCascade shape
_oce2Geant4_traverse(shapeTool, label, greg, ...[, ...])	
oce2Geant4	
convertMeshToPolyTriangulation(m)	

continues on next page

Table 1 – continued from previous page

convertMeshToShapeUsingMakeShapeOnMesh(m)
convertMeshToShape(m)
vis2oce

Attributes

defaultLinDef
deftaulAngDef

pyg4ometry.convert.**fluka2Geant4**(*flukareg*, *regions=None*, *omitRegions=None*, *worldMaterial='G4_Galactic'*, *worldDimensions=None*, *omitBlackholeRegions=True*, *quadricRegionAABBs=None*, ***kwargs*)

Convert a FLUKA registry to a Geant4 Registry.

Parameters

- **flukareg** (*FlukaRegistry*) – FlukaRegistry instance to be converted.
- **regions** (*list*) – Names of regions to be converted, by default all are converted. Mutually exclusive with omitRegions.
- **omitRegions** (*list*) – Names of regions to be omitted from the conversion. This option is mutually exclusive with the kwarg regions.
- **worldMaterial** (*string*) – name of world material to be used.
- **worldDimensions** (*list*) – dimensions of world logical volume in converted Geant4. By default this is equal to WORLD_DIMENSIONS.
- **omitBlackholeRegions** (*bool*) – whether or not to omit regions with the FLUKA material BLCKHOLE from the conversion. By default, true.
- **quadricRegionAABBs** (*dict*) – The axis-aligned aabbs of any regions featuring QUA bodies, mapping region names to fluka.AABB instances.

Developer options (to kwargs) withLengthSafety: Whether or not to apply automatic length safety.

minimiseSolids: Whether or not to minimise the boxes and tubes of Geant4 used to represent infinite solids in FLUKA.

class pyg4ometry.convert.**_PolygonProcessing**

classmethod windingNumber(*pgon*)

return the winding number of pgon :param pgon: list of points [[x1,y1], [x2,y2], ...] :type pgon: List[List[x1,y1], ...] returns: Integer winding number

classmethod reversePolygon(*pgon*)

return reversed polygon :param pgon: list of points [[x1,y1], [x2,y2], ...] :type pgon: List[List[x1,y1], ...] returns: List[List[x1,y1], ...]

classmethod makePolygonFromList(*pgon*, *type=''*)

Convert list of points [[x1,y1], [x2,y2], …] to cgal Polygon_2

Parameters

- **pgon** (*List[List[x,y], ..]*) – list of points [[x1,y1], [x2,y2], …]
- **type** – Class of polygon (Polygon_2_EPICK, Polygon_2_EPECK, Partition_traits_2_Polygon_2_EPECK)
- **type** – str

returns: Polygon_2

classmethod makeListFromPolygon(*pgon*)

Convert 2D polygon to list of points [[x1,y1], [x2,y2], …]

Parameters
pgon (*Polygon_2_EPECK or Polygon_2_EPICK*) – cgal Polygon_2 input

returns: [[x1,y1], [x2,y2], …]

classmethod decomposePolygon2d(*pgon*)

Decompose general 2D polygon (pgon) to convex 2D polygons

Parameters
pgon (*List(List[2])*) – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of polgons [pgon1, pgon2, …]

classmethod decomposePolygon2dWithHoles(*pgonOuter*, *pgonHoles*)

Decompose general 2D polygon with holes (pgon) to convex 2D polygons

Parameters

- **pgonOuter** – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]
- **pgonHoles** – List of polgons [pgon1, pgon2, …]

returns: List of polgons [pgon1, pgon2, …]

classmethod triangulatePolygon2d(*pgon*)

Triangulate general 2D polygon

Parameters
pgonOuter – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of triangles [[[x1,y1], [x2,y2], [x3,y3]], [[x1,y1], [x2,y2], [x3,y3]], …]

pyg4ometry.convert.**_rotoTranslationFromTra2**(*name*, *tra2*, *flukaregistry=None*)

pyg4ometry.convert.**geant4Reg2FlukaReg**(*greg*, *logicalVolumeName=''*)

Convert a Geant4 model to a FLUKA one. This is done by handing over a complete pyg4ometry.geant4.Registry instance.

Parameters
greg (pyg4ometry.geant4.Registry) – geant4 registry

returns: pyg4ometry.fluka.FlukaRegistry

pyg4ometry.convert.**geant4Logical2Fluka**(*logicalVolume*, *flukaRegistry=None*)

Convert a single logical volume - not the main entry point for the conversion.

pyg4ometry.convert.**geant4PhysicalVolume2Fluka**(*physicalVolume*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *flukaNameCount=0*)

pyg4ometry.convert.**geant4Solid2FlukaRegion**(*flukaNameCount*, *solid*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *flukaRegistry=None*, *addRegistry=True*, *commentName=''*)

pyg4ometry.convert.**geant4MaterialDict2Fluka**(*matr*, *freg*)

pyg4ometry.convert.**geant4Material2Fluka**(*material*, *freg*, *suggestedDensity=None*, *elementSuffix=False*, *materialNameShort=None*)

pyg4ometry.convert.**pycsgmesh2FlukaRegion**(*mesh*, *name*, *transform*, *flukaRegistry*, *commentName*)

pyg4ometry.convert.**makeStripName**(*mn*)

pyg4ometry.convert.**makeShortName**(*mn*)

class pyg4ometry.convert.**RPP**(*name*, *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *transform=None*, *flukaregistry=None*, *addRegistry=True*, *comment=''*)

Bases: BodyMixin

Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **xmin** (*float*) – lower x coordinate of RPP
- **xmax** (*float*) – upper x coordinate of RPP
- **ymin** (*float*) – lower y coordinate of RPP
- **ymax** (*float*) – upper y coordinate of RPP
- **zmin** (*float*) – lower z coordinate of RPP
- **zmax** (*float*) – upper z coordinate of RPP)

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**BOX**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

General Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the corners.
- **edge1** (*list*) – vector [x, y, z] denoting the first side of the box.
- **edge2** (*list*) – vector [x, y, z] denoting the second side of the box.
- **edge3** (*list*) – vector [x, y, z] denoting the second side of the box.

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**SPH**(*name*, *point*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Sphere

Parameters

- **name** (*str*) – of body
- **point** (*list*) – position [x, y, z] of the centre of the sphere.
- **radius** (*float*) – radius of the sphere.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**RCC**(*name*, *face*, *direction*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Circular Cylinder

Parameters

- **name** (*str*) – of body

- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **edge1** (*list*) – vector [x, y, z] denoting the direction along the length of the cylinder.
- **edge2** (*float*) – radius of the cylinder face.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**REC**(*name*, *face*, *direction*, *semiminor*, *semimajor*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Elliptical Cylinder

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **semiminor** (*list*) – vector [x, y, z] denoting the direction along the semiminor axis of the ellipse.
- **semimajor** (*list*) – vector [x, y, z] denoting the direction along the semimajor axis of the ellipse.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**TRC**(*name*, *major_centre*, *direction*, *major_radius*, *minor_radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Truncated Right-angled Cone

Parameters

- **name** (*str*) – of body
- **major_centre** (*list*) – vector [x, y, z] position of the centre of the larger face.
- **direction** (*list*) – vector [x, y, z] pointing from the larger face to the smaller face.

- **major_radius** (*float*) – radius of the larger face.
- **minor_radius** (*float*) – radius of the smaller face.

centre(*aabb=None*)

rotation()

geant4Solid(*registry*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**ELL**(*name*, *focus1*, *focus2*, *length*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Ellipsoid of Revolution

Parameters

- **name** (*str*) – of body
- **focus1** (*list*) – position [x, y, z] denoting of one of the foci.
- **focus2** (*list*) – position [x, y, z] denoting the other focus.
- **length** (*float*) – length of the ellipse axis which the foci lie on.

centre(*aabb=None*)

rotation()

_linearEccentricity()

_semiminor()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**WED**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Right Angle Wedge

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the the rectangular corners.
- **edge1** (*list*) – vector [x, y, z] denoting height of the wedge.

- **edge2** (*list*) – vector [x, y, z] denoting width of the wedge.
- **edge3** (*list*) – vector [x, y, z] denoting length of the wedge.

class pyg4ometry.convert.**RAW**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Base class representing a body as defined in FLUKA

__doc__

class pyg4ometry.convert.**ARB**(*name*, *vertices*, *facenumbers*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Arbitrary Convex Polyhedron

Parameters

- **name** (*str*) – of body
- **vertices** (*list*) – Eight vertices which make up the polyhedron as [[x1, y1, z1], [x2, y2, z2], ...]. There must be eight even if only six or seven vertices are needed to make up the polydedron.
- **facenumbers** (*float*) – The faces of the polyhedron expressed as floats where each digit of the float refers to one of the vertices which makes up that face. Six must always be provided as [1234,8765, ...], even if only four or five faces are needed. Any unneeded faces must be set to 0 (no less than 4 sides). Note that the references to the vertices are not zero-counting. The order of the vertices denoted in the facenumbers must be either all clockwise or all anticlockwise, which if not obeyed will result in erroneous output without warning.

centre(*aabb=None*)

rotation()

_faceNumbersToZeroCountingIndices()

_extent()

geant4Solid(*greg*, *aabb=None*)

_toTesselatedSolid(*verticesAndPolygons*, *greg*, *addRegistry*)

_getVerticesAndPolygons()

_toVerticesAndPolygons(*reverse*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.convert.**XYP**(*name*, *z*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the x-y plane on the z-axis. All points less than z are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.convert.**XZP**(*name*, *y*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the x-y plane on the y-axis. All points less than y are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.convert.**YZP**(*name*, *x*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the x-y plane on the x-axis. All points less than x are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.convert.**PLA**(*name*, *normal*, *point*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.

Parameters

- **name** (*str*) – of body
- **normal** (*list*) – position of a point on the plane
- **normal** – vector perpendicular to the face of the plane, pointing away from the contents of the half space.

rotation()

__repr__()

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.convert.**XCC**(*name*, *y*, *z*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()

Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.convert.**YCC**(*name*, *z*, *x*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.convert.**ZCC**(*name*, *x*, *y*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the
- **y** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.convert.**XEC**(*name*, *y*, *z*, *ysemi*, *zsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body

- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.convert.**YEC**(*name*, *z*, *x*, *zsemi*, *xsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.convert.**ZEC**(*name*, *x*, *y*, *xsemi*, *ysemi*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the

- **y** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.convert.**QUA**(*name*, *cxx*, *cyy*, *czz*, *cxy*, *cxz*, *cyz*, *cx*, *cy*, *cz*, *c*, *transform=None*, *flukaregistry=None*, *comment=''*, ***kwargs*)

Bases: BodyMixin

Generic quadric

Parameters

- **name** (*str*) – of body
- **cxx** (*float*) – x^2 coefficient
- **cyy** (*float*) – y^2 coefficient
- **czz** (*float*) – z^2 coefficient
- **cxy** (*float*) – xy coefficient
- **cxz** (*float*) – xz coefficient
- **cyz** (*float*) – yz coefficient
- **cx** (*float*) – x coefficient
- **cy** (*float*) – y coefficient
- **cz** (*float*) – z coefficient
- **c** (*constant*) – constant

centre(*aabb=None*)

rotation()

coefficientsMatrix()

static **_quadricMatrixToCoefficients**(*matrix*)

mesh(*lower*, *upper*, *capping=True*)

geant4Solid(*reg*, *aabb=None*)

_withLengthSafety(*safety*, *reg=None*)

__repr__()

flukaFreeString()

hash()

pyg4ometry.convert.**freecadDoc2Fluka**(*fcd*)

pyg4ometry.convert.**part2Region**(*obj*, *trfm*, *fgreg*, *meshDeviation=0.05*)

pyg4ometry.convert.**stl2gdml**(*stlFileName*, *gdmlFileName='output.gdml'*, *worldMaterial='G4_Galactic'*, *solidMaterial='G4_Au'*)

pyg4ometry.convert.**geant4Solid2Geant4Tessellated**(*solid*)

pyg4ometry.convert.**geant4Solid2Geant4Tessellated_NoVTK**(*solid*)

pyg4ometry.convert.**gdml2stl**(*gdmlFileName*, *stlFileName='output.gdml'*, *solidName='ws'*)

pyg4ometry.convert.**mkVtkIdList**(*it*)

pyg4ometry.convert.**pycsgMeshToVtkPolyData**(*mesh*)

pyg4ometry.convert.**vtkPolyDataToNumpy**(*data*)

pyg4ometry.convert.**pycsgMeshToObj**(*mesh*, *fileName*)

pyg4ometry.convert.**pyg42VtkTransformation**(*mtra*, *tra*)

pyg4ometry.convert.**vtkTransformation2PyG4**(*vt*)

pyg4ometry.convert.**pycsgMeshToStl**(*mesh*, *fileName*)

pyg4ometry.convert.**defaultLinDef = 0.5**

pyg4ometry.convert.**deftaulAngDef = 0.5**

pyg4ometry.convert.**oceShape_Geant4_LogicalVolume**(*name*, *solid*, *material*, *greg*)

Make a logical volume from input or get from registry

Parameters

- **name** (*str*) – Name of logical volume
- **solid** (SolidBase) – Geant4 solid
- **material** (*str or* pyg4ometry.geant4.Material) – Material for logical volume
- **greg** (geant4.Registry) – Geant4 registry

pyg4ometry.convert.**oceShape_Geant4_Assembly**(*name*, *greg*)

Make a assembly volume from input or get from registry

Parameters

- **name** (*str*) – Name of logical volume
- **greg** (geant4.Registry) – Geant4 registry

pyg4ometry.convert.**oceShape_Geant4_Tessellated**(*name*, *shape*, *greg*, *linDef=0.5*, *angDef=0.5*)

Make a tessellated solid from a OpenCascade shape

Parameters

- **name** (*str*) – Name of logical volume
- **shape** (*TopoDS_Shape*) – OpenCascade shape
- **greg** (*geant4.Registry*) – Geant4 registry

pyg4ometry.convert.**_oce2Geant4_traverse**(*shapeTool*, *label*, *greg*, *materialMap*, *labelToSkipList*, *meshQualityMap*, *badCADLabels*, *addBoundingSolids=False*, *oceName=False*)

pyg4ometry.convert.**oce2Geant4**(*shapeTool*, *shapeName*, *materialMap={}*, *labelToSkipList=[]*, *meshQualityMap={}*, *oceName=False*)

Convert CAD geometry starting from shapeName

Parameters

- **shapeTool** (*pyoce.TopoDS_Shape*) – OpenCascade TopoDS_Shape
- **shapeName** (*str*) – Name of the shape in the CAD file
- **materialMap** (*dict*) – dictionary to map shape name to material shapeName:materialName or shapeName:Material
- **meshQualityMap** (*dict*) – dictionary to map shape name to meshing quality str:[LinDef,AngDef]

pyg4ometry.convert.**convertMeshToPolyTriangulation**(*m*)

pyg4ometry.convert.**convertMeshToShapeUsingMakeShapeOnMesh**(*m*)

pyg4ometry.convert.**convertMeshToShape**(*m*)

pyg4ometry.convert.**vis2oce**(*vis*, *stepFileName='output.step'*)

pyg4ometry.features

Submodules

pyg4ometry.features._accelerator

Module Contents

Functions

beamPipeCADFeature(shape)	
beamPipe(stlFileName[, feature1, feature2, ...])	

pyg4ometry.features._accelerator.**beamPipeCADFeature**(*shape*)

pyg4ometry.features._accelerator.**beamPipe**(*stlFileName*, *feature1=-1*, *feature2=-1*, *planeAngles=[[0, 0, 0]]*, *vis=True*, *interactive=True*)

pyg4ometry.features.algos

Module Contents

Classes

Line	Line class taking a point and dir(ection)
Plane	Plane class taking a point on plane and normal
CoordinateSystem	
vtkViewer	Simple visualiser for feature extraction only
FeatureData	

Functions

vtkLoadStl(fname)	Load the given STL file, and return a vtkPolyData object for it.
vtkPolydataToActor(polydata)	Wrap the provided vtkPolyData object in a mapper and an actor, returning the actor.
vtkPolydataToConnectedEdges(polydata[, angle])	
_vtkPolydataToConnectedEdges(polydata[, angle])	Feature extract from polydata given an angular cut
vtkPolydataEdgeInformation(polydata[, bPlot])	Calculate path information for 2D vtkPolydata lines
_vtkPolydataEdgeInformation(polydata[, bPlot])	Calculate path information for 2D vtkPolydata lines
pyg4PlaneToVtkPlane(pyg4Plane)	Convert Plane to vtkPlane
pyg4ArrayToVtkPolydataLine(pyg4Array)	Convert data array to vtkPolydata
vtkCutterPlane(plane, polydata)	Calculate path information for 2D vtkPolydata lines
_vtkCutterPlane(plane, polydata)	Cutter plane on polydata
extract(inputFileName[, angle, planeQuality, ...])	
test(fileName[, featureIndexList, planeQuality, ...])	

class pyg4ometry.features.algos.**Line**(*point*, *dir*)

Line class taking a point and dir(ection)

Parameters
dir (*list or array - 3 values*) – direction

eval(*par*)

intersectPlane(*plane*)

Intersection of line (self) with plane (unimplemented)

Parameters
plane (Plane) – plane object

Returns
Point of intersection between plane and line

Return type
list or array

`__repr__`()
Return repr(self).

`__iter__`()

`__next__`()

`flatten`()

class pyg4ometry.features.algos.**Plane**(*point*, *normal*, *e2=None*, *e3=None*)
Plane class taking a point on plane and normal

Parameters

- **point** (*list, array*) – point on plane
- **normal** (*list, array*) – vector of normal

`intersect`(*object*)
Intersection between Line or Plane

Parameters
object (*Line or Plane*) – Object to intersect with Plane (either Line or Plane)

`intersectLine`(*line*)
Intersection between plane (self) and line

`intersectPlane`(*plane*)
Compute intersection between two planes self and plane

`angleBetween`(*plane*)
Compute angle between two planes (self and plane)

Parameters
plane (Plane) – Plane to intersect with self

Returns
angle between two planes

Return type
float

`__repr__`()
Return repr(self).

`__iter__`()

`__next__`()

`flatten`()

class pyg4ometry.features.algos.**CoordinateSystem**(*origin=[0, 0, 0]*, *e1=[1, 0, 0]*, *e2=[0, 1, 0]*)

`_commonInit`()

makeFromPlanes(*p1*, *p2*, *p3*)

coordinateSystem(*rotX=0.0*, *rotY=0.0*, *rotZ=0.0*)

plane(*rotX=0.0*, *rotY=0.0*, *rotZ=0.0*)

transform(*points*)

__repr__()
Return repr(self).

class pyg4ometry.features.algos.**vtkViewer**
Simple visualiser for feature extraction only

_polyDataToActor(*polydata*, *colour=[0, 0, 0, 1]*, *lineWidth=5*)
Wrap the provided vtkPolyData object in a mapper and an actor, returning the actor.

addPolydata(*key*, *polydata*, *colour=[0, 0, 0, 1]*, *lineWidth=5*, *label=None*, *labelPos=[0, 0, 0]*)

addAxis(*origin*, *length=[1, 1, 1]*, *rotation=[[1, 0, 0], [0, 1, 0], [0, 0, 1]]*, *label=False*, *disableCone=False*)

addPlane(*point*, *p1*, *p2*, *length=1*)

addLine(*line*)

view(*interactive=True*)

pyg4ometry.features.algos.**vtkLoadStl**(*fname*)
Load the given STL file, and return a vtkPolyData object for it.

pyg4ometry.features.algos.**vtkPolydataToActor**(*polydata*)
Wrap the provided vtkPolyData object in a mapper and an actor, returning the actor.

pyg4ometry.features.algos.**vtkPolydataToConnectedEdges**(*polydata*, *angle=89*)

pyg4ometry.features.algos.**_vtkPolydataToConnectedEdges**(*polydata*, *angle=89*)
Feature extract from polydata given an angular cut

Parameters
angle (*float*) – Angle between plane > angle is returned

Returns
Connected boundaries with > angle

Return type
list of vtkPolydata

pyg4ometry.features.algos.**vtkPolydataEdgeInformation**(*polydata*, *bPlot=False*)
Calculate path information for 2D vtkPolydata lines

Parameters
polydata (*vtkPlydata or list of vtkPolydata*) – Geometry to extract information

Returns
Information on each boundary

Return type
list of information dict

pyg4ometry.features.algos.**_vtkPolydataEdgeInformation**(*polydata*, *bPlot=False*)
Calculate path information for 2D vtkPolydata lines

pyg4ometry.features.algos.**pyg4PlaneToVtkPlane**(*pyg4Plane*)

Convert Plane to vtkPlane

pyg4ometry.features.algos.**pyg4ArrayToVtkPolydataLine**(*pyg4Array*)

Convert data array to vtkPolydata

pyg4ometry.features.algos.**vtkCutterPlane**(*plane*, *polydata*)

Calculate path information for 2D vtkPolydata lines

Parameters
polydata (*vtkPlydata or list of vtkPolydata*) – Geometry to extract information

Returns
Information on each boundary

Return type
list of information dict

pyg4ometry.features.algos.**_vtkCutterPlane**(*plane*, *polydata*)

Cutter plane on polydata

pyg4ometry.features.algos.**extract**(*inputFileName*, *angle=89*, *planeQuality=0.1*, *circumference=300*, *outputFileName=None*, *planes=[]*, *bViewer=False*, *bViewerInteractive=False*)

class pyg4ometry.features.algos.**FeatureData**

readFile(*fileName*)

plotFeature(*iFeature*)

plotCutter(*iCutter*)

pyg4ometry.features.algos.**test**(*fileName*, *featureIndexList=[]*, *planeQuality=0.1*, *circumference=300*, *bPlotRadii=False*)

Package Contents

Classes

CoordinateSystem	
Plane	Plane class taking a point on plane and normal
FeatureData	

Functions

extract(inputFileName[, angle, planeQuality, ...])	

class pyg4ometry.features.**CoordinateSystem**(*origin=[0, 0, 0]*, *e1=[1, 0, 0]*, *e2=[0, 1, 0]*)

_commonInit()

makeFromPlanes(*p1*, *p2*, *p3*)

coordinateSystem(*rotX=0.0*, *rotY=0.0*, *rotZ=0.0*)

plane(*rotX=0.0*, *rotY=0.0*, *rotZ=0.0*)

transform(*points*)

__repr__()

Return repr(self).

class pyg4ometry.features.**Plane**(*point*, *normal*, *e2=None*, *e3=None*)

Plane class taking a point on plane and normal

Parameters

- **point** (*list, array*) – point on plane
- **normal** (*list, array*) – vector of normal

intersect(*object*)

Intersection between Line or Plane

Parameters
object (*Line or Plane*) – Object to intersect with Plane (either Line or Plane)

intersectLine(*line*)

Intersection between plane (self) and line

intersectPlane(*plane*)

Compute intersection between two planes self and plane

angleBetween(*plane*)

Compute angle between two planes (self and plane)

Parameters
plane (Plane) – Plane to intersect with self

Returns
angle between two planes

Return type
float

__repr__()

Return repr(self).

__iter__()

__next__()

flatten()

pyg4ometry.features.**extract**(*inputFileName*, *angle=89*, *planeQuality=0.1*, *circumference=300*, *outputFileName=None*, *planes=[]*, *bViewer=False*, *bViewerInteractive=False*)

class pyg4ometry.features.**FeatureData**

readFile(*fileName*)

plotFeature(*iFeature*)

plotCutter(*iCutter*)

pyg4ometry.fluka

Subpackages

pyg4ometry.fluka.RegionExpression

Submodules

pyg4ometry.fluka.RegionExpression.RegionLexer

Module Contents

Classes

RegionLexer

Functions

serializedATN()

pyg4ometry.fluka.RegionExpression.RegionLexer.**serializedATN**()

class pyg4ometry.fluka.RegionExpression.RegionLexer.**RegionLexer**(*input=None*, *output=sys.stdout*)

Bases: Lexer

Parameters
output (*TextIO*) –

atn

decisionsToDFA

Whitespace = 1

InLineComment = 2

LineComment = 3

Newline = 4

Integer = 5

RegionName = 6

BodyName = 7

Plus = 8

Minus = 9

Bar = 10

LParen = 11

RParen = 12

channelNames = ['DEFAULT_TOKEN_CHANNEL', 'HIDDEN']

modeNames = ['DEFAULT_MODE']

literalNames = ['<INVALID>', "'+'", "'-'", "'|'", "'('", "')'"]

symbolicNames = ['<INVALID>', 'Whitespace', 'InLineComment', 'LineComment',
'Newline', 'Integer', 'RegionName',...

ruleNames = ['Whitespace', 'InLineComment', 'LineComment', 'Newline', 'Integer',
'Digit', 'RegionName',...

grammarFileName = 'RegionLexer.g4'

sempred(*localctx*, *ruleIndex*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **ruleIndex** (*int*) –
- **predIndex** (*int*) –

LineComment_sempred(*localctx*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **predIndex** (*int*) –

RegionName_sempred(*localctx*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **predIndex** (*int*) –

pyg4ometry.fluka.RegionExpression.RegionParser

Module Contents

Classes

RegionParser

Functions

serializedATN()

pyg4ometry.fluka.RegionExpression.RegionParser.**serializedATN**()

class pyg4ometry.fluka.RegionExpression.RegionParser.**RegionParser**(*input*, *output=sys.stdout*)

Bases: Parser

Parameters

- **input** (*TokenStream*) –
- **output** (*TextIO*) –

class RegionsContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

region(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class RegionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class ComplexRegionContext(*parser*, *ctx*)

Bases: *RegionContext*

Parameters
ctx (*ParserRuleContext*) –

RegionName()

Integer()

zoneUnion()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SimpleRegionContext(*parser*, *ctx*)

Bases: *RegionContext*

Parameters
ctx (*ParserRuleContext*) –

RegionName()

Integer()

zone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneUnionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class MultipleUnion2Context(*parser*, *ctx*)

Bases: *ZoneUnionContext*

Parameters
ctx (*ParserRuleContext*) –

zone(*i=None*)

Parameters
i (*Optional[int]*) –

Bar(*i=None*)

Parameters
i (*Optional[int]*) –

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class MultipleUnionContext(*parser*, *ctx*)

Bases: *ZoneUnionContext*

Parameters
ctx (*ParserRuleContext*) –

Bar(*i=None*)

Parameters
i (*Optional[int]*) –

zone(*i=None*)

Parameters
i (*Optional[int]*) –

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SingleUnionContext(*parser*, *ctx*)

Bases: *ZoneUnionContext*

Parameters
ctx (*ParserRuleContext*) –

Bar()

zone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class ZoneExprContext(*parser*, *ctx*)

Bases: *ZoneContext*

Parameters
ctx (*ParserRuleContext*) –

expr()

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneBodyContext(*parser*, *ctx*)

Bases: *ZoneContext*

Parameters
ctx (*ParserRuleContext*) –

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneSubZoneContext(*parser*, *ctx*)

Bases: *ZoneContext*

Parameters
ctx (*ParserRuleContext*) –

subZone()

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ExprContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class UnaryAndBooleanContext(*parser*, *ctx*)

Bases: *ExprContext*

Parameters
ctx (*ParserRuleContext*) –

unaryExpression()

expr()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class OneSubZoneContext(*parser*, *ctx*)

Bases: *ExprContext*

Parameters
ctx (*ParserRuleContext*) –

subZone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class UnaryAndSubZoneContext(*parser*, *ctx*)

Bases: *ExprContext*

Parameters
ctx (*ParserRuleContext*) –

subZone()

expr()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SingleUnaryContext(*parser*, *ctx*)

Bases: *ExprContext*

Parameters
ctx (*ParserRuleContext*) –

unaryExpression()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SubZoneContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

LParen()

expr()

RParen()

Minus()

Plus()

BodyName()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class **UnaryExpressionContext**(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

BodyName()

Minus()

Plus()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

grammarFileName = 'RegionParser.g4'

atn

decisionsToDFA

sharedContextCache

literalNames = ['<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>',...

symbolicNames = ['<INVALID>', 'Whitespace', 'InLineComment', 'LineComment', 'Newline', 'Integer', 'RegionName',...

```
    RULE_regions = 0

    RULE_region = 1

    RULE_zoneUnion = 2

    RULE_zone = 3

    RULE_expr = 4

    RULE_subZone = 5

    RULE_unaryExpression = 6

    ruleNames = ['regions', 'region', 'zoneUnion', 'zone', 'expr', 'subZone',
    'unaryExpression']

    EOF

    Whitespace = 1

    InLineComment = 2

    LineComment = 3

    Newline = 4

    Integer = 5

    RegionName = 6

    BodyName = 7

    Plus = 8

    Minus = 9

    Bar = 10

    LParen = 11

    RParen = 12

    regions()

    region()

    zoneUnion()

    zone()

    expr()

    subZone()

    unaryExpression()
```

pyg4ometry.fluka.RegionExpression.RegionParserVisitor

Module Contents

Classes

RegionParserVisitor

class pyg4ometry.fluka.RegionExpression.RegionParserVisitor.**RegionParserVisitor**

Bases: ParseTreeVisitor

visitRegions(*ctx*)

Parameters
ctx (RegionParser.RegionsContext) –

visitSimpleRegion(*ctx*)

Parameters
ctx (RegionParser.SimpleRegionContext) –

visitComplexRegion(*ctx*)

Parameters
ctx (RegionParser.ComplexRegionContext) –

visitMultipleUnion(*ctx*)

Parameters
ctx (RegionParser.MultipleUnionContext) –

visitSingleUnion(*ctx*)

Parameters
ctx (RegionParser.SingleUnionContext) –

visitMultipleUnion2(*ctx*)

Parameters
ctx (RegionParser.MultipleUnion2Context) –

visitZoneExpr(*ctx*)

Parameters
ctx (RegionParser.ZoneExprContext) –

visitZoneSubZone(*ctx*)

Parameters
ctx (RegionParser.ZoneSubZoneContext) –

visitZoneBody(*ctx*)

Parameters
ctx (RegionParser.ZoneBodyContext) –

visitSingleUnary(*ctx*)

Parameters
ctx (RegionParser.SingleUnaryContext) –

visitUnaryAndBoolean(*ctx*)

Parameters
ctx (RegionParser.UnaryAndBooleanContext) –

visitUnaryAndSubZone(*ctx*)

Parameters
ctx (RegionParser.UnaryAndSubZoneContext) –

visitOneSubZone(*ctx*)

Parameters
ctx (RegionParser.OneSubZoneContext) –

visitSubZone(*ctx*)

Parameters
ctx (RegionParser.SubZoneContext) –

visitUnaryExpression(*ctx*)

Parameters
ctx (RegionParser.UnaryExpressionContext) –

Package Contents

Classes

RegionParserVisitor
RegionParser
RegionLexer

class pyg4ometry.fluka.RegionExpression.**RegionParserVisitor**

Bases: ParseTreeVisitor

visitRegions(*ctx*)

Parameters
ctx (RegionParser.RegionsContext) –

visitSimpleRegion(*ctx*)

Parameters
ctx (RegionParser.SimpleRegionContext) –

visitComplexRegion(*ctx*)

Parameters
ctx (RegionParser.ComplexRegionContext) –

visitMultipleUnion(*ctx*)

Parameters
ctx (RegionParser.MultipleUnionContext) –

visitSingleUnion(*ctx*)

Parameters
ctx (RegionParser.SingleUnionContext) –

visitMultipleUnion2(*ctx*)

Parameters
ctx (RegionParser.MultipleUnion2Context) –

visitZoneExpr(*ctx*)

Parameters
ctx (RegionParser.ZoneExprContext) –

visitZoneSubZone(*ctx*)

Parameters
ctx (RegionParser.ZoneSubZoneContext) –

visitZoneBody(*ctx*)

Parameters
ctx (RegionParser.ZoneBodyContext) –

visitSingleUnary(*ctx*)

Parameters
ctx (RegionParser.SingleUnaryContext) –

visitUnaryAndBoolean(*ctx*)

Parameters
ctx (RegionParser.UnaryAndBooleanContext) –

visitUnaryAndSubZone(*ctx*)

Parameters
ctx (RegionParser.UnaryAndSubZoneContext) –

visitOneSubZone(*ctx*)

Parameters
ctx (RegionParser.OneSubZoneContext) –

visitSubZone(*ctx*)

Parameters
ctx (RegionParser.SubZoneContext) –

visitUnaryExpression(*ctx*)

Parameters
ctx (RegionParser.UnaryExpressionContext) –

class pyg4ometry.fluka.RegionExpression.**RegionParser**(*input*, *output=sys.stdout*)

Bases: Parser

Parameters

- **input** (*TokenStream*) –
- **output** (*TextIO*) –

class **RegionsContext**(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

region(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class **RegionContext**(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class **ComplexRegionContext**(*parser*, *ctx*)

Bases: RegionContext

Parameters
ctx (*ParserRuleContext*) –

RegionName()

Integer()

zoneUnion()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SimpleRegionContext(*parser*, *ctx*)

Bases: RegionContext

Parameters
ctx (*ParserRuleContext*) –

RegionName()

Integer()

zone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneUnionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class MultipleUnion2Context(*parser*, *ctx*)

Bases: ZoneUnionContext

Parameters
ctx (*ParserRuleContext*) –

zone(*i=None*)

Parameters
i (*Optional[int]*) –

Bar(*i=None*)

Parameters
i (*Optional[int]*) –

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class MultipleUnionContext(*parser*, *ctx*)

Bases: ZoneUnionContext

Parameters
ctx (*ParserRuleContext*) –

Bar(*i=None*)

Parameters
i (*Optional[int]*) –

zone(*i=None*)

Parameters
i (*Optional[int]*) –

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SingleUnionContext(*parser*, *ctx*)

Bases: ZoneUnionContext

Parameters
ctx (*ParserRuleContext*) –

Bar()

zone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class ZoneExprContext(*parser*, *ctx*)

Bases: ZoneContext

Parameters
ctx (*ParserRuleContext*) –

expr()

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneBodyContext(*parser*, *ctx*)

Bases: ZoneContext

Parameters
ctx (*ParserRuleContext*) –

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ZoneSubZoneContext(*parser*, *ctx*)

Bases: ZoneContext

Parameters
ctx (*ParserRuleContext*) –

subZone()

BodyName()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ExprContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

getRuleIndex()

copyFrom(*ctx*)

Parameters
ctx (*ParserRuleContext*) –

class UnaryAndBooleanContext(*parser*, *ctx*)

Bases: ExprContext

Parameters
ctx (*ParserRuleContext*) –

unaryExpression()

expr()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class OneSubZoneContext(*parser*, *ctx*)

Bases: ExprContext

Parameters
ctx (*ParserRuleContext*) –

subZone()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class UnaryAndSubZoneContext(*parser*, *ctx*)

Bases: ExprContext

Parameters
ctx (*ParserRuleContext*) –

subZone()

expr()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SingleUnaryContext(*parser*, *ctx*)

Bases: ExprContext

Parameters
ctx (*ParserRuleContext*) –

unaryExpression()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SubZoneContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

LParen()

expr()

RParen()

Minus()

Plus()

BodyName()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class UnaryExpressionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

BodyName()

Minus()

Plus()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

grammarFileName = 'RegionParser.g4'

atn

decisionsToDFA

sharedContextCache

literalNames = ['<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>', '<INVALID>',
'<INVALID>', '<INVALID>',...

symbolicNames = ['<INVALID>', 'Whitespace', 'InLineComment', 'LineComment',
'Newline', 'Integer', 'RegionName',...

RULE_regions = 0

RULE_region = 1

RULE_zoneUnion = 2

RULE_zone = 3

RULE_expr = 4

RULE_subZone = 5

RULE_unaryExpression = 6

ruleNames = ['regions', 'region', 'zoneUnion', 'zone', 'expr', 'subZone',
'unaryExpression']

EOF

Whitespace = 1

InLineComment = 2

LineComment = 3

Newline = 4

Integer = 5

RegionName = 6

BodyName = 7

Plus = 8

Minus = 9

Bar = 10

LParen = 11

RParen = 12

regions()

region()

zoneUnion()

zone()

expr()

subZone()

unaryExpression()

class pyg4ometry.fluka.RegionExpression.**RegionLexer**(*input=None*, *output=sys.stdout*)

Bases: Lexer

Parameters
output (*TextIO*) –

atn

decisionsToDFA

Whitespace = 1

InLineComment = 2

LineComment = 3

Newline = 4

Integer = 5

RegionName = 6

BodyName = 7

Plus = 8

Minus = 9

Bar = 10

LParen = 11

RParen = 12

channelNames = ['DEFAULT_TOKEN_CHANNEL', 'HIDDEN']

modeNames = ['DEFAULT_MODE']

literalNames = ['<INVALID>', "'+'", "'-'", "'|'", "'('", "')'"]

symbolicNames = ['<INVALID>', 'Whitespace', 'InLineComment', 'LineComment', 'Newline', 'Integer', 'RegionName',...

ruleNames = ['Whitespace', 'InLineComment', 'LineComment', 'Newline', 'Integer', 'Digit', 'RegionName',...

grammarFileName = 'RegionLexer.g4'

sempred(*localctx*, *ruleIndex*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **ruleIndex** (*int*) –
- **predIndex** (*int*) –

LineComment_sempred(*localctx*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **predIndex** (*int*) –

RegionName_sempred(*localctx*, *predIndex*)

Parameters

- **localctx** (*RuleContext*) –
- **predIndex** (*int*) –

Submodules

pyg4ometry.fluka.Writer

```
GLOBAL        20000.       -1.        1.        0.        0.        0.
DEFAULTS         0.0       0.0       0.0       0.0       0.0       0.0 EM-CASCA
BEAM            17.5       0.0   10000.0       0.0       0.0       1.0 ELECTRON
GEOBEGIN           3.                                                   COMBNAME
              0    0

USRBIN    , 10., ELECTRON, -21., 250., 0.1, 2000., ElecTop
USRBIN    , -250., -0.1, -2000., 400., 1., 400.,  &
USRBIN    , 10., POSITRON, -22., 250., 0.1, 2000., PosTop
USRBIN    , -250., -0.1, -2000., 400., 1., 400.,  &
USRBIN    , 10., PHOTON, -23., 250., 0.1, 2000., PhotTop
USRBIN    , -250., -0.1, -2000., 400., 1., 400.,  &
RANDOMIZ  , 1.
START     , 1000.
STOP
END
```

Module Contents

Classes

Writer	Class to write FLUKA input files from a fluka registry object.

class pyg4ometry.fluka.Writer.**Writer**

Class to write FLUKA input files from a fluka registry object.

```
>>> f = Writer()
>>> f.addDetectro(flukaRegObject)
>>> f.write("model.inp")
```

_flukaFFString =
'*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...'

addDetector(*flukaRegistry*)

Set the fluka registry and therefore the model for this writer instance.

write(*fileName*)

Write the output to a given filename. e.g. “model.inp”.

pyg4ometry.fluka.body

Set of classes for FLUKA bodies.

Module Contents

Classes

RPP	Rectangular Parallelepiped
BOX	General Rectangular Parallelepiped
SPH	Sphere
RCC	Right Circular Cylinder
REC	Right Elliptical Cylinder
TRC	Truncated Right-angled Cone
ELL	Ellipsoid of Revolution
WED	Right Angle Wedge
RAW	Base class representing a body as defined in FLUKA
ARB	Arbitrary Convex Polyhedron
XYP	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)
XZP	Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)
YZP	Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)
PLA	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.
XCC	Infinite Circular Cylinder parallel to the x-axis
YCC	Infinite Circular Cylinder parallel to the y-axis
ZCC	Infinite Circular Cylinder parallel to the z-axis
XEC	Infinite Elliptical Cylinder parallel to the x-axis
YEC	Infinite Elliptical Cylinder parallel to the y-axis
ZEC	Infinite Elliptical Cylinder parallel to the z-axis
QUA	Generic quadric

class pyg4ometry.fluka.body.**RPP**(*name*, *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *transform=None*, *flukaregistry=None*, *addRegistry=True*, *comment=''*)

Bases: BodyMixin

Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **xmin** (*float*) – lower x coordinate of RPP
- **xmax** (*float*) – upper x coordinate of RPP
- **ymin** (*float*) – lower y coordinate of RPP
- **ymax** (*float*) – upper y coordinate of RPP
- **zmin** (*float*) – lower z coordinate of RPP
- **zmax** (*float*) – upper z coordinate of RPP)

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.**BOX**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

General Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the corners.
- **edge1** (*list*) – vector [x, y, z] denoting the first side of the box.
- **edge2** (*list*) – vector [x, y, z] denoting the second side of the box.
- **edge3** (*list*) – vector [x, y, z] denoting the second side of the box.

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.**SPH**(*name*, *point*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Sphere

Parameters

- **name** (*str*) – of body
- **point** (*list*) – position [x, y, z] of the centre of the sphere.
- **radius** (*float*) – radius of the sphere.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.RCC(*name*, *face*, *direction*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Circular Cylinder

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **edge1** (*list*) – vector [x, y, z] denoting the direction along the length of the cylinder.
- **edge2** (*float*) – radius of the cylinder face.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.REC(*name*, *face*, *direction*, *semiminor*, *semimajor*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Elliptical Cylinder

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **semiminor** (*list*) – vector [x, y, z] denoting the direction along the semiminor axis of the ellipse.
- **semimajor** (*list*) – vector [x, y, z] denoting the direction along the semimajor axis of the ellipse.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.TRC(*name*, *major_centre*, *direction*, *major_radius*, *minor_radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Truncated Right-angled Cone

Parameters

- **name** (*str*) – of body
- **major_centre** (*list*) – vector [x, y, z] position of the centre of the larger face.
- **direction** (*list*) – vector [x, y, z] pointing from the larger face to the smaller face.
- **major_radius** (*float*) – radius of the larger face.
- **minor_radius** (*float*) – radius of the smaller face.

centre(*aabb=None*)

rotation()

geant4Solid(*registry*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.ELL(*name*, *focus1*, *focus2*, *length*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Ellipsoid of Revolution

Parameters

- **name** (*str*) – of body
- **focus1** (*list*) – position [x, y, z] denoting of one of the foci.
- **focus2** (*list*) – position [x, y, z] denoting the other focus.
- **length** (*float*) – length of the ellipse axis which the foci lie on.

centre(*aabb=None*)

rotation()

_linearEccentricity()

_semiminor()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.WED(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Right Angle Wedge

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the the rectangular corners.
- **edge1** (*list*) – vector [x, y, z] denoting height of the wedge.
- **edge2** (*list*) – vector [x, y, z] denoting width of the wedge.
- **edge3** (*list*) – vector [x, y, z] denoting length of the wedge.

class pyg4ometry.fluka.body.RAW(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Base class representing a body as defined in FLUKA

__doc__

class pyg4ometry.fluka.body.ARB(*name*, *vertices*, *facenumbers*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Arbitrary Convex Polyhedron

Parameters

- **name** (*str*) – of body
- **vertices** (*list*) – Eight vertices which make up the polyhedron as [[x1, y1, z1], [x2, y2, z2], ...]. There must be eight even if only six or seven vertices are needed to make up the polydedron.
- **facenumbers** (*float*) – The faces of the polyhedron expressed as floats where each digit of the float refers to one of the vertices which makes up that face. Six must always be provided as [1234,8765, ...], even if only four or five faces are needed. Any unneeded faces must be set to 0 (no less than 4 sides). Note that the references to the vertices are not zero-counting. The order of the vertices denoted in the facenumbers must be either all clockwise or all anticlockwise, which if not obeyed will result in erroneous output without warning.

centre(*aabb=None*)

rotation()

_faceNumbersToZeroCountingIndices()

_extent()

geant4Solid(*greg*, *aabb=None*)

_toTesselatedSolid(*verticesAndPolygons*, *greg*, *addRegistry*)

_getVerticesAndPolygons()

_toVerticesAndPolygons(*reverse*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.**XYP**(*name*, *z*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the x-y plane on the z-axis. All points less than z are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.body.**XZP**(*name*, *y*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the x-y plane on the y-axis. All points less than y are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.body.YZP(*name*, *x*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the x-y plane on the x-axis. All points less than x are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.body.PLA(*name*, *normal*, *point*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.

Parameters

- **name** (*str*) – of body
- **normal** (*list*) – position of a point on the plane
- **normal** – vector perpendicular to the face of the plane, pointing away from the contents of the half space.

rotation()

__repr__()

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.body.XCC(*name*, *y*, *z*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.body.**YCC**(*name*, *z*, *x*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.body.**ZCC**(*name*, *x*, *y*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the
- **y** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.body.XEC(*name*, *y*, *z*, *ysemi*, *zsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.YEC(*name*, *z*, *x*, *zsemi*, *xsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.ZEC(*name*, *x*, *y*, *xsemi*, *ysemi*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the
- **y** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.body.QUA(*name*, *cxx*, *cyy*, *czz*, *cxy*, *cxz*, *cyz*, *cx*, *cy*, *cz*, *c*, *transform=None*, *flukaregistry=None*, *comment=''*, ***kwargs*)

Bases: BodyMixin

Generic quadric

Parameters

- **name** (*str*) – of body
- **cxx** (*float*) – x^2 coefficient
- **cyy** (*float*) – y^2 coefficient
- **czz** (*float*) – z^2 coefficient
- **cxy** (*float*) – xy coefficient

- **cxz** (*float*) – xz coefficient
- **cyz** (*float*) – yz coefficient
- **cx** (*float*) – x coefficient
- **cy** (*float*) – y coefficient
- **cz** (*float*) – z coefficient
- **c** (*constant*) – constant

centre(*aabb=None*)

rotation()

coefficientsMatrix()

static _quadricMatrixToCoefficients(*matrix*)

mesh(*lower*, *upper*, *capping=True*)

geant4Solid(*reg*, *aabb=None*)

_withLengthSafety(*safety*, *reg=None*)

__repr__()

flukaFreeString()

hash()

pyg4ometry.fluka.boolean_algebra

Module Contents

Functions

expressionToZone(zone, zoneExpr)	
zoneToDNFZones(zone)	
regionToAlgebraicExpression(region)	
zoneToAlgebraicExpression(zone)	
_getMeshAndAABB(body[, aabb])	
pruneRegion(reg[, aabb])	
isZoneContradiction(zone)	
pruneZone(zone[, aabb0, aabb])	
squashDegenerateBodies(zone[, bodystore])	
simplifyRegion(region)	
_filterRedunantZonesSymbollicaly(zoneData)	
_filterRedundantZonesMetricCheck(zoneData)	
nTermsDNF(expr)	
_nTermsDNFZone(zone)	

Attributes

_MeshedZoneInfo	

pyg4ometry.fluka.boolean_algebra.**expressionToZone**(*zone*, *zoneExpr*)

pyg4ometry.fluka.boolean_algebra.**zoneToDNFZones**(*zone*)

pyg4ometry.fluka.boolean_algebra.**regionToAlgebraicExpression**(*region*)

pyg4ometry.fluka.boolean_algebra.**zoneToAlgebraicExpression**(*zone*)

pyg4ometry.fluka.boolean_algebra.**_getMeshAndAABB**(*body*, *aabb=None*)

pyg4ometry.fluka.boolean_algebra.**pruneRegion**(*reg*, *aabb=None*)

pyg4ometry.fluka.boolean_algebra.**isZoneContradiction**(*zone*)

pyg4ometry.fluka.boolean_algebra.**pruneZone**(*zone*, *aabb0=None*, *aabb=None*)

pyg4ometry.fluka.boolean_algebra.**squashDegenerateBodies**(*zone*, *bodystore=None*)

pyg4ometry.fluka.boolean_algebra.**_MeshedZoneInfo**

pyg4ometry.fluka.boolean_algebra.**simplifyRegion**(*region*)

pyg4ometry.fluka.boolean_algebra.**_filterRedunantZonesSymbollicaly**(*zoneData*)

pyg4ometry.fluka.boolean_algebra.**_filterRedundantZonesMetricCheck**(*zoneData*)

pyg4ometry.fluka.boolean_algebra.**nTermsDNF**(*expr*)

pyg4ometry.fluka.boolean_algebra.**_nTermsDNFZone**(*zone*)

pyg4ometry.fluka.card

Module Contents

Classes

Card	Card class for representing a FLUKA input card. To construct

Functions

_attempt_float_coercion(string)	
freeFormatStringSplit(string)	Method to split a string in FLUKA FREE format into its components.
main(filein)	

class pyg4ometry.fluka.card.**Card**(*keyword*, *what1=None*, *what2=None*, *what3=None*, *what4=None*, *what5=None*, *what6=None*, *sdum=None*)

Card class for representing a FLUKA input card. To construct instances from a of FLUKA input, use the fromFree or fromFixed class method for FREE and FIXED format, respectively.

__repr__()

Return repr(self).

toList()

toFreeString(*delim=', '*)

toFixedString()

nonesToZero()

Return a class instance with same contents as this instance, but with all entries of None set to 0.0 instead.

classmethod fromFree(*line*)

classmethod fromFixed(*line*)

pyg4ometry.fluka.card.**_attempt_float_coercion**(*string*)

pyg4ometry.fluka.card.**freeFormatStringSplit**(*string*)

Method to split a string in FLUKA FREE format into its components.

pyg4ometry.fluka.card.**main**(*filein*)

pyg4ometry.fluka.directive

Module Contents

Classes

MatrixConvertibleMixin	
Transform	expansion, translation, rotoTranslation can be either a single
RotoTranslation	translation in mm, angles in degrees
RecursiveRotoTranslation	container for dealing with a recursively defined

Functions

rotoTranslationFromTBxyz(name, tbxyz[, flukaregistry])	tbxyz = trait bryan angles in radians
rotoTranslationFromTra2(name, tra2[, flukaregistry])	
_translationTo4DMatrix(translation)	
_expansionFactorTo4DMatrix(factor)	
_rightMultiplyMatrices(matrices)	

class pyg4ometry.fluka.directive.**MatrixConvertibleMixin**

toScaleMatrix()

toTranslationMatrix()

toRotationMatrix()

netTranslation()

netExpansion()

leftMultiplyVector(*vector*)

leftMultiplyRotation(*matrix*)

hash()

class pyg4ometry.fluka.directive.**Transform**(*, *expansion=None*, *translation=None*, *rotoTranslation=None*, *invertRotoTranslation=None*)

Bases: *MatrixConvertibleMixin*

expansion, translation, rotoTranslation can be either a single instance of RotoTranslation or a multiple instances of RotoTranslation and RecursiveRotoTranslation

_expansionsTo4DMatrices()

_translationsTo4DMatrices()

_rotoTranslationsTo4DMatrices()

to4DMatrix()

class pyg4ometry.fluka.directive.**RotoTranslation**(*name*, *axis=None*, *polar=0.0*, *azimuth=0.0*, *translation=None*, *transformationIndex=None*, *flukaregistry=None*)

Bases: *MatrixConvertibleMixin*

translation in mm, angles in degrees

__repr__()

Return repr(self).

to4DMatrix()

toCard()

flukaFreeString()

classmethod fromCard(*card*)

hasTranslation()

hasRotation()

isPureTranslation()

class pyg4ometry.fluka.directive.**RecursiveRotoTranslation**(*name*, *rotoTranslations*)

Bases: collections.abc.MutableSequence, *MatrixConvertibleMixin*

container for dealing with a recursively defined rototranslation. they must also refer to the same rototrans, i.e., have the same name. for a list of rototranslations supplied:

[a, b, c], the order of evaluation acting on a vector v is c*b*a*v. so teh first rototrans is applied first.. and so on.

property transformationIndex

__repr__()

Return repr(self).

__getitem__(*i*)

_raiseIfDifferentName(*name*)

__setitem__(*i*, *obj*)

__delitem__(*i*)

__len__()

insert(*i*, *obj*)

S.insert(index, value) – insert value before index

to4DMatrix()

flukaFreeString()

_transformationIndices()

areAllTheSameTransformationIndices()

transformationIndex()

pyg4ometry.fluka.directive.**rotoTranslationFromTBxyz**(*name*, *tbxyz*, *flukaregistry=None*)

tbxyz = trait bryan angles in radians

pyg4ometry.fluka.directive.**rotoTranslationFromTra2**(*name*, *tra2*, *flukaregistry=None*)

pyg4ometry.fluka.directive.**_translationTo4DMatrix**(*translation*)

pyg4ometry.fluka.directive.**_expansionFactorTo4DMatrix**(*factor*)

pyg4ometry.fluka.directive.**_rightMultiplyMatrices**(*matrices*)

pyg4ometry.fluka.elcfield

Module Contents

Classes

elcfield	

class pyg4ometry.fluka.elcfield.**elcfield**(*maxStepAngle=57*, *minBoundaryAccuracy=0.05*, *minStep=0.1*, *ex=0*, *ey=9*, *ez=0*, *dpdxFactor=None*)

pyg4ometry.fluka.extruder

Module Contents

Classes

Extruder	Base class for all solids

class pyg4ometry.fluka.extruder.**Extruder**(*name=''*, *length=1000*, *regions={}*, *registry=None*)

Bases: *pyg4ometry.geant4.solid.SolidBase*

Base class for all solids

addRegion(*name*)

addPointToRegion(*name*, *pntIndx*)

setRegionToOuterBoundary(*name*)

buildCgalPolygons()

buildGeant4Extrusions()

plot(*decompositions=False*)

mesh()

pyg4ometry.fluka.flair

Module Contents

Classes

Flair

Functions

_getFLUKATemplateFileName()

Attributes

_flukaTemplateFileName

pyg4ometry.fluka.flair.**_flukaTemplateFileName**

pyg4ometry.fluka.flair.**_getFLUKATemplateFileName**()

class pyg4ometry.fluka.flair.**Flair**(*flukaInputFileName='fluka.inp'*, *extent=None*)

write(*fileName*)

addMaterialColour(*materialName*, *color=(0, 0, 0, 0)*)

pyg4ometry.fluka.fluka_registry

Module Contents

Classes

FlukaRegistry	Object to store geometry for FLUKA input and output. All of the FLUKA classes can be used without storing them in the Registry. The registry is used to write the FLUKA output file.
RotoTranslationStore	only get by names.
FlukaBodyStore	A MutableMapping is a generic container for associating
BaseCacher	
Cacheable	
HalfSpaceCacher	
InfiniteCylinderCacher	
FlukaBodyStoreExact	

Attributes

logger

pyg4ometry.fluka.fluka_registry.**logger**

class pyg4ometry.fluka.fluka_registry.**FlukaRegistry**

Object to store geometry for FLUKA input and output. All of the FLUKA classes can be used without storing them in the Registry. The registry is used to write the FLUKA output file.

addBody(*body*)

makeBody(*clas*, **args*, ***kwargs*)

getDegenerateBody(*body*)

addRotoTranslation(*rototrans*)

addRegion(*region*, *addBodies=False*)

makeRegionsDNF()

addLattice(*lattice*)

getBody(*name*)

getBodyToRegionsMap()

printDefinitions()

regionAABBs(*write=None*)

latticeAABBs()

addMaterial(*material*, *recursive=False*)

getMaterial(*name*)

addMaterialAssignments(*mat*, **regions*, *elc=None*, *mgn=None*)

assignma(*material*, **regions*)

addCard(*card*)

addTitle(*title='FLUKA simulation'*)

addDefaults(*default='EM-CASCA'*)

addGlobal()

addBeam(*energy*, *particle='ELECTRON'*)

addBeamPos(*xpos=0*, *ypos=0*, *zpos=0*, *xdc=0*, *ydc=0*)

addLowMat(*flukaMat*, *lowENeutron1*, *lowENeutron2*, *lowENeutron3*)

addLowMatAllMaterials()
Add LOWMAT card to all materials

addLowPwxs(*what1=None*, *lowerMaterial=None*, *upperMaterial=None*)

addMgnCreat()

addMgnField()

addElcField()

addUsrBin(*mesh=0*, *particle='ENERGY'*, *lunOutput=-21*, *e1max=1*, *e2max=1*, *e3max=1*, *name='name'*, *e1min=-1*, *e2min=-1*, *e3min=-1*, *e1nbin=10*, *e2nbin=10*, *e3nbin=10*)

addRotprBin(*precision=0*, *rotDefi=0*, *printEventBin=0*, *lowerBin=None*, *upperBin=None*)

addUsrBdx(*binning*, *scoringDir*, *scoringType*, *type*, *reg1*, *reg2*, *name*, *area=1.0*, *lunOutput=-22*, *maxKE=None*, *minKE=None*, *nKEbin=None*, *maxSA=None*, *minSA=None*, *nSAbin=None*)

addUsricall()

addUsrocall()

addUsrDump(*mgdraw=100*, *lun=70*, *mgdrawOpt=-1*, *what4=0*, *sdum=None*)

addRandomiz(*seedLun=1*, *seed=54217137*)

addStart(*maxPrimHistories=1*, *timeTermSec=None*, *coreDump=None*, *eachHistoryOutput=None*)

addPhotonuc(*what1*, *mat_low*, *mat_high*, *mat_step*, *sdum=''*)

addMuphoton(*what1*, *mat_low*, *mat_high*, *mat_step*)

addPairbrem(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*)

addDeltaRay(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*, *dsum='NOPRINT'*)

addIonFluct(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*, *dsum='PRIM-ION'*)

printDumps(*detail=1*)

findLastBodyIndex()

Find last body index (if the numbering was performed by a geant4 -> fluka conversion

findLastRegionIndex()

Find last region index (if the numbering was performed by a geant4 -> fluka conversion

findLastMaterialIndex()

Find last material index (if the numbering was performed by a geant4 -> fluka conversion

findLastTransformationIndex()

checkBodyName(*bodyName*)

checkRegionName(*regionName*)

checkMaterialName(*materialName*)

addRegistry(*flukaRegistry*, *outerRegion=None*, *rotation=[0, 0, 0]*, *translation=[0, 0, 0]*, *removeRegions=[]*, *removeRegionDependents=False*)

class pyg4ometry.fluka.fluka_registry.**RotoTranslationStore**

Bases: collections.abc.MutableMapping

only get by names.

__getitem__(*name*)

__setitem__(*name*, *rtrans*)

addRotoTranslation(*rtrans*)

allTransformationIndices()

__delitem__(*key*)

__iter__()

__len__()

flukaFreeString()

class pyg4ometry.fluka.fluka_registry.**FlukaBodyStore**

Bases: collections.abc.MutableMapping

A MutableMapping is a generic container for associating key/value pairs.

This class provides concrete generic implementations of all methods except for __getitem__, __setitem__, __delitem__, __iter__, and __len__.

_bodyNames()

_bodies()

_getCacherFromBody(*body*)

make(*clas*, **args*, ***kwargs*)

getDegenerateBody(*body*)

addBody(*body*)

__setitem__(*key*, *value*)

__getitem__(*key*)

__delitem__(*key*)

__len__()

__contains__(*key*)

__iter__()

__repr__()
Return repr(self).

class pyg4ometry.fluka.fluka_registry.BaseCacher(*df*)

COLUMNS = ['name', 'body']

appendData(*variables*)

append(*body*)

setBody(*body*)

addBody(*body*)

remove(*key*)

make(*clas*, **args*, ***kwargs*)

getDegenerateBody(*body*)

__repr__()
Return repr(self).

class pyg4ometry.fluka.fluka_registry.Cacheable(*df*)
Bases: *BaseCacher*

getDegenerateBody(*body*)

getMask(*columns*, *values*, *predicates*)

class pyg4ometry.fluka.fluka_registry.HalfSpaceCacher(*df*)
Bases: *Cacheable*

COLUMNS = ['name', 'body', 'planeNormal', 'pointOnPlane']

append(*body*)

mask(*body*)

class pyg4ometry.fluka.fluka_registry.**InfiniteCylinderCacher**(*df*)

Bases: *Cacheable*

COLUMNS = ['name', 'body', 'direction', 'pointOnLine', 'radius']

append(*body*)

mask(*body*)

static _cylinderPoint(*body*)

class pyg4ometry.fluka.fluka_registry.**FlukaBodyStoreExact**

_bodyNames()

_bodies()

make(*cls*, **args*, ***kwargs*)

getDegenerateBody(*body*)

addBody(*body*)

keys()

values()

__setitem__(*key*, *value*)

__getitem__(*key*)

__delitem__(*key*)

__len__()

__contains__(*key*)

__iter__()

__repr__()

Return repr(self).

pyg4ometry.fluka.lattice

The Lattice class is defined in this module for handling LATTICEs in FLUKA geometries.

Module Contents

Classes

Lattice

class pyg4ometry.fluka.lattice.**Lattice**(*cellRegion*, *rotoTranslation*, *invertRotoTranslation=False*, *flukaregistry=None*)

flukaFreeString(*delim=', '*)

getTransform()

pyg4ometry.fluka.material

Module Contents

Classes

BuiltIn	
multiGroupNeutronCrossSections	
_MatProp	
Material	A FLUKA material consisting of a single element. This corresponds
Compound	A FLUKA compound material. This corresponds to the case in

Functions

predefinedMaterialNames()	
defineBuiltInFlukaMaterials([flukaregistry])	
_appendFractionPairs(card, fractions, fractionTypes)	
_parseFraction(what1, what2)	Returns the (name, fraction, fractionType). This is for handling
_formatFlukaMaterialPair(pair, namePrefix, fractionPrefix)	Names and fractions maybe stored as "negative" numbers in the
_grouper(n, iterable[, fillvalue])	grouper(3, 'ABCDEFG', 'x') --> ABC DEF Gxx

Attributes

_PREDEFINED_ELEMENTS	
_PREDEFINED_COMPOUNDS	

pyg4ometry.fluka.material._PREDEFINED_ELEMENTS = [('BLCKHOLE', 0, 0, 0), ('VACUUM', 0, 0, 0), ('HYDROGEN', 1.00794, 1.0, 8.37e-05), ('HELIUM',...

pyg4ometry.fluka.material._PREDEFINED_COMPOUNDS = [('WATER', 1.0), ('POLYSTYR', 1.06), ('PLASCINT', 1.032), ('PMMA', 1.19), ('BONECOMP', 1.85),...

pyg4ometry.fluka.material.**predefinedMaterialNames**()

class pyg4ometry.fluka.material.**BuiltIn**(*name*, *, *atomicNumber=None*, *atomicMass=None*, *density=None*, *flukaregistry=None*)

__repr__()

Return repr(self).

flukaFreeString(*delim=''*)

pyg4ometry.fluka.material.**defineBuiltInFlukaMaterials**(*flukaregistry=None*)

class pyg4ometry.fluka.material.**multiGroupNeutronCrossSections**(*fileName=None*)

_readFile()

findMaterial(*Z*, *A*, *T*, *selfShield=False*)

class pyg4ometry.fluka.material.**_MatProp**

isGas()

makeMatPropCard()

class pyg4ometry.fluka.material.**Material**(*name*, *atomicNumber*, *density*, *massNumber=None*, *atomicMass=None*, *pressure=None*, *flukaregistry=None*, *comment=''*)

Bases: *_MatProp*

A FLUKA material consisting of a single element. This corresponds to the case in FLUKA of a single MATERIAL card with no associated COMPOUND cards, as well as a possible MAT-PROP card (only if a pressure is provided, other options of MAT-PROP are unsupported).

Parameters

- **name** (*str*) – The name of the material
- **atomicNumber** (*int*) – the atomic number, Z, of the element.
- **density** (*float*) – the density in g/cm3 of the material.
- **massNumber** (*int, None*) – Optional mass number, will be inferred in FLUKA based on atomicNumber. Allows one to specify a specific isotope.
- **atomicMass** (*float*) – The mass of the atom in g/mole. Will be inferred in FLUKA based on atomicNumber.
- **pressure** (*float*) – Optional pressure if the material is a gas.
- **flukaregistry** (FlukaRegistry) – Optional FlukaRegistry instance the material is to be added to.

toCards()

flukaFreeString(*delim=', '*)

__repr__()

Return repr(self).

classmethod fromCard(*card*, *flukaregistry*)

rename(*newName*, *recursive=False*, *iIndex=0*)

class pyg4ometry.fluka.material.Compound(*name*, *density*, *fractions*, *fractionType*, *pressure=None*, *flukaregistry=None*, *comment=''*)

Bases: *_MatProp*

A FLUKA compound material. This corresponds to the case in FLUKA of a single MATERIAL card with one or more associated COMPOUND cards.

Parameters

- **name** (*str*) – The name of the compound.
- **density** (*float*) – The density of the compound in g/cm3
- **fractions** (*list*) – List of (Element, fraction) and (Compound, fraction) tuples corresponding to the fractional proportion of that material.
- **fractionType** (*str*) – The type of the fractions listed in the fractions parameter, either atomic, mass, or volume.
- **flukaregistry** (FlukaRegistry) – Optional FlukaRegistry instance the Compound is to be added to.

toCards()

flukaFreeString(*delim=', '*)

classmethod fromCards(*cards*, *flukareg*)

__repr__()

Return repr(self).

totalWeighting(*densityWeighted=False*)

rename(*newName*, *recursive=True*, *iIndex=0*)

pyg4ometry.fluka.material._appendFractionPairs(*card*, *fractions*, *fractionTypes*)

pyg4ometry.fluka.material._parseFraction(*what1*, *what2*)

Returns the (name, fraction, fractionType). This is for handling the different permutations found in the FLUKA manual COMPOUND entry.

pyg4ometry.fluka.material._formatFlukaMaterialPair(*pair*, *namePrefix*, *fractionPrefix*)

Names and fractions maybe stored as "negative" numbers in the FLUKA input, permutations between which mean different types of fractions.

pyg4ometry.fluka.material._grouper(*n*, *iterable*, *fillvalue=None*)

grouper(3, 'ABCDEFG', 'x') –> ABC DEF Gxx https://docs.python.org/3/library/itertools.html#recipes

pyg4ometry.fluka.mgnfield

Module Contents

Classes

mgnfield	
mgnfield_mgncreat	
mgncreat	
mgndata_array	

class pyg4ometry.fluka.mgnfield.**mgnfield**(*maxStepAngle=57*, *minBoundaryAccuracy=0.05*, *minStep=0.1*, *bx=0*, *by=9*, *bz=0*)

class pyg4ometry.fluka.mgnfield.**mgnfield_mgncreat**(*fieldStrength=1*, *rotDefi=None*, *regionLattice=0*, *lowerRegion=None*, *upperRegion=None*, *stepRegion=1*, *fieldName='field'*)

class pyg4ometry.fluka.mgnfield.**mgncreat**(*analyticalField=0*, *interpolatedField=0*, *interpolationRadius=0*, *xoffset=0*, *yoffset=0*, *xsym=0*, *ysym=0*, *zsym=0*, *fieldName=None*)

class pyg4ometry.fluka.mgnfield.**mgndata_array**(*mgndata*, *fieldName*)

pyg4ometry.fluka.preprocessor

Module Contents

Classes

_IfInfo	Tells us about current conditional and its state at the current line.
_Calc	A node visitor base class that walks the abstract syntax tree and calls a

Functions

_degree_input(f)	
_degree_output(f)	
preprocess(filein)	
_parse_preprocessor_define(directive, split_line, defines)	
_parse_preprocessor_conditional(directive, split_line, ...)	
_parse_preprocessor_include(directory, directive, ...)	

Attributes

_FLUKA_PREDEFINED_IDS	
_FLUKA_PREDEFINED_FUNCTIONS	

pyg4ometry.fluka.preprocessor.**_degree_input**(*f*)

pyg4ometry.fluka.preprocessor.**_degree_output**(*f*)

pyg4ometry.fluka.preprocessor.**_FLUKA_PREDEFINED_IDS**

pyg4ometry.fluka.preprocessor.**_FLUKA_PREDEFINED_FUNCTIONS**

pyg4ometry.fluka.preprocessor.**preprocess**(*filein*)

class pyg4ometry.fluka.preprocessor.**_IfInfo**(*any_branch_satisfied*, *read_until_next*)

Tells us about current conditional and its state at the current line.

- If the conditional at this level has been satisfied (e.g. if the IF has been satisfied then we don't read any subsequent ELIF or ELSE clauses, we skip until the next ENDIF).
- If we should read until the next conditional directive (e.g. if the IF or ELIF has been satisfied then we should read all lines following this directive until we reach the next conditional directive.

__repr__()

Return repr(self).

pyg4ometry.fluka.preprocessor.**_parse_preprocessor_define**(*directive*, *split_line*, *defines*)

pyg4ometry.fluka.preprocessor.**_parse_preprocessor_conditional**(*directive*, *split_line*, *defines*, *if_stack*)

pyg4ometry.fluka.preprocessor.**_parse_preprocessor_include**(*directory*, *directive*, *split_line*, *line_stack*)

class pyg4ometry.fluka.preprocessor._Calc(*definitions*)

Bases: ast.NodeVisitor

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

Per default the visitor functions for the nodes are `'visit_'` + class name of the node. So a *TryFinally* node visit function would be *visit_TryFinally*. This behavior can be changed by overriding the *visit* method. If no visitor function exists for a node (return value *None*) the *generic_visit* visitor is used instead.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversing. For this a special visitor exists (*NodeTransformer*) that allows modifications.

op_map

evaluate(*expression*)

visit_BinOp(*node*)

visit_UnaryOp(*node*)

visit_Constant(*node*)

visit_Name(*node*)

visit_Call(*node*)

visit_Expr(*node*)

pyg4ometry.fluka.reader

Module Contents

Classes

Reader	Class to read a FLUKA file.
RegionVisitor	A visitor class for accumulating the region definitions. The body
SensitiveErrorListener	ANTLR4 by default is very passive regarding parsing errors, it will

Functions

_getConstituentMaterialNamesFromCompound(con
_make_body(body_parts, expansion_stack, ...)
main(filein)

Attributes

_BODY_NAMES

pyg4ometry.fluka.reader.**_BODY_NAMES**

class pyg4ometry.fluka.reader.**Reader**(*filename*)

Class to read a FLUKA file.

getRegistry()

Get the fluka registry

_load()

Load the FLUKA input file

_findLines()

_parseBodies()

_parseRegions()

_parseCards()

_parseRotDefinis()

_parseGeometryDirective(*line_parts*, *expansion_stack*, *translation_stack*, *transform_stack*)

_parseMaterialAssignments()

_parseLattice()

_parseMaterials()

_splitMaterialCards()

Return a tuple as (Elements, Compounds, Matprop (not yet implemented)). Each is keyed with the name of the material, and the value corresponds to the card (Element) or cards (Compounds) that define the material.

pyg4ometry.fluka.reader.**_getConstituentMaterialNamesFromCompound**(*compoundCards*)

pyg4ometry.fluka.reader.**_make_body**(*body_parts*, *expansion_stack*, *translation_stack*, *transform_stack*, *flukareg*)

class pyg4ometry.fluka.reader.**RegionVisitor**(*flukaregistry*)

Bases: *pyg4ometry.fluka.RegionExpression.RegionParserVisitor*

A visitor class for accumulating the region definitions. The body instances are provided at instatiation, and then these are used when traversing the tree to build up a dictionary of region name and pyfluka.geometry.Region instances.

visitSimpleRegion(*ctx*)

visitComplexRegion(*ctx*)

visitUnaryAndBoolean(*ctx*)

visitUnaryExpression(*ctx*)

visitUnaryAndSubZone(*ctx*)

visitSingleUnion(*ctx*)

visitMultipleUnion(*ctx*)

visitMultipleUnion2(*ctx*)

visitSubZone(*ctx*)

visitZoneExpr(*ctx*)

visitZoneSubZone(*ctx*)

visitZoneBody(*ctx*)

class pyg4ometry.fluka.reader.**SensitiveErrorListener**

Bases: antlr4.error.ErrorListener.ErrorListener

ANTLR4 by default is very passive regarding parsing errors, it will just carry on parsing and potentially build a nonsense-tree. This is not ideal as pyfluka has a very convoluted syntax; we want to be very strict about what our parser can and can't do. For that reason this is a very sensitive error listener, throwing exceptions readily.

syntaxError(*recognizer*, *offendingSymbol*, *line*, *column*, *msg*, *e*)

pyg4ometry.fluka.reader.**main**(*filein*)

pyg4ometry.fluka.region

Module Contents

Classes

_Boolean	
Subtraction	
Intersection	
Union	
Zone	Represents a zone which consists of one or more body intersections
Region	Represents a region which consists of a region name, one or more

Functions

bracket_depth(zone)	
bracket_number(zone)	
zone_to_sympy(zone)	
region_to_sympy(region)	
sympy_to_zone(sympy_expr, freg)	
sympy_to_region(sympy_expr, freg[, regionName])	Convert sympy boolean to fluka region
simplify_region(region)	
_generate_name(typename, index, name, isZone, rootname)	Try to generate a sensible name for intermediate
_get_relative_rot_matrix(first, second)	
_get_relative_translation(first, second, aabb)	
_get_relative_rotation(first, second)	
_getRelativeTransform(first, second, aabb)	
_randomName()	Returns a random name that is syntactically correct for use in GDML.
_makeWorldLogicalVolume(reg)	
_getAxisAlignedBoundingBox(aabb, boolean)	aabb should really be a dictionary of

Attributes

logger

pyg4ometry.fluka.region.**logger**

pyg4ometry.fluka.region.**bracket_depth**(*zone*)

pyg4ometry.fluka.region.**bracket_number**(*zone*)

pyg4ometry.fluka.region.**zone_to_sympy**(*zone*)

pyg4ometry.fluka.region.**region_to_sympy**(*region*)

pyg4ometry.fluka.region.**sympy_to_zone**(*sympy_expr*, *freg*)

pyg4ometry.fluka.region.**sympy_to_region**(*sympy_expr*, *freg*, *regionName='name'*)

Convert sympy boolean to fluka region Must be the OR of multiple convex zones

pyg4ometry.fluka.region.**simplify_region**(*region*)

pyg4ometry.fluka.region.**_generate_name**(*typename*, *index*, *name*, *isZone*, *rootname*)

Try to generate a sensible name for intermediate Geant4 booleans which have no FLUKA analogue.

class pyg4ometry.fluka.region.**_Boolean**

generate_name(*index*, *rootname=None*)

class pyg4ometry.fluka.region.**Subtraction**(*body*)

Bases: *_Boolean*

class pyg4ometry.fluka.region.**Intersection**(*body*)

Bases: *_Boolean*

class pyg4ometry.fluka.region.**Union**(*body*)

Bases: *_Boolean*

class pyg4ometry.fluka.region.**Zone**(*name=None*)

Bases: *pyg4ometry.fluka.vis.ViewableMixin*

Represents a zone which consists of one or more body intersections and zero or more body subtractions. May also be used to represent subzones, which are zones nested within zones, for example the form +A -B -(+C -D). Once instantiated, intersections and subtractions can be added to this instance with the addIntersection and addSubtraction method.

Parameters

name (*string*) – Optional name for the zone.

addSubtraction(*body*)

Add a body or subzone as a subtraction to this Zone instance.

Parameters

body (*Body or Zone*) – Body or Zone instance.

addIntersection(*body*)

Add a body or subzone as an intersection to this Zone instance.

Parameters

body – Body or Zone instance.

Type

body: Body or *Zone*

convertToDNF(*fluka_registry*)

centre(*aabb=None*)

rotation()

tbxyz()

static _getSolidFromBoolean(*boolean*, *g4reg*, *aabb*)

mesh(*aabb=None*)

geant4Solid(*reg*, *aabb=None*)

Translate this zone to a geant4solid, adding the constituent primitive solids and any Booleans to the Geant4 registry. Returns the geant4 solid equivalent in geometry to this Zone.

Parameters

- **reg** (*AABB or dict*) – The Registry (geant4) instance to which the resulting Geant4 definitions should be added.
- **aabb** – Optional reference AABB or dictionary of body name to AABB instances with which the geant4 solid should be evaluated with respect to. This is the entry point to which solid minimisation can be performed.

_combine_booleans(*body0*, *start*, *collection*, *operation*, *reg*, *aabb*)

_geant4MultiUnionSubtraction(*body0*, *start*, *reg*, *aabb*)

dumps()

Returns a string of this Zone instance in the equivalent FLUKA syntax.

dumpsDebug()

Returns a string of this Zone instance in the equivalent FLUKA syntax with extra debug information

withLengthSafety(*bigger_flukareg*, *smaller_flukareg*, *shrink_intersections*)

allBodiesToRegistry(*flukaregistry*)

Add all the bodies that contitute this Zone to the provided FlukaRegistry instance.

Parameters

flukaregistry (*FlukaRegistry*) – FlukaRegistry instance to which constituent bodies will be added.

bodies()

Return the set of unique bodies that constitute this Zone.

removeBody(*name*)

Remove a body from this zone by name.

Parameters

name (*string*) – The name of the body to be removed.

makeUnique(*nameSuffix*, *flukaregistry*)

Get this zone with every constituent body recreated with a unique name by appending nameSuffix.

Parameters

- **nameSuffix** – The string to append to the names of the bodies.
- **flukaregistry** – the FlukaRegisytr instance to add the uniquely defined bodies to.

isNull(*aabb=None*)

toDNF(*name*)

isDNF()

leafCount()

class pyg4ometry.fluka.region.**Region**(*name*, *comment=''*)

Bases: *pyg4ometry.fluka.vis.ViewableMixin*

Represents a region which consists of a region name, one or more zones, and a single material. Metadata may be provided with the comment kwarg, which is used when writing to FLUKA to provide contextual information to the physicist.

Parameters

- **name** (*str*) – The name of this region.

- **material** (*str*) – The name of a material.
- **comment** – Optional descriptive comment.

addZone(*zone*)

Add a Zone instance to this region.

Parameters
zone (Zone) – The Zone instance to be added.

addIntersection(*zone*)

addSubtraction(*zone*)

extend(*region*)

removeNullZones()

convertToDNF(*fluka_registry*)

centre(*aabb=None*)

tbxyz()

rotation()

bodies()

Return the set of unique bodies that constitute this Zone.

mesh(*aabb=None*)

geant4Solid(*reg*, *aabb=None*)

Get the geant4Solid instance corresponding to this Region.

dumps()

flukaFreeString()

withLengthSafety(*bigger_flukareg*, *smaller_flukareg*)

allBodiesToRegistry(*registry*)

Add all the bodies that constitute this Region to the provided FlukaRegistry instance.

Parameters
flukaregistry (FlukaRegistry) – FlukaRegistry instance to which constituent bodies will be added.

zoneGraph(*zoneAABBs=None*, *aabb=None*)

_zoneGraphPycgal()

connectedZones(*zoneAABBs=None*, *aabb=None*)

zoneAABBs(*aabb=None*)

aabb(*aabb=None*)

removeBody(*name*)

Remove a body from this region by name.

Parameters
name (*string*) – The name of the body to be removed.

makeUnique(*nameSuffix*, *flukaregistry*)

Get this Region instance with every constituent body with a unique name by appending nameSuffix to each Body instance.

Parameters

- **nameSuffix** – string to append to each Body instance.
- **flukaregistry** – FlukaRegisty instance to add each newly-defined body to.

isNull(*aabb=None*)

toDNF(*name*)

isDNF()

filterNullZones(*aabb=None*)

leafCount()

simplify()

__repr__()

removeZones(*indices*)

Remove zones by index

pyg4ometry.fluka.region.**_get_relative_rot_matrix**(*first*, *second*)

pyg4ometry.fluka.region.**_get_relative_translation**(*first*, *second*, *aabb*)

pyg4ometry.fluka.region.**_get_relative_rotation**(*first*, *second*)

pyg4ometry.fluka.region.**_getRelativeTransform**(*first*, *second*, *aabb*)

pyg4ometry.fluka.region.**_randomName**()

Returns a random name that is syntactically correct for use in GDML.

pyg4ometry.fluka.region.**_makeWorldLogicalVolume**(*reg*)

pyg4ometry.fluka.region.**_getAxisAlignedBoundingBox**(*aabb*, *boolean*)

aabb should really be a dictionary of {bodyName: extentInstance}.

pyg4ometry.fluka.vector

Module Contents

Classes

Three
AABB

Functions

areAABBsOverlapping(first, second)	Check if two AABB instances are overlapping.
pointOnLineClosestToPoint(point, point_on_line, direction)	Line is defined in terms of two vectors: a point on the line and
pointOnPlaneClosestToPoint(planeNormal, planePoint, point)	Get point on plane which is closest to point not on the plane.
areParallelOrAntiParallel(v1, v2)	

class pyg4ometry.fluka.vector.**Three**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: numpy.ndarray

property x

property y

property z

parallel_to(*other*, *tolerance=1e-10*)

Check if instance is parallel to some other vector, v

unit()

Get this as a unit vector.

length()

vector length (l2 norm)

dot(*other*)

cross(*other*)

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__add__(*other*)

__radd__(*other*)

__sub__(*other*)

__rsub__(*other*)

__iadd__(*other*)

__isub__(*other*)

class pyg4ometry.fluka.vector.**AABB**(*lower*, *upper*)

__repr__()

Return repr(self).

__eq__(*other*)
Return self==value.

cornerDistance()

classmethod fromMesh(*csgmesh*)

intersects(*other*)

coplanarIntersects(*other*)

envelops(*other*)

intersect(*other*)

union(*other*)

isNull()

pyg4ometry.fluka.vector.**areAABBsOverlapping**(*first*, *second*)
Check if two AABB instances are overlapping.

pyg4ometry.fluka.vector.**pointOnLineClosestToPoint**(*point*, *point_on_line*, *direction*)
Line is defined in terms of two vectors: a point on the line and the direction of the line.

point is the point with which the distance to the line is to be minimised.

pyg4ometry.fluka.vector.**pointOnPlaneClosestToPoint**(*planeNormal*, *planePoint*, *point*)
Get point on plane which is closest to point not on the plane.

pyg4ometry.fluka.vector.**areParallelOrAntiParallel**(*v1*, *v2*)

pyg4ometry.fluka.vis

Module Contents

Classes

ViewableMixin

class pyg4ometry.fluka.vis.**ViewableMixin**

view(*aabb=None*)

Package Contents

Classes

RPP	Rectangular Parallelepiped
BOX	General Rectangular Parallelepiped
SPH	Sphere
RCC	Right Circular Cylinder
REC	Right Elliptical Cylinder
TRC	Truncated Right-angled Cone
ELL	Ellipsoid of Revolution
WED	Right Angle Wedge
RAW	Base class representing a body as defined in FLUKA
ARB	Arbitrary Convex Polyhedron
XYP	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)
XZP	Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)
YZP	Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)
PLA	Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.
XCC	Infinite Circular Cylinder parallel to the x-axis
YCC	Infinite Circular Cylinder parallel to the y-axis
ZCC	Infinite Circular Cylinder parallel to the z-axis
XEC	Infinite Elliptical Cylinder parallel to the x-axis
YEC	Infinite Elliptical Cylinder parallel to the y-axis
ZEC	Infinite Elliptical Cylinder parallel to the z-axis
QUA	Generic quadric
Reader	Class to read a FLUKA file.
Writer	
FlukaRegistry	Object to store geometry for FLUKA input and output. All of the FLUKA classes can be used without storing them in the Registry. The registry is used to write the FLUKA output file.
FlukaBodyStoreExact	
Three	
AABB	
Zone	Represents a zone which consists of one or more body intersections
Region	Represents a region which consists of a region name, one or more
Transform	expansion, translation, rotoTranslation can be either a single
RotoTranslation	translation in mm, angles in degrees
RecursiveRotoTranslation	container for dealing with a recursively defined

continues on next page

Table 2 – continued from previous page

Lattice	
Flair	
BuiltIn	
Material	A FLUKA material consisting of a single element. This corresponds
Compound	A FLUKA compound material. This corresponds to the case in
Card	Card class for representing a FLUKA input card. To construct
Extruder	Base class for all solids

Functions

infinity(inf)	Use this to temporarily modify INFINITY, with it resetting back
bracket_depth(zone)	
bracket_number(zone)	
zone_to_sympy(zone)	
region_to_sympy(region)	
sympy_to_region(sympy_expr, freg[, regionName])	Convert sympy boolean to fluka region

class pyg4ometry.fluka.**RPP**(*name*, *xmin*, *xmax*, *ymin*, *ymax*, *zmin*, *zmax*, *transform=None*, *flukaregistry=None*, *addRegistry=True*, *comment=''*)

Bases: BodyMixin

Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **xmin** (*float*) – lower x coordinate of RPP
- **xmax** (*float*) – upper x coordinate of RPP
- **ymin** (*float*) – lower y coordinate of RPP
- **ymax** (*float*) – upper y coordinate of RPP
- **zmin** (*float*) – lower z coordinate of RPP
- **zmax** (*float*) – upper z coordinate of RPP)

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**BOX**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

General Rectangular Parallelepiped

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the corners.
- **edge1** (*list*) – vector [x, y, z] denoting the first side of the box.
- **edge2** (*list*) – vector [x, y, z] denoting the second side of the box.
- **edge3** (*list*) – vector [x, y, z] denoting the second side of the box.

centre(*aabb=None*)

rotation()

lengths()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**SPH**(*name*, *point*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Sphere

Parameters

- **name** (*str*) – of body
- **point** (*list*) – position [x, y, z] of the centre of the sphere.
- **radius** (*float*) – radius of the sphere.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.RCC(*name*, *face*, *direction*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Circular Cylinder

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **edge1** (*list*) – vector [x, y, z] denoting the direction along the length of the cylinder.
- **edge2** (*float*) – radius of the cylinder face.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.REC(*name*, *face*, *direction*, *semiminor*, *semimajor*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Right Elliptical Cylinder

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the faces of the cylinder.
- **semiminor** (*list*) – vector [x, y, z] denoting the direction along the semiminor axis of the ellipse.
- **semimajor** (*list*) – vector [x, y, z] denoting the direction along the semimajor axis of the ellipse.

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**TRC**(*name*, *major_centre*, *direction*, *major_radius*, *minor_radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Truncated Right-angled Cone

Parameters

- **name** (*str*) – of body
- **major_centre** (*list*) – vector [x, y, z] position of the centre of the larger face.
- **direction** (*list*) – vector [x, y, z] pointing from the larger face to the smaller face.
- **major_radius** (*float*) – radius of the larger face.
- **minor_radius** (*float*) – radius of the smaller face.

centre(*aabb=None*)

rotation()

geant4Solid(*registry*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**ELL**(*name*, *focus1*, *focus2*, *length*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Ellipsoid of Revolution

Parameters

- **name** (*str*) – of body
- **focus1** (*list*) – position [x, y, z] denoting of one of the foci.
- **focus2** (*list*) – position [x, y, z] denoting the other focus.
- **length** (*float*) – length of the ellipse axis which the foci lie on.

centre(*aabb=None*)

rotation()

_linearEccentricity()

_semiminor()

geant4Solid(*greg*, *aabb=None*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**WED**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Right Angle Wedge

Parameters

- **name** (*str*) – of body
- **vertex** (*list*) – position [x, y, z] of one of the the rectangular corners.
- **edge1** (*list*) – vector [x, y, z] denoting height of the wedge.
- **edge2** (*list*) – vector [x, y, z] denoting width of the wedge.
- **edge3** (*list*) – vector [x, y, z] denoting length of the wedge.

class pyg4ometry.fluka.**RAW**(*name*, *vertex*, *edge1*, *edge2*, *edge3*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _WED_RAW

Base class representing a body as defined in FLUKA

__doc__

class pyg4ometry.fluka.**ARB**(*name*, *vertices*, *facenumbers*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin

Arbitrary Convex Polyhedron

Parameters

- **name** (*str*) – of body
- **vertices** (*list*) – Eight vertices which make up the polyhedron as [[x1, y1, z1], [x2, y2, z2], ...]. There must be eight even if only six or seven vertices are needed to make up the polydedron.
- **facenumbers** (*float*) – The faces of the polyhedron expressed as floats where each digit of the float refers to one of the vertices which makes up that face. Six must always be provided as [1234,8765, ...], even if only four or five faces are needed. Any unneeded faces must be set to 0 (no less than 4 sides). Note that the references to the vertices are not zero-counting. The order of the vertices denoted in the facenumbers must be either all clockwise or all anticlockwise, which if not obeyed will result in erroneous output without warning.

centre(*aabb=None*)

rotation()

_faceNumbersToZeroCountingIndices()

_extent()

geant4Solid(*greg*, *aabb=None*)

_toTesselatedSolid(*verticesAndPolygons*, *greg*, *addRegistry*)

_getVerticesAndPolygons()

_toVerticesAndPolygons(*reverse*)

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

class pyg4ometry.fluka.**XYP**(*name*, *z*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis)

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the x-y plane on the z-axis. All points less than z are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.**XZP**(*name*, *y*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the y-axis)

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the x-y plane on the y-axis. All points less than y are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.**YZP**(*name*, *x*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the x-axis)

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the x-y plane on the x-axis. All points less than x are considered to be part of this body.

__repr__()

_withLengthSafety(*safety*, *reg*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.**PLA**(*name*, *normal*, *point*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _HalfSpaceMixin

Infinite half-space delimited by the x-y plane (perpendicular to the z-axis) Generic infinite half-space.

Parameters

- **name** (*str*) – of body
- **normal** (*list*) – position of a point on the plane
- **normal** – vector perpendicular to the face of the plane, pointing away from the contents of the half space.

rotation()

__repr__()

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

toPlane()

class pyg4ometry.fluka.**XCC**(*name*, *y*, *z*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()

Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.YCC(*name*, *z*, *x*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()

Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.ZCC(*name*, *x*, *y*, *radius*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: _InfiniteCylinderMixin, _ShiftableCylinderMixin

Infinite Circular Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the
- **y** (*float*) – position of the centre on the
- **radius** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

__repr__()

Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

point()

direction()

class pyg4ometry.fluka.XEC(*name*, *y*, *z*, *ysemi*, *zsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the x-axis

Parameters

- **name** (*str*) – of body
- **y** (*float*) – position of the centre on the
- **z** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()

Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.YEC(*name*, *z*, *x*, *zsemi*, *xsemi*, *transform=None*, *flukaregistry=None*, *comment=''*)

Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the y-axis

Parameters

- **name** (*str*) – of body
- **z** (*float*) – position of the centre on the
- **x** (*float*) – position of the centre on the
- **zsemi** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.ZEC(*name*, *x*, *y*, *xsemi*, *ysemi*, *transform=None*, *flukaregistry=None*, *comment=''*)
Bases: BodyMixin, _ShiftableCylinderMixin

Infinite Elliptical Cylinder parallel to the z-axis

Parameters

- **name** (*str*) – of body
- **x** (*float*) – position of the centre on the
- **y** (*float*) – position of the centre on the
- **xsemi** (*float*) – position of the centre on the
- **ysemi** (*float*) – position of the centre on the

centre(*aabb=None*)

rotation()

geant4Solid(*reg*, *aabb=None*)

__repr__()
Return repr(self).

_withLengthSafety(*safety*, *reg=None*)

flukaFreeString()

hash()

class pyg4ometry.fluka.QUA(*name*, *cxx*, *cyy*, *czz*, *cxy*, *cxz*, *cyz*, *cx*, *cy*, *cz*, *c*, *transform=None*, *flukaregistry=None*, *comment=''*, ***kwargs*)
Bases: BodyMixin

Generic quadric

Parameters

- **name** (*str*) – of body
- **cxx** (*float*) – x^2 coefficient
- **cyy** (*float*) – y^2 coefficient
- **czz** (*float*) – z^2 coefficient
- **cxy** (*float*) – xy coefficient
- **cxz** (*float*) – xz coefficient
- **cyz** (*float*) – yz coefficient
- **cx** (*float*) – x coefficient
- **cy** (*float*) – y coefficient

- **cz** (*float*) – z coefficient
- **c** (*constant*) – constant

centre(*aabb=None*)

rotation()

coefficientsMatrix()

static _quadricMatrixToCoefficients(*matrix*)

mesh(*lower*, *upper*, *capping=True*)

geant4Solid(*reg*, *aabb=None*)

_withLengthSafety(*safety*, *reg=None*)

__repr__()

flukaFreeString()

hash()

pyg4ometry.fluka.**infinity**(*inf*)

Use this to temporarily modify INFINITY, with it resetting back to the default once the block has exited. INFINITY is used throughout the bodies to approximate the infinite size of infinity (elliptical) cylinders, half spaces, and quadrics.

Parameters
inf – the value to temporarily set INFINITY to.

class pyg4ometry.fluka.**Reader**(*filename*)

Class to read a FLUKA file.

getRegistry()

Get the fluka registry

_load()

Load the FLUKA input file

_findLines()

_parseBodies()

_parseRegions()

_parseCards()

_parseRotDefinis()

_parseGeometryDirective(*line_parts*, *expansion_stack*, *translation_stack*, *transform_stack*)

_parseMaterialAssignments()

_parseLattice()

_parseMaterials()

_splitMaterialCards()

Return a tuple as (Elements, Compounds, Matprop (not yet implemented)). Each is keyed with the name of the material, and the value corresponds to the card (Element) or cards (Compounds) that define the material.

class pyg4ometry.fluka.**Writer**

Class to write FLUKA input files from a fluka registry object.

```
>>> f = Writer()
>>> f.addDetectro(flukaRegObject)
>>> f.write("model.inp")
```

_flukaFFString =
'*...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+...'

addDetector(*flukaRegistry*)

Set the fluka registry and therefore the model for this writer instance.

write(*fileName*)

Write the output to a given filename. e.g. "model.inp".

class pyg4ometry.fluka.**FlukaRegistry**

Object to store geometry for FLUKA input and output. All of the FLUKA classes can be used without storing them in the Registry. The registry is used to write the FLUKA output file.

addBody(*body*)

makeBody(*clas*, **args*, ***kwargs*)

getDegenerateBody(*body*)

addRotoTranslation(*rototrans*)

addRegion(*region*, *addBodies=False*)

makeRegionsDNF()

addLattice(*lattice*)

getBody(*name*)

getBodyToRegionsMap()

printDefinitions()

regionAABBs(*write=None*)

latticeAABBs()

addMaterial(*material*, *recursive=False*)

getMaterial(*name*)

addMaterialAssignments(*mat*, **regions*, *elc=None*, *mgn=None*)

assignma(*material*, **regions*)

addCard(*card*)

addTitle(*title='FLUKA simulation'*)

addDefaults(*default='EM-CASCA'*)

addGlobal()

addBeam(*energy*, *particle='ELECTRON'*)

addBeamPos(*xpos=0*, *ypos=0*, *zpos=0*, *xdc=0*, *ydc=0*)

addLowMat(*flukaMat*, *lowENeutron1*, *lowENeutron2*, *lowENeutron3*)

addLowMatAllMaterials()
Add LOWMAT card to all materials

addLowPwxs(*what1=None*, *lowerMaterial=None*, *upperMaterial=None*)

addMgnCreat()

addMgnField()

addElcField()

addUsrBin(*mesh=0*, *particle='ENERGY'*, *lunOutput=-21*, *e1max=1*, *e2max=1*, *e3max=1*, *name='name'*, *e1min=-1*, *e2min=-1*, *e3min=-1*, *e1nbin=10*, *e2nbin=10*, *e3nbin=10*)

addRotprBin(*precision=0*, *rotDefi=0*, *printEventBin=0*, *lowerBin=None*, *upperBin=None*)

addUsrBdx(*binning*, *scoringDir*, *scoringType*, *type*, *reg1*, *reg2*, *name*, *area=1.0*, *lunOutput=-22*, *maxKE=None*, *minKE=None*, *nKEbin=None*, *maxSA=None*, *minSA=None*, *nSAbin=None*)

addUsricall()

addUsrocall()

addUsrDump(*mgdraw=100*, *lun=70*, *mgdrawOpt=-1*, *what4=0*, *sdum=None*)

addRandomiz(*seedLun=1*, *seed=54217137*)

addStart(*maxPrimHistories=1*, *timeTermSec=None*, *coreDump=None*, *eachHistoryOutput=None*)

addPhotonuc(*what1*, *mat_low*, *mat_high*, *mat_step*, *sdum=''*)

addMuphoton(*what1*, *mat_low*, *mat_high*, *mat_step*)

addPairbrem(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*)

addDeltaRay(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*, *dsum='NOPRINT'*)

addIonFluct(*what1*, *what2*, *what3*, *mat_low*, *mat_high*, *mat_step*, *dsum='PRIM-ION'*)

printDumps(*detail=1*)

findLastBodyIndex()
Find last body index (if the numbering was performed by a geant4 -> fluka conversion

findLastRegionIndex()
Find last region index (if the numbering was performed by a geant4 -> fluka conversion

findLastMaterialIndex()
Find last material index (if the numbering was performed by a geant4 -> fluka conversion

findLastTransformationIndex()

checkBodyName(*bodyName*)

checkRegionName(*regionName*)

checkMaterialName(*materialName*)

addRegistry(*flukaRegistry*, *outerRegion=None*, *rotation=[0, 0, 0]*, *translation=[0, 0, 0]*, *removeRegions=[]*, *removeRegionDependents=False*)

class pyg4ometry.fluka.FlukaBodyStoreExact

_bodyNames()

_bodies()

make(*cls*, **args*, ***kwargs*)

getDegenerateBody(*body*)

addBody(*body*)

keys()

values()

__setitem__(*key*, *value*)

__getitem__(*key*)

__delitem__(*key*)

__len__()

__contains__(*key*)

__iter__()

__repr__()
Return repr(self).

class pyg4ometry.fluka.Three(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)
Bases: numpy.ndarray

property x

property y

property z

parallel_to(*other*, *tolerance=1e-10*)
Check if instance is parallel to some other vector, v

unit()
Get this as a unit vector.

length()
vector length (l2 norm)

dot(*other*)

cross(*other*)

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

__add__(*other*)

__radd__(*other*)

__sub__(*other*)

__rsub__(*other*)

__iadd__(*other*)

__isub__(*other*)

class pyg4ometry.fluka.**AABB**(*lower*, *upper*)

__repr__()
Return repr(self).

__eq__(*other*)
Return self==value.

cornerDistance()

classmethod fromMesh(*csgmesh*)

intersects(*other*)

coplanarIntersects(*other*)

envelops(*other*)

intersect(*other*)

union(*other*)

isNull()

class pyg4ometry.fluka.**Zone**(*name=None*)
Bases: *pyg4ometry.fluka.vis.ViewableMixin*

Represents a zone which consists of one or more body intersections and zero or more body subtractions. May also be used to represent subzones, which are zones nested within zones, for example the form +A -B -(+C -D). Once instantiated, intersections and subtractions can be added to this instance with the addIntersection and addSubtraction method.

Parameters
name (*string*) – Optional name for the zone.

addSubtraction(*body*)

Add a body or subzone as a subtraction to this Zone instance.

Parameters
body (*Body or Zone*) – Body or Zone instance.

addIntersection(*body*)

Add a body or subzone as an intersection to this Zone instance.

Parameters
body – Body or Zone instance.

Type
body: Body or *Zone*

convertToDNF(*fluka_registry*)

centre(*aabb=None*)

rotation()

tbxyz()

static _getSolidFromBoolean(*boolean*, *g4reg*, *aabb*)

mesh(*aabb=None*)

geant4Solid(*reg*, *aabb=None*)

Translate this zone to a geant4solid, adding the constituent primitive solids and any Booleans to the Geant4 registry. Returns the geant4 solid equivalent in geometry to this Zone.

Parameters

- **reg** (*AABB or dict*) – The Registry (geant4) instance to which the resulting Geant4 definitions should be added.
- **aabb** – Optional reference AABB or dictionary of body name to AABB instances with which the geant4 solid should be evaluated with respect to. This is the entry point to which solid minimisation can be performed.

_combine_booleans(*body0*, *start*, *collection*, *operation*, *reg*, *aabb*)

_geant4MultiUnionSubtraction(*body0*, *start*, *reg*, *aabb*)

dumps()

Returns a string of this Zone instance in the equivalent FLUKA syntax.

dumpsDebug()

Returns a string of this Zone instance in the equivalent FLUKA syntax with extra debug information

withLengthSafety(*bigger_flukareg*, *smaller_flukareg*, *shrink_intersections*)

allBodiesToRegistry(*flukaregistry*)

Add all the bodies that contitute this Zone to the provided FlukaRegistry instance.

Parameters
flukaregistry (*FlukaRegistry*) – FlukaRegistry instance to which constituent bodies will be added.

bodies()

Return the set of unique bodies that constitute this Zone.

removeBody(*name*)

Remove a body from this zone by name.

Parameters
name (*string*) – The name of the body to be removed.

makeUnique(*nameSuffix*, *flukaregistry*)

Get this zone with every constituent body recreated with a unique name by appending nameSuffix.

Parameters

- **nameSuffix** – The string to append to the names of the bodies.
- **flukaregistry** – the FlukaRegisytr instance to add the uniquely defined bodies to.

isNull(*aabb=None*)

toDNF(*name*)

isDNF()

leafCount()

class pyg4ometry.fluka.**Region**(*name*, *comment=''*)

Bases: *pyg4ometry.fluka.vis.ViewableMixin*

Represents a region which consists of a region name, one or more zones, and a single material. Metadata may be provided with the comment kwarg, which is used when writing to FLUKA to provide contextual information to the physicist.

Parameters

- **name** (*str*) – The name of this region.
- **material** (*str*) – The name of a material.
- **comment** – Optional descriptive comment.

addZone(*zone*)

Add a Zone instance to this region.

Parameters
zone (Zone) – The Zone instance to be added.

addIntersection(*zone*)

addSubtraction(*zone*)

extend(*region*)

removeNullZones()

convertToDNF(*fluka_registry*)

centre(*aabb=None*)

tbxyz()

rotation()

bodies()

Return the set of unique bodies that constitute this Zone.

mesh(*aabb=None*)

geant4Solid(*reg*, *aabb=None*)

Get the geant4Solid instance corresponding to this Region.

dumps()

flukaFreeString()

withLengthSafety(*bigger_flukareg*, *smaller_flukareg*)

allBodiesToRegistry(*registry*)

Add all the bodies that constitute this Region to the provided FlukaRegistry instance.

Parameters
flukaregistry (FlukaRegistry) – FlukaRegistry instance to which constituent bodies will be added.

zoneGraph(*zoneAABBs=None*, *aabb=None*)

_zoneGraphPycgal()

connectedZones(*zoneAABBs=None*, *aabb=None*)

zoneAABBs(*aabb=None*)

aabb(*aabb=None*)

removeBody(*name*)

Remove a body from this region by name.

Parameters
name (*string*) – The name of the body to be removed.

makeUnique(*nameSuffix*, *flukaregistry*)

Get this Region instance with every constituent body with a unique name by appending nameSuffix to each Body instance.

Parameters

- **nameSuffix** – string to append to each Body instance.
- **flukaregistry** – FlukaRegisty instance to add each newly-defined body to.

isNull(*aabb=None*)

toDNF(*name*)

isDNF()

filterNullZones(*aabb=None*)

leafCount()

simplify()

__repr__()

removeZones(*indices*)

Remove zones by index

pyg4ometry.fluka.**bracket_depth**(*zone*)

pyg4ometry.fluka.**bracket_number**(*zone*)

pyg4ometry.fluka.**zone_to_sympy**(*zone*)

pyg4ometry.fluka.**region_to_sympy**(*region*)

pyg4ometry.fluka.**sympy_to_region**(*sympy_expr*, *freg*, *regionName='name'*)

Convert sympy boolean to fluka region Must be the OR of multiple convex zones

class pyg4ometry.fluka.**Transform**(***, *expansion=None*, *translation=None*, *rotoTranslation=None*, *invertRotoTranslation=None*)

Bases: MatrixConvertibleMixin

expansion, translation, rotoTranslation can be either a single instance of RotoTranslation or a multiple instances of RotoTranslation and RecursiveRotoTranslation

_expansionsTo4DMatrices()

_translationsTo4DMatrices()

_rotoTranslationsTo4DMatrices()

to4DMatrix()

class pyg4ometry.fluka.**RotoTranslation**(*name*, *axis=None*, *polar=0.0*, *azimuth=0.0*, *translation=None*, *transformationIndex=None*, *flukaregistry=None*)

Bases: MatrixConvertibleMixin

translation in mm, angles in degrees

__repr__()

Return repr(self).

to4DMatrix()

toCard()

flukaFreeString()

classmethod fromCard(*card*)

hasTranslation()

hasRotation()

isPureTranslation()

class pyg4ometry.fluka.**RecursiveRotoTranslation**(*name*, *rotoTranslations*)

Bases: collections.abc.MutableSequence, MatrixConvertibleMixin

container for dealing with a recursively defined rototranslation. they must also refer to the same rototrans, i.e., have the same name. for a list of rototranslations supplied:

[a, b, c], the order of evaluation acting on a vector v is c*b*a*v. so teh first rototrans is applied first.. and so on.

property transformationIndex

__repr__()

Return repr(self).

__getitem__(*i*)

_raiseIfDifferentName(*name*)

__setitem__(*i*, *obj*)

__delitem__(*i*)

__len__()

insert(*i*, *obj*)

S.insert(index, value) – insert value before index

to4DMatrix()

flukaFreeString()

_transformationIndices()

areAllTheSameTransformationIndices()

transformationIndex()

class pyg4ometry.fluka.Lattice(*cellRegion*, *rotoTranslation*, *invertRotoTranslation=False*, *flukaregistry=None*)

flukaFreeString(*delim=', '*)

getTransform()

class pyg4ometry.fluka.Flair(*flukaInputFileName='fluka.inp'*, *extent=None*)

write(*fileName*)

addMaterialColour(*materialName*, *color=(0, 0, 0, 0)*)

class pyg4ometry.fluka.BuiltIn(*name*, *, *atomicNumber=None*, *atomicMass=None*, *density=None*, *flukaregistry=None*)

__repr__()

Return repr(self).

flukaFreeString(*delim=''*)

class pyg4ometry.fluka.Material(*name*, *atomicNumber*, *density*, *massNumber=None*, *atomicMass=None*, *pressure=None*, *flukaregistry=None*, *comment=''*)

Bases: _MatProp

A FLUKA material consisting of a single element. This corresponds to the case in FLUKA of a single MATERIAL card with no associated COMPOUND cards, as well as a possible MAT-PROP card (only if a pressure is provided, other options of MAT-PROP are unsupported).

Parameters

- **name** (*str*) – The name of the material
- **atomicNumber** (*int*) – the atomic number, Z, of the element.
- **density** (*float*) – the density in g/cm3 of the material.
- **massNumber** (*int, None*) – Optional mass number, will be inferred in FLUKA based on atomicNumber. Allows one to specify a specific isotope.

- **atomicMass** (*float*) – The mass of the atom in g/mole. Will be inferred in FLUKA based on atomicNumber.
- **pressure** (*float*) – Optional pressure if the material is a gas.
- **flukaregistry** (FlukaRegistry) – Optional FlukaRegistry instance the material is to be added to.

toCards()

flukaFreeString(*delim=', '*)

__repr__()

Return repr(self).

classmethod fromCard(*card*, *flukaregistry*)

rename(*newName*, *recursive=False*, *iIndex=0*)

class pyg4ometry.fluka.**Compound**(*name*, *density*, *fractions*, *fractionType*, *pressure=None*, *flukaregistry=None*, *comment=''*)

Bases: _MatProp

A FLUKA compound material. This corresponds to the case in FLUKA of a single MATERIAL card with one or more associated COMPOUND cards.

Parameters

- **name** (*str*) – The name of the compound.
- **density** (*float*) – The density of the compound in g/cm3
- **fractions** (*list*) – List of (Element, fraction) and (Compound, fraction) tuples corresponding to the fractional proportion of that material.
- **fractionType** (*str*) – The type of the fractions listed in the fractions parameter, either atomic, mass, or volume.
- **flukaregistry** (FlukaRegistry) – Optional FlukaRegistry instance the Compound is to be added to.

toCards()

flukaFreeString(*delim=', '*)

classmethod fromCards(*cards*, *flukareg*)

__repr__()

Return repr(self).

totalWeighting(*densityWeighted=False*)

rename(*newName*, *recursive=True*, *iIndex=0*)

class pyg4ometry.fluka.**Card**(*keyword*, *what1=None*, *what2=None*, *what3=None*, *what4=None*, *what5=None*, *what6=None*, *sdum=None*)

Card class for representing a FLUKA input card. To construct instances from a of FLUKA input, use the fromFree or fromFixed class method for FREE and FIXED format, respectively.

__repr__()

Return repr(self).

toList()

toFreeString(*delim=', '*)

toFixedString()

nonesToZero()

Return a class instance with same contents as this instance, but with all entries of None set to 0.0 instead.

classmethod fromFree(*line*)

classmethod fromFixed(*line*)

class pyg4ometry.fluka.**Extruder**(*name=''*, *length=1000*, *regions={}*, *registry=None*)

Bases: *pyg4ometry.geant4.solid.SolidBase*

Base class for all solids

addRegion(*name*)

addPointToRegion(*name*, *pntIndx*)

setRegionToOuterBoundary(*name*)

buildCgalPolygons()

buildGeant4Extrusions()

plot(*decompositions=False*)

mesh()

pyg4ometry.freecad

Submodules

pyg4ometry.freecad.Reader

Module Contents

Classes

Reader

Functions

MeshToFacetList(mesh)	
WriteSMeshFile(mesh, filename)	
FacetListAxisAlignedExtent(facetList)	
PartFeatureGlobalPlacement(obj, placement)	
MeshAnalysis(m)	
MeshCleaning(m)	

class pyg4ometry.freecad.Reader.**Reader**(*fileName*, *registryOn=True*, *fileNameAux=None*)

load(*fileName*)

simplifyModel(*volumeCut=500000.0*)

relabelModel()

loadAuxilaryData(*fileName*, *colorByMaterial=True*)

convertStructure()

Convert file with structure

setLogicalVolumeMaterial(*logicalVolumeName*, *material='G4_Galactic'*)

convertFlat(*meshDeviation=0.05*, *centreName=''*, *globalOffset=_fc.Vector()*, *globalRotation=_fc.Rotation()*, *extentScale=1.0*, *daughterMaterial='G4_Galactic'*, *storePartCentrePos=False*, *meshShrinkFactor=1e-06*)

Convert file without structure

getRegistry()

printPartFeatures(*fileName=None*, *randomColors=False*)

Print to screen or write to file Part::Features with color and material

printStructure()

recursePrintObjectTree(*obj*)

recurseObjectTree(*obj*)

pyg4ometry.freecad.Reader.**MeshToFacetList**(*mesh*)

pyg4ometry.freecad.Reader.**WriteSMeshFile**(*mesh*, *filename*)

pyg4ometry.freecad.Reader.**FacetListAxisAlignedExtent**(*facetList*)

pyg4ometry.freecad.Reader.**PartFeatureGlobalPlacement**(*obj*, *placement*)

pyg4ometry.freecad.Reader.**MeshAnalysis**(*m*)

pyg4ometry.freecad.Reader.**MeshCleaning**(*m*)

Package Contents

pyg4ometry.freecad.**useFreeCAD** = **True**

pyg4ometry.freecad.**useFreeCAD** = **False**

pyg4ometry.gdml

Subpackages

pyg4ometry.gdml.GdmlExpression

Submodules

pyg4ometry.gdml.GdmlExpression.GdmlExpressionEval

Module Contents

Classes

GdmlExpressionEvalVisitor
ExpressionParser

class pyg4ometry.gdml.GdmlExpression.GdmlExpressionEval.**GdmlExpressionEvalVisitor**

Bases: *pyg4ometry.gdml.GdmlExpression.GdmlExpressionVisitor.GdmlExpressionVisitor*

visitVariable(*ctx*)

visitPrintExpr(*ctx*)

visitScientific(*ctx*)

visitMultiplyingExpression(*ctx*)

visitExpression(*ctx*)

visitPowExpression(*ctx*)

visitMinExpression(*ctx*)

visitMaxExpression(*ctx*)

visitMatrixElement(*ctx*)

visitParens(*ctx*)

visitSignedAtom(*ctx*)

visitAtom(*ctx*)

visitFunc(*ctx*)

visitFuncname(*ctx*)

visitConstant(*ctx*)

class pyg4ometry.gdml.GdmlExpression.GdmlExpressionEval.**ExpressionParser**

parse(*expression*)

evaluate(*parse_tree*, *define_dict={}*)

get_variables(*parse_tree*)

pyg4ometry.gdml.GdmlExpression.GdmlExpressionLexer

Module Contents

Classes

GdmlExpressionLexer

Functions

serializedATN()

pyg4ometry.gdml.GdmlExpression.GdmlExpressionLexer.**serializedATN**()

class pyg4ometry.gdml.GdmlExpression.GdmlExpressionLexer.**GdmlExpressionLexer**(*input=None*, *output=sys.stdout*)

Bases: Lexer

Parameters
output (*TextIO*) –

atn

decisionsToDFA

COS = 1

SIN = 2

TAN = 3

ACOS = 4

ASIN = 5

```
    ATAN = 6

    LN = 7

    LOG = 8

    SQRT = 9

    EXP = 10

    POWER = 11

    ABS = 12

    LPAREN = 13

    RPAREN = 14

    LBRACKET = 15

    RBRACKET = 16

    PLUS = 17

    MINUS = 18

    TIMES = 19

    DIV = 20

    GT = 21

    LT = 22

    EQ = 23

    COMMA = 24

    POINT = 25

    POW = 26

    PI = 27

    EULER = 28

    I = 29

    MIN = 30

    MAX = 31

    VARIABLE = 32

    SCIENTIFIC_NUMBER = 33

    WS = 34

    channelNames = ['DEFAULT_TOKEN_CHANNEL', 'HIDDEN']

    modeNames = ['DEFAULT_MODE']
```

```
literalNames = ['<INVALID>', "'cos'", "'sin'", "'tan'", "'acos'", "'asin'",
"'atan'", "'log'", "'log10'",...

symbolicNames = ['<INVALID>', 'COS', 'SIN', 'TAN', 'ACOS', 'ASIN', 'ATAN', 'LN',
'LOG', 'SQRT', 'EXP', 'POWER',...

ruleNames = ['COS', 'SIN', 'TAN', 'ACOS', 'ASIN', 'ATAN', 'LN', 'LOG', 'SQRT',
'EXP', 'POWER', 'ABS',...

grammarFileName = 'GdmlExpression.g4'
```

pyg4ometry.gdml.GdmlExpression.GdmlExpressionParser

Module Contents

Classes

GdmlExpressionParser

Functions

serializedATN()

pyg4ometry.gdml.GdmlExpression.GdmlExpressionParser.**serializedATN**()

class pyg4ometry.gdml.GdmlExpression.GdmlExpressionParser.**GdmlExpressionParser**(*input*, *output=sys.stdout*)

Bases: Parser

Parameters

- **input** (*TokenStream*) –
- **output** (*TextIO*) –

class EquationContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

expression(*i=None*)

Parameters

i (*Optional[int]*) –

relop()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ExpressionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

multiplyingExpression(*i=None*)

Parameters
i (*Optional[int]*) –

operatorAddSub(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class MultiplyingExpressionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

powExpression(*i=None*)

Parameters
i (*Optional[int]*) –

operatorMulDiv(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class OperatorAddSubContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

PLUS()

MINUS()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class OperatorMulDivContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

TIMES()

DIV()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class PowExpressionContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

signedAtom(*i=None*)

Parameters
i (*Optional[int]*) –

POW(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class SignedAtomContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

PLUS()

signedAtom()

MINUS()

func()

atom()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class AtomContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

scientific()

matrixElement()

variable()

constant()

LPAREN()

expression()

RPAREN()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ScientificContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

SCIENTIFIC_NUMBER()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class MatrixElementContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

variable()

LBRACKET()

expression(*i=None*)

Parameters
i (*Optional[int]*) –

RBRACKET()

COMMA(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class ConstantContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

PI()

EULER()

I()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class VariableContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

VARIABLE()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class FuncContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

funcname()

LPAREN()

expression(*i=None*)

Parameters
i (*Optional[int]*) –

RPAREN()

COMMA(*i=None*)

Parameters
i (*Optional[int]*) –

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class FuncnameContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

COS()

TAN()

SIN()

ACOS()

ATAN()

ASIN()

LOG()

LN()

EXP()

SQRT()

POWER()

ABS()

MIN()

MAX()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

class RelopContext(*parser*, *parent=None*, *invokingState=-1*)

Bases: ParserRuleContext

Parameters

- **parent** (*ParserRuleContext*) –
- **invokingState** (*int*) –

__slots__ = 'parser'

EQ()

GT()

LT()

getRuleIndex()

accept(*visitor*)

Parameters
visitor (*ParseTreeVisitor*) –

grammarFileName = 'GdmlExpression.g4'

atn

decisionsToDFA

sharedContextCache

literalNames = ['<INVALID>', "'cos'", "'sin'", "'tan'", "'acos'", "'asin'", "'atan'", "'log'", "'log10'",...

symbolicNames = ['<INVALID>', 'COS', 'SIN', 'TAN', 'ACOS', 'ASIN', 'ATAN', 'LN', 'LOG', 'SQRT', 'EXP', 'POWER',...

RULE_equation = 0

RULE_expression = 1

RULE_multiplyingExpression = 2

RULE_operatorAddSub = 3

RULE_operatorMulDiv = 4

RULE_powExpression = 5

RULE_signedAtom = 6

RULE_atom = 7

RULE_scientific = 8

RULE_matrixElement = 9

RULE_constant = 10

RULE_variable = 11

RULE_func = 12

RULE_funcname = 13

RULE_relop = 14

ruleNames = ['equation', 'expression', 'multiplyingExpression', 'operatorAddSub', 'operatorMulDiv',...

EOF

COS = 1

SIN = 2

TAN = 3

```
ACOS = 4

ASIN = 5

ATAN = 6

LN = 7

LOG = 8

SQRT = 9

EXP = 10

POWER = 11

ABS = 12

LPAREN = 13

RPAREN = 14

LBRACKET = 15

RBRACKET = 16

PLUS = 17

MINUS = 18

TIMES = 19

DIV = 20

GT = 21

LT = 22

EQ = 23

COMMA = 24

POINT = 25

POW = 26

PI = 27

EULER = 28

I = 29

MIN = 30

MAX = 31

VARIABLE = 32

SCIENTIFIC_NUMBER = 33

WS = 34
```

equation()

expression()

multiplyingExpression()

operatorAddSub()

operatorMulDiv()

powExpression()

signedAtom()

atom()

scientific()

matrixElement()

constant()

variable()

func()

funcname()

relop()

pyg4ometry.gdml.GdmlExpression.GdmlExpressionVisitor

Module Contents

Classes

GdmlExpressionVisitor

class pyg4ometry.gdml.GdmlExpression.GdmlExpressionVisitor.**GdmlExpressionVisitor**

Bases: ParseTreeVisitor

visitEquation(*ctx*)

Parameters
ctx (GdmlExpressionParser.EquationContext) –

visitExpression(*ctx*)

Parameters
ctx (GdmlExpressionParser.ExpressionContext) –

visitMultiplyingExpression(*ctx*)

Parameters
ctx (GdmlExpressionParser.MultiplyingExpressionContext) –

visitOperatorAddSub(*ctx*)

Parameters
ctx (GdmlExpressionParser.OperatorAddSubContext) –

visitOperatorMulDiv(*ctx*)

Parameters
ctx (GdmlExpressionParser.OperatorMulDivContext) –

visitPowExpression(*ctx*)

Parameters
ctx (GdmlExpressionParser.PowExpressionContext) –

visitSignedAtom(*ctx*)

Parameters
ctx (GdmlExpressionParser.SignedAtomContext) –

visitAtom(*ctx*)

Parameters
ctx (GdmlExpressionParser.AtomContext) –

visitScientific(*ctx*)

Parameters
ctx (GdmlExpressionParser.ScientificContext) –

visitMatrixElement(*ctx*)

Parameters
ctx (GdmlExpressionParser.MatrixElementContext) –

visitConstant(*ctx*)

Parameters
ctx (GdmlExpressionParser.ConstantContext) –

visitVariable(*ctx*)

Parameters
ctx (GdmlExpressionParser.VariableContext) –

visitFunc(*ctx*)

Parameters
ctx (GdmlExpressionParser.FuncContext) –

visitFuncname(*ctx*)

Parameters
ctx (GdmlExpressionParser.FuncnameContext) –

visitRelop(*ctx*)

Parameters
ctx (GdmlExpressionParser.RelopContext) –

Package Contents

Classes

ExpressionParser	

class pyg4ometry.gdml.GdmlExpression.**ExpressionParser**

parse(*expression*)

evaluate(*parse_tree*, *define_dict={}*)

get_variables(*parse_tree*)

Submodules

pyg4ometry.gdml.Defines

Module Contents

Classes

BasicExpression	Holds an expression as a string and can use the expression parser
DefineBase	Common bits for a define. Must have a name and a registry. Adding
ScalarBase	Base class for all scalars (Constants, Quantity, Variable and 'Expression')
Constant	GDML constant define wrapper object
Quantity	GDML quantity define wrapper object
Variable	GDML variable define wrapper object
Expression	General expression, does not have an analogue in GDML
VectorBase	
Position	GDML position define wrapper object
Rotation	GDML rotation define wrapper object
Scale	GDML scale define wrapper object
Matrix	GDML matrix define wrapper object
Auxiliary	Auxiliary information container object

Functions

upgradeToStringExpression(reg, obj)	Take a float, str, ScalarBase and return string expression.
evaluateToFloat(reg, obj)	
upgradeToExpression(reg, obj)	Helper functions that takes a string and returns an expression object or a string
upgradeToVector(var, reg[, type, unit, addRegistry])	Take a list [x,y,z] and create a vector
upgradeToTransformation(var, reg[, addRegistry])	Take a list of lists [[rx,ry,rz],[x,y,z]] and create a transformation [Rotation,Position]
operationReturnType(name, strExpr, v1, v2, type1, ...)	
sin(arg)	Sin of a ScalarBase object, returns a Constant
cos(arg)	Cosine of a ScalarBase object, returns a Constant
tan(arg)	Tangent of a ScalarBase object, returns a Constant
asin(arg)	ArcSin of a ScalarBase object, returns a Constant
acos(arg)	ArcCos of a ScalarBase object, returns a Constant
atan(arg)	ArcTan of a ScalarBase object, returns a Constant
exp(arg)	Exponential of a ScalarBase object, returns a Constant
log(arg)	Natural logarithm of a ScalarBase object, returns a Constant
log10(arg)	Base 10 logarithm of a ScalarBase object, returns a Constant
sqrt(arg)	Square root of a ScalarBase object, returns a Constant
pow(arg, power)	arg raised to power
abs(arg)	absolute value of arg
min(arg1, arg2)	absolute value of arg
max(arg1, arg2)	absolute value of arg
MatrixFromVectors(e, v, name, registry[, eunit, vunit])	Creates a GDML Matrix from an energy and a value vector

class pyg4ometry.gdml.Defines.**BasicExpression**(*name*, *expressionString*, *registry*)

Holds an expression as a string and can use the expression parser in the supplied registry to evaluate it. A registry is required.

Parameters

- **name** (*str*) – Name of the expression object
- **expressionString** (*str*) – Expression itself as a string e.g. "12.0" or "a + 3.0"
- **registry** (pyg4ometry.geant4.Registry.Registry) – The registry object to give context for any variables used.

```
>>> r = pyg4ometry.geant4.Registry()
>>> a = BasicExpression("a", "3.0", r)
>>> float(a)
>>> str(a)
```

eval()

variables(*allDependents=False*)

simp()

__repr__()
Return repr(self).

__float__()

str()

pyg4ometry.gdml.Defines.**upgradeToStringExpression**(*reg*, *obj*)
Take a float, str, ScalarBase and return string expression.

Parameters

- **reg** (Registry) – Registry for lookup in define dictionary
- **obj** (*str*,*float*,ScalarBase) – Object to upgrade

Returns
String expression

Return type
str

pyg4ometry.gdml.Defines.**evaluateToFloat**(*reg*, *obj*)

pyg4ometry.gdml.Defines.**upgradeToExpression**(*reg*, *obj*)
Helper functions that takes a string and returns an expression object or a string

pyg4ometry.gdml.Defines.**upgradeToVector**(*var*, *reg*, *type='position'*, *unit=''*, *addRegistry=False*)
Take a list [x,y,z] and create a vector

Parameters

- **var** (*list of str, float*, Constant, Quantity, Variable) – input list to create a position, rotation or scale
- **reg** (Registry) – registry
- **type** (*str*) – class type of vector (position, rotation, scale)
- **addRegistry** (*bool*) – flag to add to registry

pyg4ometry.gdml.Defines.**upgradeToTransformation**(*var*, *reg*, *addRegistry=False*)
Take a list of lists [[rx,ry,rz],[x,y,z]] and create a transformation [Rotation,Position]

Parameters

- **var** (*list of str, float*, Constant, Quantity, Variable) – input list to create a transformation
- **reg** (Registry) – registry
- **addRegistry** (*bool*) – flag to add to registry

pyg4ometry.gdml.Defines.**operationReturnType**(*name*, *strExpr*, *v1*, *v2*, *type1*, *type2*, *reg*)

class pyg4ometry.gdml.Defines.**DefineBase**(*name=''*, *registry=None*)
Common bits for a define. Must have a name and a registry. Adding to the registry can't be done here as it must be done by the derived type.

setName(*name*)
Set name of the object.

Parameters
name (*str*) – name of object

setRegistry(*registry*)

class pyg4ometry.gdml.Defines.ScalarBase(*typeName*, *name=''*, *registry=None*)

Bases: DefineBase

Base class for all scalars (Constants, Quantity, Variable and 'Expression')

__radd__

__rmul__

setName(*name*)

Set name of scalar

Parameters
name (str) – name of object

setExpression(*expressionString*)

Take a string and make it into the BasicExpression type for this object.

Parameters
expressionString (str) – Expression to store.

setRegistry(*registry*)

eval()

Evaluate the expression

Returns
numerical evaluation of Constant

Return type
float

__repr__()

Return repr(self).

__float__()

__add__(*other*)

__sub__(*other*)

__rsub__(*other*)

__mul__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

__neg__()

__abs__()

__pow__(*power*)

pyg4ometry.gdml.Defines.sin(*arg*)

Sin of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of sin

pyg4ometry.gdml.Defines.**cos**(*arg*)

Cosine of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of cos

pyg4ometry.gdml.Defines.**tan**(*arg*)

Tangent of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of tan

pyg4ometry.gdml.Defines.**asin**(*arg*)

ArcSin of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of asin

pyg4ometry.gdml.Defines.**acos**(*arg*)

ArcCos of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of acos

pyg4ometry.gdml.Defines.**atan**(*arg*)

ArcTan of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of tan

pyg4ometry.gdml.Defines.**exp**(*arg*)

Exponential of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of exp

pyg4ometry.gdml.Defines.**log**(*arg*)

Natural logarithm of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of log

pyg4ometry.gdml.Defines.**log10**(*arg*)

Base 10 logarithm of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of log10

pyg4ometry.gdml.Defines.**sqrt**(*arg*)

Square root of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of sin

pyg4ometry.gdml.Defines.**pow**(*arg*, *power*)

arg raised to power

Parameters

- **arg** (Constant, Quantity, Variable *or* Expression) – Argument of x**y
- **power** (*float*) – y

pyg4ometry.gdml.Defines.**abs**(*arg*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

pyg4ometry.gdml.Defines.**min**(*arg1*, *arg2*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

pyg4ometry.gdml.Defines.**max**(*arg1*, *arg2*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

class pyg4ometry.gdml.Defines.**Constant**(*name*, *value*, *registry*, *addRegistry=True*)

Bases: *ScalarBase*

GDML constant define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__lt__(*other*)

Return self<value.

__gt__(*other*)

Return self>value.

__le__(*other*)

Return self<=value.

__ge__(*other*)

Return self>=value.

class pyg4ometry.gdml.Defines.**Quantity**(*name*, *value*, *unit*, *type*, *registry*, *addRegistry=True*)

Bases: *ScalarBase*

GDML quantity define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **unit** (*str*) – unit of the quantity

- **type** (*not sure*) – type of quantity
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__repr__()
Return repr(self).

eval()
Evaluate the expression

Returns
numerical evaluation of Constant

Return type
float

class pyg4ometry.gdml.Defines.**Variable**(*name*, *value*, *registry*, *addRegistry=True*)
Bases: *ScalarBase*

GDML variable define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define

class pyg4ometry.gdml.Defines.**Expression**(*name*, *value*, *registry*, *addRegistry=False*)
Bases: *ScalarBase*

General expression, does not have an analogue in GDML

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__int__()

class pyg4ometry.gdml.Defines.**VectorBase**(*typeName*, *name*, *registry*)

__rmul__

__repr__()
Return repr(self).

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__truediv__(*other*)

setName(*name*)

Set name of vector

Parameters
name (*str*) – name of object

eval()

Evaluate vector

Returns
numerical evaluation of vector

Return type
list of floats

nonzero()

Evaluate vector

Returns
Check if the vector is trivial (all elements zero)

Return type
bool

__getitem__(*key*)

setRegistry(*registry*)

class pyg4ometry.gdml.Defines.**Position**(*name*, *x*, *y*, *z*, *unit='mm'*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML position define wrapper object

Parameters

- **x** (*float*, Constant, Quantity, Variable) – x component of position
- **y** (*float*, Constant, Quantity, Variable) – y component of position
- **z** (*float*, Constant, Quantity, Variable) – z component of position

class pyg4ometry.gdml.Defines.**Rotation**(*name*, *rx*, *ry*, *rz*, *unit='rad'*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML rotation define wrapper object

Parameters

- **rx** (*float*, Constant, Quantity, Variable) – rotation around x axis
- **ry** (*float*, Constant, Quantity, Variable) – rotation around y axis
- **rz** (*float*, Constant, Quantity, Variable) – rotation around z axis

class pyg4ometry.gdml.Defines.**Scale**(*name*, *sx*, *sy*, *sz*, *unit=None*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML scale define wrapper object

Parameters

- **sx** (*float*, Constant, Quantity, Variable) – x component of scale
- **sy** (*float*, Constant, Quantity, Variable) – y component of scale
- **sz** (*float*, Constant, Quantity, Variable) – z component of scale

class pyg4ometry.gdml.Defines.**Matrix**(*name*, *coldim*, *values*, *registry=None*, *addRegistry=True*)

GDML matrix define wrapper object

Parameters

- **name** (*str*) – of matrix for registry
- **coldim** – is number of columns
- **coldim** – int
- **values** (*list of float, str, Constant, Quantity, Variable*) – list of values for matrix
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add matrix to registry

eval()

Evaluate matrix

Returns
numerical evaluation of matrix

Return type
numpy.array

__repr__()

Return repr(self).

__getitem__(*key*)

pyg4ometry.gdml.Defines.**MatrixFromVectors**(*e*, *v*, *name*, *registry*, *eunit='eV'*, *vunit=''*)

Creates a GDML Matrix from an energy and a value vector

Parameters

- **name** (*str*) – of matrix of registry
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **registry** (Registry) – for storing define
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.gdml.Defines.**Auxiliary**(*auxtype*, *auxvalue*, *registry=None*, *unit=''*, *addRegistry=True*)

Auxiliary information container object

addSubAuxiliary(*aux*)

Add a sub-auxiliary inside the scope of the current auxiliary

Parameters
aux (*object, gdml.Defines.Auxiliary*) – auxiliary definition

pyg4ometry.gdml.Reader

Module Contents

Classes

Reader	Read a GDML file.

Functions

isComment(node)	
_StripPointer(name)	

pyg4ometry.gdml.Reader.**isComment**(*node*)

class pyg4ometry.gdml.Reader.**Reader**(*fileName*, *registryOn=True*, *skipMaterials=False*, *reduceNISTMaterialsToPredefined=False*)

Read a GDML file.

Parameters

- **fileName** (*str*) – path to gdml file to load
- **registryOn** (*bool*) – whether to build a registry
- **reduceNISTMaterialsToPredefined** (*bool*) – change NIST-named materials to predefined ones

When loading a GDML file that was exported by Geant4, the NIST materials may be fully expanded to include their full element / isotope composition. With the reduceNISTMaterialsToPredefined flag set to True, these will be ignored and the materials that have a name that matches a NIST one will be 'reduced' back to a predefined material by name only.

load()

getRegistry()

parseDefines(*xmldoc*)

parseVector(*node*, *type='position'*, *addRegistry=True*)

parseMaterials(*xmldoc*)

_makeMaterials(*materials*, *elements*, *isotopes*)

Construct the isotopes, elements and materials in that order. These aren't returned, but simply constructed in and therefore exist in the registry.

parseUserInfo(*xmldoc*)

_parseAuxiliary(*xmlnode*, *register=True*)

parseSolids(*xmldoc*)

parseBox(*node*)

parseTube(*node*)

parseCutTube(*node*)

parseCone(*node*)

parsePara(*node*)

parseTrd(*node*)

parseTrap(*node*)

parseSphere(*node*)

parseOrb(*node*)

parseTorus(*node*)

parsePolycone(*node*)

parseGenericPolycone(*node*)

parsePolyhedra(*node*)

parseGenericPolyhedra(*node*)

parseEllipticalTube(*node*)

parseEllipsoid(*node*)

parseEllipticalCone(*node*)

parseParaboloid(*node*)

parseHype(*node*)

parseTet(*node*)

parseExtrudedSolid(*node*)

parseTwistedBox(*node*)

parseTwistedTrap(*node*)

parseTwistedTrd(*node*)

parseTwistedTubs(*node*)

parseGenericTrap(*node*)

parseTessellatedSolid(*node*)

parseUnion(*node*)

parseSubtraction(*node*)

parseIntersection(*node*)

parseMultiUnion(*node*)

parseOpticalSurface(*node*)

parseScaledSolid(*node*)

parseSolidLoop(*node*)

parseStructure(*xmldoc*, *materialSubstitutionNames=None*)

extractStructureNodeData(*node*, *materialSubstitutionNames=None*)

parsePhysicalVolumeChildren(*node*, *vol*)

pyg4ometry.gdml.Reader.**_StripPointer**(*name*)

pyg4ometry.gdml.Units

Module Contents

Functions

unit(unitString)

Attributes

units

pyg4ometry.gdml.Units.**units**

pyg4ometry.gdml.Units.**unit**(*unitString*)

pyg4ometry.gdml.Writer

Module Contents

Classes

Writer

class pyg4ometry.gdml.Writer.**Writer**(*prepend=''*)

addDetector(*registry*)

write(*filename*)

writeGMADTesterNoBeamline(*gmad*, *gdml*)

writeGmadTester(*filenameGmad*, *filenameGDML*, *writeDefaultLattice=False*, *preprocessGDML=True*, *energy=250*)

writeDefaultLattice(*filename='lattice.gmad'*)

checkDefineName(*defineName*)

checkMaterialName(*materialName*)

checkSolidName(*solidName*)

checkLogicalVolumeName(*logicalVolumeName*)

checkPhysicalVolumeName(*physicalVolumeName*)

writeVectorVariable(*node*, *vector_var*, *allow_ref=True*, *suppress_trivial=True*)

Writes an XML child node for a vector variable - position, rotation, scale. If allow_ref is enabled, it will write a ref to a registry define where possible. If suppress_trivial is enabled it won't write vectors with all elements zero.

writeDefine(*define*)

writeMaterialProps(*material*, *oe*)

writeMaterial(*material*)

writeLogicalVolume(*lv*)

writeAuxiliary(*aux*, *parent=None*)

writeAssemblyVolume(*lv*)

writePhysicalVolume(*pv*)

writeReplicaVolume(*instance*)

writeDivisionVolume(*instance*)

writeParametrisedVolume(*instance*)

writeSkinSurface(*instance*)

writeBorderSurface(*instance*)

writeSolid(*solid*)

Dispatch to correct member function based on type string in SolidBase.

getValueOrExpr(*var*)

getValueOrExprFromInstance(*instance*, *variable*, *index=None*)

writeBox(*instance*)

writeCons(*instance*)

writeCutTubs(*instance*)

writeEllipsoid(*instance*)

writeEllipticalCone(*instance*)

writeEllipticalTube(*instance*)

createTwoDimVertex(*x*, *y*)

createSection(*zOrder*, *zPosition*, *xOffset*, *yOffset*, *scalingFactor*)

writeExtrudedSolid(*instance*)

createrzPoint(*r*, *z*)

writeGenericPolycone(*instance*)

writeGenericPolyhedra(*instance*)

createTriangularFacet(*vertex1*, *vertex2*, *vertex3*)

createQuadrangularFacet(*vertex1*, *vertex2*, *vertex3*, *vertex4*)

writeTessellatedSolid(*instance*)

writeHype(*instance*)

writeIntersection(*instance*)

writeOpticalSurface(*instance*)

writeOrb(*instance*)

writePara(*instance*)

writeParaboloid(*instance*)

createzPlane(*rInner*, *rOuter*, *zplane*)

writePolycone(*instance*)

writePolyhedra(*instance*)

writeSphere(*instance*)

writeGenericTrap(*instance*)

createPosition(*name*, *x*, *y*, *z*)

writeTet(*instance*)

writeTorus(*instance*)

writeTrap(*instance*)

writeTrd(*instance*)

writeTubs(*instance*)

writeTwistedBox(*instance*)

writeTwistedTrd(*instance*)

writeTwistedTrap(*instance*)

writeTwistedTubs(*instance*)

writeUnion(*instance*)

writeSubtraction(*instance*)

writeIntersection(*instance*)

writeMultiUnion(*instance*)

writeScaled(*instance*)

Package Contents

Classes

Reader	
_Material	This class provides an interface to GDML material definitions.
_Element	This class provides an interface to GDML material definitions. Because of the different options
_Isotope	This class that handles isotopes as components of composite materials. An element can be
Writer	
BasicExpression	Holds an expression as a string and can use the expression parser
DefineBase	Common bits for a define. Must have a name and a registry. Adding
ScalarBase	Base class for all scalars (Constants, Quantity, Variable and 'Expression')
Constant	GDML constant define wrapper object
Quantity	GDML quantity define wrapper object
Variable	GDML variable define wrapper object
Expression	General expression, does not have an analogue in GDML
VectorBase	
Position	GDML position define wrapper object
Rotation	GDML rotation define wrapper object
Scale	GDML scale define wrapper object
Matrix	GDML matrix define wrapper object
Auxiliary	Auxiliary information container object

Functions

isComment(node)	
_StripPointer(name)	
upgradeToStringExpression(reg, obj)	Take a float, str, ScalarBase and return string expression.
evaluateToFloat(reg, obj)	
upgradeToExpression(reg, obj)	Helper functions that takes a string and returns an expression object or a string
upgradeToVector(var, reg[, type, unit, addRegistry])	Take a list [x,y,z] and create a vector
upgradeToTransformation(var, reg[, addRegistry])	Take a list of lists [[rx,ry,rz],[x,y,z]] and create a transformation [Rotation,Position]
operationReturnType(name, strExpr, v1, v2, type1, ...)	
sin(arg)	Sin of a ScalarBase object, returns a Constant
cos(arg)	Cosine of a ScalarBase object, returns a Constant
tan(arg)	Tangent of a ScalarBase object, returns a Constant
asin(arg)	ArcSin of a ScalarBase object, returns a Constant
acos(arg)	ArcCos of a ScalarBase object, returns a Constant
atan(arg)	ArcTan of a ScalarBase object, returns a Constant
exp(arg)	Exponential of a ScalarBase object, returns a Constant
log(arg)	Natural logarithm of a ScalarBase object, returns a Constant
log10(arg)	Base 10 logarithm of a ScalarBase object, returns a Constant
sqrt(arg)	Square root of a ScalarBase object, returns a Constant
pow(arg, power)	arg raised to power
abs(arg)	absolute value of arg
min(arg1, arg2)	absolute value of arg
max(arg1, arg2)	absolute value of arg
MatrixFromVectors(e, v, name, registry[, eunit, vunit])	Creates a GDML Matrix from an energy and a value vector

pyg4ometry.gdml.**isComment**(*node*)

class pyg4ometry.gdml.**Reader**(*fileName*, *registryOn=True*, *skipMaterials=False*, *reduceNISTMaterialsToPredefined=False*)

Read a GDML file.

Parameters

- **fileName** (*str*) – path to gdml file to load
- **registryOn** (*bool*) – whether to build a registry
- **reduceNISTMaterialsToPredefined** (*bool*) – change NIST-named materials to predefined ones

When loading a GDML file that was exported by Geant4, the NIST materials may be fully expanded to include their full element / isotope composition. With the reduceNISTMaterialsToPredefined flag set to True, these will be ignored and the materials that have a name that matches a NIST one will be 'reduced' back to a predefined material by name only.

load()

getRegistry()

parseDefines(*xmldoc*)

parseVector(*node*, *type='position'*, *addRegistry=True*)

parseMaterials(*xmldoc*)

_makeMaterials(*materials*, *elements*, *isotopes*)

Construct the isotopes, elements and materials in that order. These aren't returned, but simply constructed in and therefore exist in the registry.

parseUserInfo(*xmldoc*)

_parseAuxiliary(*xmlnode*, *register=True*)

parseSolids(*xmldoc*)

parseBox(*node*)

parseTube(*node*)

parseCutTube(*node*)

parseCone(*node*)

parsePara(*node*)

parseTrd(*node*)

parseTrap(*node*)

parseSphere(*node*)

parseOrb(*node*)

parseTorus(*node*)

parsePolycone(*node*)

parseGenericPolycone(*node*)

parsePolyhedra(*node*)

parseGenericPolyhedra(*node*)

parseEllipticalTube(*node*)

parseEllipsoid(*node*)

parseEllipticalCone(*node*)

parseParaboloid(*node*)

parseHype(*node*)

parseTet(*node*)

parseExtrudedSolid(*node*)

parseTwistedBox(*node*)

parseTwistedTrap(*node*)

parseTwistedTrd(*node*)

parseTwistedTubs(*node*)

parseGenericTrap(*node*)

parseTessellatedSolid(*node*)

parseUnion(*node*)

parseSubtraction(*node*)

parseIntersection(*node*)

parseMultiUnion(*node*)

parseOpticalSurface(*node*)

parseScaledSolid(*node*)

parseSolidLoop(*node*)

parseStructure(*xmldoc*, *materialSubstitutionNames=None*)

extractStructureNodeData(*node*, *materialSubstitutionNames=None*)

parsePhysicalVolumeChildren(*node*, *vol*)

pyg4ometry.gdml.**_StripPointer**(*name*)

class pyg4ometry.gdml.**_Material**(***kwargs*)

Bases: MaterialBase

This class provides an interface to GDML material definitions.

Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

MaterialSingleElement MaterialCompound MaterialPredefined

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string density - float atomic_number - int atomic_weight - float number_of_components - int state - string pressure - float pressure_unit - string temperature - float temperature_unit - string

property state_variables

add_element_massfraction(*element*, *massfraction*)

Add an element as a component to a material as a fraction of the material mass. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance massfraction - float, 0.0 < massfraction <= 1.0

add_element_natoms(*element*, *natoms*)

Add an element as a component to a material as a number of atoms in the material molecule. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance natoms - int, number of atoms in the compound molecule

add_material(*material*, *fractionmass*)

Add a material as a component to another material (mixture) as a fraction of the mixture mass. Can only add new materials to materials defined as composite.

Inputs:

material - pyg4ometry.geant4.Material.Material instance massfraction - float, 0.0 < massfraction <= 1.0

set_pressure(*value*, *unit='pascal'*)

set_temperature(*value*, *unit='K'*)

__str__()

Return str(self).

addProperty(*name*, *matrix*)

Add a material property from a matrix.

Parameters

- **name** (*str*) – key of the material property
- **matrix** (Matrix) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str,float,int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.gdml._Element(***kwargs*)

Bases: MaterialBase

This class provides an interface to GDML material definitions. Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

ElementSimple ElementIsotopeMixture

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string symbol - string Z - int A - int n_comp - int

add_isotope(*isotope*, *abundance*)

Add an isotope as a component to an element as an abundance fraction in the element.

Inputs:
element - pyg4ometry.geant4.Material.Isotope instance abundance - float, 0.0 < abundance <= 1.0

class pyg4ometry.gdml.**_Isotope**(*name*, *Z*, *N*, *a*, *registry=None*)

Bases: MaterialBase

This class that handles isotopes as components of composite materials. An element can be defined as a mixture of isotopes.

Inputs:
name - string Z - int, atomic number N - int, mass number a - float, molar weight in g/mole

class pyg4ometry.gdml.**Writer**(*prepend=''*)

addDetector(*registry*)

write(*filename*)

writeGMADTesterNoBeamline(*gmad*, *gdml*)

writeGmadTester(*filenameGmad*, *filenameGDML*, *writeDefaultLattice=False*, *preprocessGDML=True*, *energy=250*)

writeDefaultLattice(*filename='lattice.gmad'*)

checkDefineName(*defineName*)

checkMaterialName(*materialName*)

checkSolidName(*solidName*)

checkLogicalVolumeName(*logicalVolumeName*)

checkPhysicalVolumeName(*physicalVolumeName*)

writeVectorVariable(*node*, *vector_var*, *allow_ref=True*, *suppress_trivial=True*)

Writes an XML child node for a vector variable - position, rotation, scale. If allow_ref is enabled, it will write a ref to a registry define where possible. If suppress_trivial is enabled it won't write vectors with all elements zero.

writeDefine(*define*)

writeMaterialProps(*material*, *oe*)

writeMaterial(*material*)

writeLogicalVolume(*lv*)

writeAuxiliary(*aux*, *parent=None*)

writeAssemblyVolume(*lv*)

writePhysicalVolume(*pv*)

writeReplicaVolume(*instance*)

writeDivisionVolume(*instance*)

writeParametrisedVolume(*instance*)

writeSkinSurface(*instance*)

writeBorderSurface(*instance*)

writeSolid(*solid*)

Dispatch to correct member function based on type string in SolidBase.

getValueOrExpr(*var*)

getValueOrExprFromInstance(*instance*, *variable*, *index=None*)

writeBox(*instance*)

writeCons(*instance*)

writeCutTubs(*instance*)

writeEllipsoid(*instance*)

writeEllipticalCone(*instance*)

writeEllipticalTube(*instance*)

createTwoDimVertex(*x*, *y*)

createSection(*zOrder*, *zPosition*, *xOffset*, *yOffset*, *scalingFactor*)

writeExtrudedSolid(*instance*)

createrzPoint(*r*, *z*)

writeGenericPolycone(*instance*)

writeGenericPolyhedra(*instance*)

createTriangularFacet(*vertex1*, *vertex2*, *vertex3*)

createQuadrangularFacet(*vertex1*, *vertex2*, *vertex3*, *vertex4*)

writeTessellatedSolid(*instance*)

writeHype(*instance*)

writeIntersection(*instance*)

writeOpticalSurface(*instance*)

writeOrb(*instance*)

writePara(*instance*)

writeParaboloid(*instance*)

createzPlane(*rInner*, *rOuter*, *zplane*)

writePolycone(*instance*)

writePolyhedra(*instance*)

writeSphere(*instance*)

writeGenericTrap(*instance*)

createPosition(*name*, *x*, *y*, *z*)

writeTet(*instance*)

writeTorus(*instance*)

writeTrap(*instance*)

writeTrd(*instance*)

writeTubs(*instance*)

writeTwistedBox(*instance*)

writeTwistedTrd(*instance*)

writeTwistedTrap(*instance*)

writeTwistedTubs(*instance*)

writeUnion(*instance*)

writeSubtraction(*instance*)

writeIntersection(*instance*)

writeMultiUnion(*instance*)

writeScaled(*instance*)

class pyg4ometry.gdml.**BasicExpression**(*name*, *expressionString*, *registry*)

Holds an expression as a string and can use the expression parser in the supplied registry to evaluate it. A registry is required.

Parameters

- **name** (*str*) – Name of the expression object
- **expressionString** (*str*) – Expression itself as a string e.g. "12.0" or "a + 3.0"
- **registry** (pyg4ometry.geant4.Registry.Registry) – The registry object to give context for any variables used.

```
>>> r = pyg4ometry.geant4.Registry()
>>> a = BasicExpression("a", "3.0", r)
>>> float(a)
>>> str(a)
```

eval()

variables(*allDependents=False*)

simp()

__repr__()
Return repr(self).

__float__()

str()

pyg4ometry.gdml.**upgradeToStringExpression**(*reg*, *obj*)
Take a float, str, ScalarBase and return string expression.

Parameters

- **reg** (Registry) – Registry for lookup in define dictionary
- **obj** (*str*,*float*,ScalarBase) – Object to upgrade

Returns
String expression

Return type
str

pyg4ometry.gdml.**evaluateToFloat**(*reg*, *obj*)

pyg4ometry.gdml.**upgradeToExpression**(*reg*, *obj*)
Helper functions that takes a string and returns an expression object or a string

pyg4ometry.gdml.**upgradeToVector**(*var*, *reg*, *type='position'*, *unit=''*, *addRegistry=False*)
Take a list [x,y,z] and create a vector

Parameters

- **var** (*list of str, float*, Constant, Quantity, Variable) – input list to create a position, rotation or scale
- **reg** (Registry) – registry
- **type** (*str*) – class type of vector (position, rotation, scale)
- **addRegistry** (*bool*) – flag to add to registry

pyg4ometry.gdml.**upgradeToTransformation**(*var*, *reg*, *addRegistry=False*)
Take a list of lists [[rx,ry,rz],[x,y,z]] and create a transformation [Rotation,Position]

Parameters

- **var** (*list of str, float*, Constant, Quantity, Variable) – input list to create a transformation
- **reg** (Registry) – registry
- **addRegistry** (*bool*) – flag to add to registry

pyg4ometry.gdml.**operationReturnType**(*name*, *strExpr*, *v1*, *v2*, *type1*, *type2*, *reg*)

class pyg4ometry.gdml.**DefineBase**(*name=''*, *registry=None*)
Common bits for a define. Must have a name and a registry. Adding to the registry can't be done here as it must be done by the derived type.

setName(*name*)
Set name of the object.

Parameters
name (*str*) – name of object

setRegistry(*registry*)

class pyg4ometry.gdml.**ScalarBase**(*typeName*, *name=''*, *registry=None*)

Bases: *DefineBase*

Base class for all scalars (Constants, Quantity, Variable and 'Expression')

__radd__

__rmul__

setName(*name*)

Set name of scalar

Parameters
name (*str*) – name of object

setExpression(*expressionString*)

Take a string and make it into the BasicExpression type for this object.

Parameters
expressionString (*str*) – Expression to store.

setRegistry(*registry*)

eval()

Evaluate the expression

Returns
numerical evaluation of Constant

Return type
float

__repr__()

Return repr(self).

__float__()

__add__(*other*)

__sub__(*other*)

__rsub__(*other*)

__mul__(*other*)

__truediv__(*other*)

__rtruediv__(*other*)

__neg__()

__abs__()

__pow__(*power*)

pyg4ometry.gdml.**sin**(*arg*)

Sin of a ScalarBase object, returns a Constant

Parameters
arg (*Constant, Quantity, Variable or Expression*) – Argument of sin

pyg4ometry.gdml.**cos**(*arg*)

Cosine of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of cos

pyg4ometry.gdml.**tan**(*arg*)

Tangent of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of tan

pyg4ometry.gdml.**asin**(*arg*)

ArcSin of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of asin

pyg4ometry.gdml.**acos**(*arg*)

ArcCos of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of acos

pyg4ometry.gdml.**atan**(*arg*)

ArcTan of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of tan

pyg4ometry.gdml.**exp**(*arg*)

Exponential of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of exp

pyg4ometry.gdml.**log**(*arg*)

Natural logarithm of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of log

pyg4ometry.gdml.**log10**(*arg*)

Base 10 logarithm of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of log10

pyg4ometry.gdml.**sqrt**(*arg*)

Square root of a ScalarBase object, returns a Constant

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of sin

pyg4ometry.gdml.**pow**(*arg*, *power*)

arg raised to power

Parameters

- **arg** (Constant, Quantity, Variable *or* Expression) – Argument of x**y
- **power** (*float*) – y

pyg4ometry.gdml.**abs**(*arg*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

pyg4ometry.gdml.**min**(*arg1*, *arg2*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

pyg4ometry.gdml.**max**(*arg1*, *arg2*)

absolute value of arg

Parameters
arg (Constant, Quantity, Variable *or* Expression) – Argument of abs(arg)

class pyg4ometry.gdml.**Constant**(*name*, *value*, *registry*, *addRegistry=True*)

Bases: *ScalarBase*

GDML constant define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__lt__(*other*)

Return self<value.

__gt__(*other*)

Return self>value.

__le__(*other*)

Return self<=value.

__ge__(*other*)

Return self>=value.

class pyg4ometry.gdml.**Quantity**(*name*, *value*, *unit*, *type*, *registry*, *addRegistry=True*)

Bases: *ScalarBase*

GDML quantity define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **unit** (*str*) – unit of the quantity

- **type** (*not sure*) – type of quantity
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__repr__()
Return repr(self).

eval()
Evaluate the expression

Returns
numerical evaluation of Constant

Return type
float

class pyg4ometry.gdml.**Variable**(*name*, *value*, *registry*, *addRegistry=True*)
Bases: *ScalarBase*

GDML variable define wrapper object

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define

class pyg4ometry.gdml.**Expression**(*name*, *value*, *registry*, *addRegistry=False*)
Bases: *ScalarBase*

General expression, does not have an analogue in GDML

Parameters

- **name** (*str*) – of constant for registry
- **value** (*float*,*str*,Constant,Quantity,Variable) – expression for constant
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add constant to registry

__int__()

class pyg4ometry.gdml.**VectorBase**(*typeName*, *name*, *registry*)

__rmul__

__repr__()
Return repr(self).

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__truediv__(*other*)

setName(*name*)

Set name of vector

Parameters
name (*str*) – name of object

eval()

Evaluate vector

Returns
numerical evaluation of vector

Return type
list of floats

nonzero()

Evaluate vector

Returns
Check if the vector is trivial (all elements zero)

Return type
bool

__getitem__(*key*)

setRegistry(*registry*)

class pyg4ometry.gdml.**Position**(*name*, *x*, *y*, *z*, *unit='mm'*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML position define wrapper object

Parameters

- **x** (*float*, Constant, Quantity, Variable) – x component of position
- **y** (*float*, Constant, Quantity, Variable) – y component of position
- **z** (*float*, Constant, Quantity, Variable) – z component of position

class pyg4ometry.gdml.**Rotation**(*name*, *rx*, *ry*, *rz*, *unit='rad'*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML rotation define wrapper object

Parameters

- **rx** (*float*, Constant, Quantity, Variable) – rotation around x axis
- **ry** (*float*, Constant, Quantity, Variable) – rotation around y axis
- **rz** (*float*, Constant, Quantity, Variable) – rotation around z axis

class pyg4ometry.gdml.**Scale**(*name*, *sx*, *sy*, *sz*, *unit=None*, *registry=None*, *addRegistry=True*)

Bases: *VectorBase*

GDML scale define wrapper object

Parameters

- **sx** (*float*, Constant, Quantity, Variable) – x component of scale
- **sy** (*float*, Constant, Quantity, Variable) – y component of scale
- **sz** (*float*, Constant, Quantity, Variable) – z component of scale

class pyg4ometry.gdml.**Matrix**(*name*, *coldim*, *values*, *registry=None*, *addRegistry=True*)

GDML matrix define wrapper object

Parameters

- **name** (*str*) – of matrix for registry
- **coldim** – is number of columns
- **coldim** – int
- **values** (*list of float, str, Constant, Quantity, Variable*) – list of values for matrix
- **registry** (Registry) – for storing define
- **addRegistry** (*bool*) – add matrix to registry

eval()

Evaluate matrix

Returns
numerical evaluation of matrix

Return type
numpy.array

__repr__()

Return repr(self).

__getitem__(*key*)

pyg4ometry.gdml.**MatrixFromVectors**(*e*, *v*, *name*, *registry*, *eunit='eV'*, *vunit=''*)

Creates a GDML Matrix from an energy and a value vector

Parameters

- **name** (*str*) – of matrix of registry
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **registry** (Registry) – for storing define
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.gdml.**Auxiliary**(*auxtype*, *auxvalue*, *registry=None*, *unit=''*, *addRegistry=True*)

Auxiliary information container object

addSubAuxiliary(*aux*)

Add a sub-auxiliary inside the scope of the current auxiliary

Parameters
aux (*object, gdml.Defines.Auxiliary*) – auxiliary definition

pyg4ometry.geant4

Geant4 classes. The classes mainly match those of Geant4

Subpackages

pyg4ometry.geant4.solid

Submodules

pyg4ometry.geant4.solid.Box

Module Contents

Classes

Box	Constructs a box. Note the lengths are full lengths and not half lengths as in Geant4

Functions

cubeNet(vecList)	

pyg4ometry.geant4.solid.Box.**cubeNet**(*vecList*)

class pyg4ometry.geant4.solid.Box.**Box**(*name*, *pX*, *pY*, *pZ*, *registry*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a box. Note the lengths are full lengths and not half lengths as in Geant4

Parameters

- **name** (*str*) – of solid for registry
- **pX** (*float*, Constant, Quantity, Variable, Expression) – length along x
- **pY** (*float*, Constant, Quantity, Variable, Expression) – length along y
- **pZ** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.Cons

Module Contents

Classes

Cons	Constructs a conical section.

class pyg4ometry.geant4.solid.Cons.**Cons**(*name*, *pRmin1*, *pRmax1*, *pRmin2*, *pRmax2*, *pDz*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a conical section.

Parameters

- **name** (*str*) – of the solid
- **pRMin1** (*float*, Constant, Quantity, Variable) – inner radius at -pDz/2
- **pRMax1** (*float*, Constant, Quantity, Variable) – outer radius at -pDz/2
- **pRMin2** (*float*, Constant, Quantity, Variable) – inner radius at +pDZ/2
- **pRMax2** (*float*, Constant, Quantity, Variable) – outer radius at +pDz/2
- **pDz** (*float*, Constant, Quantity, Variable) – length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment in radians
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

checkParameters()

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.CutTubs

Module Contents

Classes

CutTubs	Constructs a cylindrical section with end face cuts. Note pLowNorm and pHighNorm can be

class pyg4ometry.geant4.solid.CutTubs.**CutTubs**(*name*, *pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *pLowNorm*, *pHighNorm*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cylindrical section with end face cuts. Note pLowNorm and pHighNorm can be lists of floats, Constants, Quantities or Variables.

Parameters

- **name** (*str*) – of solid for registry
- **pRMin** (*float*, Constant, Quantity, Variable) – Inner radius
- **pRMax** (*float*, Constant, Quantity, Variable) – Outer radius
- **pDz** (*float*, Constant, Quantity, Variable) – length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment
- **pLowNorm** (*list*) – normal vector of the cut plane at -pDz/2
- **pHighNorm** (*list*) – normal vector of the cut plane at +pDz/2

__repr__()

__str__()

mesh()

pyg4ometry.geant4.solid.Ellipsoid

Module Contents

Classes

Ellipsoid	Constructs an ellipsoid optinoally cut by planes perpendicular to the z-axis.

class pyg4ometry.geant4.solid.Ellipsoid.**Ellipsoid**(*name*, *pxSemiAxis*, *pySemiAxis*, *pzSemiAxis*, *pzBottomCut*, *pzTopCut*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs an ellipsoid optinoally cut by planes perpendicular to the z-axis.

Parameters

- **name** (*str*) – of the solid
- **pxSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of x semi axis
- **pySemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of y semi axis
- **pzSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of z semi axis
- **pzBottomCut** (*float*, Constant, Quantity, Variable, Expression) – z-position of bottom cut plane
- **pzTopCut** (*float*, Constant, Quantity, Variable, Expression) – z-position of top cut plane
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.EllipticalCone

Module Contents

Classes

EllipticalCone	Constructs a cone with elliptical cross-section and an

class pyg4ometry.geant4.solid.EllipticalCone.**EllipticalCone**(*name*, *pxSemiAxis*, *pySemiAxis*, *zMax*, *pzTopCut*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cone with elliptical cross-section and an optional cut. Both zMax and pzTopCut are half lengths extending from the centre of the cone, at z=0.

Parameters

- **name** (*str*) – name of the volume
- **pxSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – semiaxis in x at z=0 as a fraction of zMax.
- **pySemiAxis** (*float*, Constant, Quantity, Variable, Expression) – semiaxis in y at z=0 as a fraction of zMax
- **zMax** (*float*, Constant, Quantity, Variable, Expression) – half length of the cone.
- **pzTopCut** (*float*, Constant, Quantity, Variable, Expression) – half length of the cut.

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.EllipticalTube

Module Contents

Classes

EllipticalTube	Constructs a tube of elliptical cross-section.

class pyg4ometry.geant4.solid.EllipticalTube.**EllipticalTube**(*name*, *pDx*, *pDy*, *pDz*, *registry*, *lunit='mm'*, *nstack=None*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tube of elliptical cross-section.

Parameters

- **name** (*str*) – name of the solid
- **pDx** (*float*, Constant, Quantity, Variable, Expression) – length in x
- **pDy** (*float*, Constant, Quantity, Variable, Expression) – length in y
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

__str__()

mesh()

new meshing based of Tubs meshing

pyg4ometry.geant4.solid.ExtrudedSolid

Module Contents

Classes

ExtrudedSolid	Construct an extruded solid

class pyg4ometry.geant4.solid.ExtrudedSolid.**ExtrudedSolid**(*name*, *pPolygon*, *pZslices*, *registry*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Construct an extruded solid

Parameters

- **name** (*str*) – of solid
- **pPolygon** (*list of lists*) – x-y coordinates of vertices for the polygon.
- **pZslices** (*list of lists*) – z-coordinates of a slice, slice offsets in x-y and scaling
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

Example: Triangular prism with 2 slices pPoligon = [[x1,y1],[x2,y2],[x3,y3]] - vertices of polygon in clockwise order zSlices = [[z1,[offsx1, offsy1],scale1],[z2,[offsx2, offsy2],scale2]]

__repr__()

Return repr(self).

__str__()

Return str(self).

polygon_area(*vertices*)

evaluateParameterWithUnits(*varName*)

mesh()

pyg4ometry.geant4.solid.GenericPolycone

Module Contents

Classes

GenericPolycone	Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

class pyg4ometry.geant4.solid.GenericPolycone.**GenericPolycone**(*name*, *pSPhi*, *pDPhi*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – of solid
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

pyg4ometry.geant4.solid.GenericPolyhedra

Module Contents

Classes

GenericPolyhedra	Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

class pyg4ometry.geant4.solid.GenericPolyhedra.**GenericPolyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – name
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **numSide** (*float*, Constant, Quantity, Variable, Expression) – number of polygon sides
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate list
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate list

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

pyg4ometry.geant4.solid.GenericTrap

Module Contents

Classes

GenericTrap	Constructs an arbitrary trapezoid using two quadrilaterals sitting

class pyg4ometry.geant4.solid.GenericTrap.**GenericTrap**(*name*, *v1x*, *v1y*, *v2x*, *v2y*, *v3x*, *v3y*, *v4x*, *v4y*, *v5x*, *v5y*, *v6x*, *v6y*, *v7x*, *v7y*, *v8x*, *v8y*, *dz*, *registry*, *nstack=20*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs an arbitrary trapezoid using two quadrilaterals sitting on two parallel planes. Vertices 1-4 define the quadrilateral at -dz and vertices 5-8 define the quadrilateral at +dz. This solid is called Arb8 in GDML notation.

Parameters

- **name** – string, name of the volume
- **v1x** – vertex 1 x position
- **v1y** – vertex 1 y position
- **v2x** – vertex 2 x position
- **v2y** – vertex 2 y position
- **v3x** – vertex 3 x position

- **v3y** – vertex 3 y position
- **v4x** – vertex 4 x position
- **v4y** – vertex 4 y position
- **v5x** – vertex 5 x position
- **v5y** – vertex 5 y position
- **v6x** – vertex 6 x position
- **v6y** – vertex 6 y position
- **v7x** – vertex 7 x position
- **v7y** – vertex 7 y position
- **v8x** – vertex 8 x position
- **v8y** – vertex 8 y position
- **dz** – half length along z
- **registry** (Registry) – for storing solid

__repr__()

__str__()

polygon_area(*vertices*)

get_vertex(*index*)

makeLayers(*verts_bot*, *verts_top*)

mesh()

pyg4ometry.geant4.solid.Hype

Module Contents

Classes

Hype	Constructs a tube with hyperbolic profile.

class pyg4ometry.geant4.solid.Hype.**Hype**(*name*, *innerRadius*, *outerRadius*, *innerStereo*, *outerStereo*, *lenZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tube with hyperbolic profile.

Parameters

- **name** (*str*) – of solid
- **innerRadius** (*float*, Constant, Quantity, Variable, Expression) – inner radius

- **outerRadius** (*float*, Constant, Quantity, Variable, Expression) – outer radius
- **innerStereo** (*float*, Constant, Quantity, Variable, Expression) – inner stereo angle
- **outerStereo** (*float*, Constant, Quantity, Variable, Expression) – outer stereo angle
- **lenZ** – length along z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

checkParameters()

mesh()

pyg4ometry.geant4.solid.Intersection

Module Contents

Classes

Intersection	Intersection between two solids

class pyg4ometry.geant4.solid.Intersection.**Intersection**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Intersection between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]
- **registry** (Registry) – for storing solid

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.Layer

Module Contents

Classes

Layer	

class pyg4ometry.geant4.solid.Layer.**Layer**(*p1*, *p2*, *p3*, *p4*, *z*)

__getitem__(*index*)

Rotated(*angle*)

__repr__()
Return repr(self).

pyg4ometry.geant4.solid.MultiUnion

Module Contents

Classes

MultiUnion	Union between two or more solids.

class pyg4ometry.geant4.solid.MultiUnion.**MultiUnion**(*name*, *objects*, *transformations*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Union between two or more solids.

Parameters

- **name** (*str*) – of solid
- **objects** – unrotated, untranslated solid objects to form union

- **transformations** – [[rot1,tra1],[rot2,tra2], [rot3,tra3] ..] or [[[a,b,g],[dx,dy,dz]], [[a,b,g],[dx,dy,dz]], [[a,b,g],[dx,dy,dz]], . . .]
- **registry** (Registry) – for storing solid
- **addRegistry** – Add solid to registry

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.OpticalSurface

Module Contents

Classes

OpticalSurface

class pyg4ometry.geant4.solid.OpticalSurface.**OpticalSurface**(*name*, *finish*, *model*, *surf_type*, *value*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

allowed_models = ['glisur', 'unified', 'LUT', 'DAVIS', 'dichroic']

allowed_types = ['dielectric_metal', 'dielectric_dielectric', 'dielectric_LUT', 'dielectric_LUTDAVIS',...

allowed_finishes = ['polished', 'polishedfrontpainted', 'polishedbackpainted', 'ground', 'groundfrontpainted',...

__repr__()

__str__()

addProperty(*name*, *matrix*)

Add a property to this surface from a matrix.

Parameters

- **name** (*str*) – key of the surface property
- **matrix** (Matrix) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit

- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str*,*float*,*int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

pyg4ometry.geant4.solid.Orb

Module Contents

Classes

Orb	Constructs a solid sphere.

class pyg4ometry.geant4.solid.Orb.**Orb**(*name*, *pRMax*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid sphere.

Parameters

- **name** (*str*) – of the sold
- **pRMax** (*float*, Constant, Quantity, Variable, Expression) – outer radius
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.Para

Module Contents

Classes

Para	Constructs a parallelepiped.

class pyg4ometry.geant4.solid.Para.**Para**(*name*, *pDx*, *pDy*, *pDz*, *pAlpha*, *pTheta*, *pPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a parallelepiped.

Parameters

- **name** (*str*) – of the volume
- **pX** (*float*, Constant, Quantity, Variable) – length along x
- **pY** (*float*, Constant, Quantity, Variable) – length along y
- **pZ** (*float*, Constant, Quantity, Variable) – length along z
- **pAlpha** (*float*, Constant, Quantity, Variable) – angle formed by the y axis and the plane joining the centres of the faces parallel tothe z-x plane at -dy/2 and +dy/2
- **pTheta** (*float*, Constant, Quantity, Variable) – polar angle of the line joining the centres of the faces at -dz/2 and +dz/2 in z
- **pPhi** (*float*, Constant, Quantity, Variable) – azimuthal angle of the line joining the centres of the faces at -dx/2 and +dz/2 in z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

pyg4ometry.geant4.solid.Paraboloid

Module Contents

Classes

Paraboloid	Constructs a paraboloid with possible cuts along the z axis.

class pyg4ometry.geant4.solid.Paraboloid.**Paraboloid**(*name*, *pDz*, *pR1*, *pR2*, *registry*, *lunit='mm'*, *nslice=16*, *nstack=8*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a paraboloid with possible cuts along the z axis.

Parameters

- **name** (*str*) – of solid
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **pR1** (*float*, Constant, Quantity, Variable, Expression) – radius at -Dz/2
- **pR2** (*float*, Constant, Quantity, Variable, Expression) – radius at +Dz/2 (pR2 > pR1)
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.Plane

Module Contents

Classes

Plane	Constructs a *infinite* plane. Should not be used to construct geant4 geometry.

class pyg4ometry.geant4.solid.Plane.**Plane**(*name*, *normal*, *dist*, *zlength=10000*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a *infinite* plane. Should not be used to construct geant4 geometry.

Parameters

- **name** (*str*) – of object in registry
- **normal** (*tuple*) – normal [x,y,z]
- **dist** (*float*) – distance from origin to plane
- **zlength** (*float*) – large transverse box size to emulate infinite plane

__repr__()

pycsgmesh()

pyg4ometry.geant4.solid.Polycone

Module Contents

Classes

Polycone	Constructs a solid of rotation using an arbitrary 2D surface.

class pyg4ometry.geant4.solid.Polycone.**Polycone**(*name*, *pSPhi*, *pDPhi*, *pZpl*, *pRMin*, *pRMax*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface.

Parameters

- **name** (*str*) – of the solid
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – starting rotation angle in radians
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – total rotation angle in radius
- **pZPlns** (*list of float*, Constant, Quantity, Variable, Expression) – z-positions of planes used
- **pRInr** (*list of float*, Constant, Quantity, Variable, Expression) – inner radii of surface at each z-plane
- **pROut** (*list of float*, Constant, Quantity, Variable, Expression) – outer radii of surface at each z-plane
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

Optional registration as this solid is used as a temporary solid in Polyhedra and needn't be always registered.

__repr__()

__str__()

mesh()

pyg4ometry.geant4.solid.Polyhedra

Module Contents

Classes

Polyhedra	Constructs a polyhedra.

class pyg4ometry.geant4.solid.Polyhedra.**Polyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *numZPlanes*, *zPlane*, *rInner*, *rOuter*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a polyhedra.

Parameters

- **name** (*str*) – of solid
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – start phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – delta phi angle
- **numSide** (*int*) – number of sides
- **numZPlanes** (*int*) – number of planes along z
- **zPlane** (*list of float*, Constant, Quantity, Variable, Expression) – position of z planes
- **rInner** (*list of float*, Constant, Quantity, Variable, Expression) – tangent distance to inner surface per z plane
- **rOuter** (*list of float*, Constant, Quantity, Variable, Expression) – tangent distance to outer surface per z plane
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

__repr__()

__str__()

mesh()

pyg4ometry.geant4.solid.Scaled

Module Contents

Classes

Scaled	Constructs a scaled sold.

class pyg4ometry.geant4.solid.Scaled.**Scaled**(*name*, *solid*, *pX*, *pY*, *pZ*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a scaled sold.

Parameters

- **name** (*str*) – of solid for registry
- **pX** (*float*, Constant, Quantity, Variable, Expression) – scale in x
- **pY** (*float*, Constant, Quantity, Variable, Expression) – scale in y
- **pZ** (*float*, Constant, Quantity, Variable, Expression) – scale in z
- **registry** (Registry) – for storing solid

Poram solid
reference for scaling

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

pyg4ometry.geant4.solid.SolidBase

Module Contents

Classes

SolidBase	Base class for all solids

class pyg4ometry.geant4.solid.SolidBase.**SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

pyg4ometry.geant4.solid.Sphere

Module Contents

Classes

Sphere	Constructs a section of a spherical shell.

class pyg4ometry.geant4.solid.Sphere.**Sphere**(*name*, *pRmin*, *pRmax*, *pSPhi*, *pDPhi*, *pSTheta*, *pDTheta*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a section of a spherical shell.

Parameters

- **name** (*str*) – of object in registry
- **pRmin** (*float*, Constant, Quantity, Variable) – inner radius of the shell
- **pRmax** (*float*, Constant, Quantity, Variable) – outer radius of the shell
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle in radians
- **pSTheta** (*float*, Constant, Quantity, Variable) – starting theta angle in radians
- **pDPhi** (*float*, Constant, Quantity, Variable) – delta phi angle in radians
- **pDTheta** (*float*, Constant, Quantity, Variable) – delta theta angle in radians
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

checkParameters()

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

working off 0 < phi < 2pi 0 < theta < pi

pyg4ometry.geant4.solid.Subtraction

Module Contents

Classes

Subtraction	Subtraction between two solids

class pyg4ometry.geant4.solid.Subtraction.**Subtraction**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Subtraction between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra2
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]
- **registry** (Registry) – for storing solid

__repr__()

__str__()

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.TessellatedSolid

Module Contents

Classes

TessellatedSolid	Constructs a tessellated solid

Functions

createTessellatedSolid(name, polygons, reg)	Args:

class pyg4ometry.geant4.solid.TessellatedSolid.**TessellatedSolid**(*name*, *meshTess*, *registry*, *meshtype=MeshType.Freecad*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tessellated solid

Parameters

- **name** (*str*) – of solid
- **mesh** (*Freecad, Gdml or Stl*) – mesh
- **registry** (Registry) – for storing solid
- **meshtype** (*MeshType.Freecad*) – type of mesh

class MeshType

Freecad = 1

Gdml = 2

Stl = 3

__repr__()

Return repr(self).

__str__()

Return str(self).

addVertex(*vertex*)

addTriangle(*triangle*)

removeDuplicateVertices()

mesh()

pyg4ometry.geant4.solid.TessellatedSolid.**createTessellatedSolid**(*name*, *polygons*, *reg*)

Args:

name: Name of the tessallated solid polygons: list of polygons (list of points given in clockwise order). All polygons should have the same number of points. reg: registry

Returns: TessellatedSolid

pyg4ometry.geant4.solid.Tet

Module Contents

Classes

Tet	Constructs a tetrahedra.

class pyg4ometry.geant4.solid.Tet.**Tet**(*name*, *anchor*, *p2*, *p3*, *p4*, *registry*, *lunit='mm'*, *degeneracyFlag=False*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tetrahedra.

Parameters

- **name** – of the solid
- **anchor** (*list*) – point 1 (anchor point)
- **p2** (*list*) – point 2
- **p3** (*list*) – point 3
- **p4** (*list*) – point 4
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **degeneracyFlag** – bool, indicates degeneracy of points

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.Torus

Module Contents

Classes

Torus	Constructs a torus.

class pyg4ometry.geant4.solid.Torus.**Torus**(*name*, *pRmin*, *pRmax*, *pRtor*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a torus.

Parameters

- **name** (*str*) – string, name of the volume
- **pRmin** (*float*, Constant, Quantity, Variable, Expression) – innder radius
- **pRmax** – outer radius
- **pRtor** (*float*, Constant, Quantity, Variable, Expression) – swept radius of torus
- **pSphi** (*float*, Constant, Quantity, Variable, Expression) – start phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – delta phi angle
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

__str__()

mesh()

pyg4ometry.geant4.solid.Trap

Module Contents

Classes

Trap	Constructs a general trapezoid.

class pyg4ometry.geant4.solid.Trap.**Trap**(*name*, *pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pAlp1*, *pDy2*, *pDx3*, *pDx4*, *pAlp2*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a general trapezoid.

Parameters

- **name** (*str*) – name of the volume
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – half length along z
- **pTheta** (*float*, Constant, Quantity, Variable, Expression) – polar angle of the line joining the centres of the faces at -/+pDz
- **pPhi** (*float*, Constant, Quantity, Variable, Expression) – azimuthal angle of the line joining the centres of the faces at -/+pDz
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – half-length at -pDz

- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – half length along x of the side at y=-pDy1
- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – half length along x of the side at y=+pDy1
- **pAlp1** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axis from the centre of the side (lower endcap)
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – half-length at +pDz
- **pDx3** (*float*, Constant, Quantity, Variable, Expression) – half-length of the side at y=-pDy2 of the face at +pDz
- **pDx4** (*float*, Constant, Quantity, Variable, Expression) – half-length of the side at y=+pDy2 of the face at +pDz
- **pAlp2** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axis from the centre of the side (upper endcap)

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.Trd

Module Contents

Classes

Trd	Constructs a trapezoid.

class pyg4ometry.geant4.solid.Trd.**Trd**(*name*, *pDx1*, *pDx2*, *pDy1*, *pDy2*, *pDz*, *registry*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a trapezoid.

Parameters

- **name** (*str*) – of the solid
- **pDx1** (*float*, Constant, Quantity, Variable) – length along x at the surface positioned at -dz/2
- **pDx2** (*float*, Constant, Quantity, Variable) – length along x at the surface postitioned at +dz/2
- **pDy1** (*float*, Constant, Quantity, Variable) – length along y at the surface positioned at -dz/2
- **pDy2** (*float*, Constant, Quantity, Variable) – length along y at the surface positioned at +dz/2

- **dz** (*float*, Constant, Quantity, Variable) – length along the z axis
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

mesh()

pyg4ometry.geant4.solid.Tubs

Module Contents

Classes

Tubs	Constructs a cylindrical section.

class pyg4ometry.geant4.solid.Tubs.**Tubs**(*name*, *pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cylindrical section.

Parameters

- **name** (*str*) – of solid for registry
- **pRMin** (*float*, Constant, Quantity, Variable) – inner radius
- **pRMax** (*float*, Constant, Quantity, Variable) – outer radius
- **pDz** (*float*, Constant, Quantity, Variable) – full length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment in phi
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.TwistedBox

Module Contents

Classes

TwistedBox	Constructs a box that is twisted though angle twisted angle

class pyg4ometry.geant4.solid.TwistedBox.**TwistedBox**(*name*, *twistedangle*, *pDx*, *pDy*, *pDz*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *refine=0*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a box that is twisted though angle twisted angle

Parameters

- **name** (*str*) – of the solid
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle, must be less than pi/2
- **pDx** (*float*, Constant, Quantity, Variable, Expression) – length in x
- **pDy** (*float*, Constant, Quantity, Variable, Expression) – length in y
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **refine** (*int*) – number of steps to iteratively smoothen the mesh by doubling the number of vertices at every step
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nstack** (*int*) – Not written

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

makeLayers(*p1*, *p2*, *p3*, *p4*, *pDz*, *theta*, *nstack*)

mesh_old()

makeLayerEdge(*pDx*, *pDy*, *pTwistedAngle=0*, *nx=20*, *ny=20*)

mesh()

pyg4ometry.geant4.solid.TwistedSolid

Module Contents

Classes

TwistedSolid	

class pyg4ometry.geant4.solid.TwistedSolid.**TwistedSolid**

makeFaceFromLayer(*layer*, *reverse=False*)

makeSide(*pal*, *pbl*, *pau*, *pbu*, *zl*, *zu*, *nsl*)

p = point a = first b = second u = upper l = lower

meshFromLayers(*layers*, *nsl*)

pyg4ometry.geant4.solid.TwistedTrap

Module Contents

Classes

TwistedTrap	Constructs a general trapezoid with a twist around one axis.

class pyg4ometry.geant4.solid.TwistedTrap.**TwistedTrap**(*name*, *twistedAngle*, *pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pDy2*, *pDx3*, *pDx4*, *pAlp*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a general trapezoid with a twist around one axis.

Parameters

- **name** (*str*) – of the solid
- **twistedAngle** – angle of twist (<90 deg)
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – length along x of the side at y=-pDy1/2
- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – length along x of the side at y=+pDy1/2
- **pTheta** (*float*, Constant, Quantity, Variable, Expression) – polar angle of the line joining the centres of the faces at -/+pDz/2

- **pPhi** (*float*, Constant, Quantity, Variable, Expression) – azimuthal angle of the line joining the centres of the faces at -/+pDz/2
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – length at -pDz/2
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – length at +pDz/2
- **pDx3** (*float*, Constant, Quantity, Variable, Expression) – length of the side at y=-pDy2 of the face at +pDz/2
- **pDx4** (*float*, Constant, Quantity, Variable, Expression) – length of the side at y=+pDy2 of the face at +pDz/2
- **pAlp** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axi from the centre of the side
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

checkParameters()

__repr__()

Return repr(self).

__str__()

Return str(self).

makeLayers(*pl1*, *pl2*, *pl3*, *pl4*, *pu1*, *pu2*, *pu3*, *pu4*, *pDz*, *twist*, *theta*, *nsl*)

mesh()

pyg4ometry.geant4.solid.TwistedTrd

Module Contents

Classes

TwistedTrd	Constructs a twisted general trapezoid.

class pyg4ometry.geant4.solid.TwistedTrd.**TwistedTrd**(*name*, *twistedangle*, *pDx1*, *pDx2*, *pDy1*, *pDy2*, *pDz*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *refine=0*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a twisted general trapezoid.

Parameters

- **name** (*str*) – of solid
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle, must be less than 0.5*pi

- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – length in x at surface positioned at -pDz/2
- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – length in x at surface positioned at +pDz/2
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – length in y at surface positioned at -pDz/2
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – length in y at surface positioned at +pDz/2
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **refine** (*int*) – number of steps to iteratively smoothen the mesh by doubling the number of vertices at every step
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

makeLayers(*pl1*, *pl2*, *pl3*, *pl4*, *pu1*, *pu2*, *pu3*, *pu4*, *pDz*, *theta*, *nsl*)

mesh()

pyg4ometry.geant4.solid.TwistedTubs

Module Contents

Classes

TwistedTubs	Creates a twisted tube segement. Note that only 1 constructor is supprted.

class pyg4ometry.geant4.solid.TwistedTubs.**TwistedTubs**(*name*, *endinnerrad*, *endouterrad*, *zlen*, *phi*, *twistedangle*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Creates a twisted tube segement. Note that only 1 constructor is supprted.

Parameters

- **name** (*str*) – of solid

- **endinnerrad** (*float*, Constant, Quantity, Variable, Expression) – inner radius at the end of the segment
- **endinnerrad** – outer radius at the end of the segment
- **zlen** (*float*, Constant, Quantity, Variable, Expression) – length of the tube segement
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

makeLayers(*verts_bot*, *verts_top*)

mesh()

pyg4ometry.geant4.solid.TwoVector

Module Contents

Classes

TwoVector

class pyg4ometry.geant4.solid.TwoVector.**TwoVector**(*xIn*, *yIn*)

Rotated(*angle*)

__repr__()
Return repr(self).

__getitem__(*index*)

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__abs__()

dot(*other*)

cross(*other*)

pyg4ometry.geant4.solid.Union

Module Contents

Classes

Union	Union between two solids

class pyg4ometry.geant4.solid.Union.**Union**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Union between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra2
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]
- **registry** (Registry) – for storing solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.Wedge

Module Contents

Classes

Wedge	Constructs a *infinite* wedge. Should not be used to construct geant4 geometry.

class pyg4ometry.geant4.solid.Wedge.**Wedge**(*name*, *pRMax=1000*, *pSPhi=0*, *pDPhi=1.5*, *halfzlength=10000*, *nslice=None*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a *infinite* wedge. Should not be used to construct geant4 geometry.

Parameters

- **name** (*str*) – of object in registry
- **normal** (*tuple*) – normal [x,y,z]
- **dist** (*float*) – distance from origin to plane
- **zlength** (*float*) – large transverse box size to emulate infinite plane

__repr__()

pycsgmesh()

Package Contents

Classes

TwoVector	
_SolidBase	Base class for all solids
Plane	Constructs a *infinite* plane. Should not be used to construct geant4 geometry.
_SolidBase	Base class for all solids
Wedge	Constructs a *infinite* wedge. Should not be used to construct geant4 geometry.
_SolidBase	Base class for all solids
Box	
_SolidBase	Base class for all solids
Tubs	
_SolidBase	Base class for all solids
CutTubs	Constructs a cylindrical section with end face cuts. Note pLowNorm and pHighNorm can be
_SolidBase	Base class for all solids

continues on next page

Table 3 – continued from previous page

Sphere	Constructs a section of a spherical shell.
_SolidBase	Base class for all solids
Cons	
_SolidBase	Base class for all solids
Ellipsoid	Constructs an ellipsoid optinoally cut by planes perpendicular to the z-axis.
_SolidBase	Base class for all solids
Trd	
_SolidBase	Base class for all solids
Torus	Constructs a torus.
_SolidBase	Base class for all solids
_GenericPolyhedra	Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.
Polycone	Constructs a solid of rotation using an arbitrary 2D surface.
_GenericPolyhedra	Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.
_SolidBase	Base class for all solids
Polyhedra	Constructs a polyhedra.
_SolidBase	Base class for all solids
ExtrudedSolid	Construct an extruded solid
_SolidBase	Base class for all solids
Union	
_SolidBase	Base class for all solids
Intersection	
_SolidBase	Base class for all solids
Subtraction	Subtraction between two solids
_SolidBase	Base class for all solids
OpticalSurface	
_SolidBase	Base class for all solids
Para	Constructs a parallelepiped.
_SolidBase	Base class for all solids
Trap	Constructs a general trapezoid.
_SolidBase	Base class for all solids
Orb	
_SolidBase	Base class for all solids
EllipticalTube	Constructs a tube of elliptical cross-section.
_SolidBase	Base class for all solids
EllipticalCone	
_SolidBase	Base class for all solids
Paraboloid	Constructs a paraboloid with possible cuts along the z axis.
_SolidBase	Base class for all solids
Hype	Constructs a tube with hyperbolic profile.
_SolidBase	Base class for all solids

continues on next page

Table 3 – continued from previous page

Tet	
_SolidBase	Base class for all solids
_TwistedSolid	
_TwoVector	
_Layer	
TwistedBox	
_SolidBase	Base class for all solids
_TwistedSolid	
_TwoVector	
_Layer	
TwistedTrap	
_SolidBase	Base class for all solids
_TwoVector	
_Layer	
_TwistedSolid	
TwistedTrd	
_SolidBase	Base class for all solids
TwistedTubs	
_SolidBase	Base class for all solids
_GenericPolyhedra	Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.
GenericPolycone	
_SolidBase	Base class for all solids
GenericPolyhedra	
_SolidBase	Base class for all solids
GenericTrap	Constructs an arbitrary trapezoid using two quadrilaterals sitting
_SolidBase	Base class for all solids
TessellatedSolid	
_SolidBase	Base class for all solids
MultiUnion	
_SolidBase	Base class for all solids
Scaled	

continues on next page

Table 3 – continued from previous page

SolidBase	Base class for all solids

Functions

cubeNet(vecList)	
rad2deg(rad)	Convert rad in radians into degrees
deg2rad(deg)	Convert deg in degrees into radians
grad2rad(gradians)	Convert rad in gradians into radians
tbxyz2axisangle(rv)	Tait-Bryan x-y-z rotation to axis-angle representation
matrix2axisangle(matrix)	Convert 3x3 transformation matrix to axis angle representation
axisangle2matrix(axis, angle)	Convert axis angle to transformation matrix
matrix2tbxyz(matrix)	Convert rotation matrix to Tait-Bryan angles.
axisangle2tbxyz(axis, angle)	Convert axis and angle to tait bryan angles
tbxyz2matrix(angles)	Convert tait bryan angles to a single passive rotation matrix.
tbzyx2matrix(angles)	Convert Tait-Bryan angles to a single passive rotation matrix.
matrix_from(v_from, v_to)	Returns the rotation matrix that rotates v_from to parallel to
_rodrigues_anti_parallel(v_from, v_to)	v_from = vector FROM
are_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
are_anti_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
reverse(angles)	Invert the rotation represented by these angles.
two_fold_orientation(v1, v2, e1, e2)	matrix_from will align one vector with another, but there are
rad2deg(rad)	Convert rad in radians into degrees
deg2rad(deg)	Convert deg in degrees into radians
grad2rad(gradians)	Convert rad in gradians into radians
tbxyz2axisangle(rv)	Tait-Bryan x-y-z rotation to axis-angle representation
matrix2axisangle(matrix)	Convert 3x3 transformation matrix to axis angle representation
axisangle2matrix(axis, angle)	Convert axis angle to transformation matrix
matrix2tbxyz(matrix)	Convert rotation matrix to Tait-Bryan angles.
axisangle2tbxyz(axis, angle)	Convert axis and angle to tait bryan angles
tbxyz2matrix(angles)	Convert tait bryan angles to a single passive rotation matrix.
tbzyx2matrix(angles)	Convert Tait-Bryan angles to a single passive rotation matrix.
matrix_from(v_from, v_to)	Returns the rotation matrix that rotates v_from to parallel to
_rodrigues_anti_parallel(v_from, v_to)	v_from = vector FROM
are_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
are_anti_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to

continues on next page

Table 4 – continued from previous page

reverse(angles)	Invert the rotation represented by these angles.
two_fold_orientation(v1, v2, e1, e2)	matrix_from will align one vector with another, but there are
rad2deg(rad)	Convert rad in radians into degrees
deg2rad(deg)	Convert deg in degrees into radians
grad2rad(gradians)	Convert rad in gradians into radians
tbxyz2axisangle(rv)	Tait-Bryan x-y-z rotation to axis-angle representation
matrix2axisangle(matrix)	Convert 3x3 transformation matrix to axis angle representation
axisangle2matrix(axis, angle)	Convert axis angle to transformation matrix
matrix2tbxyz(matrix)	Convert rotation matrix to Tait-Bryan angles.
axisangle2tbxyz(axis, angle)	Convert axis and angle to tait bryan angles
tbxyz2matrix(angles)	Convert tait bryan angles to a single passive rotation matrix.
tbzyx2matrix(angles)	Convert Tait-Bryan angles to a single passive rotation matrix.
matrix_from(v_from, v_to)	Returns the rotation matrix that rotates v_from to parallel to
_rodrigues_anti_parallel(v_from, v_to)	v_from = vector FROM
are_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
are_anti_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
reverse(angles)	Invert the rotation represented by these angles.
two_fold_orientation(v1, v2, e1, e2)	matrix_from will align one vector with another, but there are
_cubeNet(vecList)	
_cubeNet(vecList)	
createTessellatedSolid(name, polygons, reg)	Args:
rad2deg(rad)	Convert rad in radians into degrees
deg2rad(deg)	Convert deg in degrees into radians
grad2rad(gradians)	Convert rad in gradians into radians
tbxyz2axisangle(rv)	Tait-Bryan x-y-z rotation to axis-angle representation
matrix2axisangle(matrix)	Convert 3x3 transformation matrix to axis angle representation
axisangle2matrix(axis, angle)	Convert axis angle to transformation matrix
matrix2tbxyz(matrix)	Convert rotation matrix to Tait-Bryan angles.
axisangle2tbxyz(axis, angle)	Convert axis and angle to tait bryan angles
tbxyz2matrix(angles)	Convert tait bryan angles to a single passive rotation matrix.
tbzyx2matrix(angles)	Convert Tait-Bryan angles to a single passive rotation matrix.
matrix_from(v_from, v_to)	Returns the rotation matrix that rotates v_from to parallel to
_rodrigues_anti_parallel(v_from, v_to)	v_from = vector FROM
are_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
are_anti_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
reverse(angles)	Invert the rotation represented by these angles.

continues on next page

Table 4 – continued from previous page

two_fold_orientation(v1, v2, e1, e2)	matrix_from will align one vector with another, but there are

class pyg4ometry.geant4.solid.**TwoVector**(*xIn*, *yIn*)

Rotated(*angle*)

__repr__()

Return repr(self).

__getitem__(*index*)

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__abs__()

dot(*other*)

cross(*other*)

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Plane**(*name*, *normal*, *dist*, *zlength=10000*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a *infinite* plane. Should not be used to construct geant4 geometry.

Parameters

- **name** (*str*) – of object in registry
- **normal** (*tuple*) – normal [x,y,z]

- **dist** (*float*) – distance from origin to plane
- **zlength** (*float*) – large transverse box size to emulate infinite plane

__repr__()

pycsgmesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Wedge**(*name*, *pRMax=1000*, *pSPhi=0*, *pDPhi=1.5*, *halfzlength=10000*, *nslice=None*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a *infinite* wedge. Should not be used to construct geant4 geometry.

Parameters

- **name** (*str*) – of object in registry
- **normal** (*tuple*) – normal [x,y,z]
- **dist** (*float*) – distance from origin to plane
- **zlength** (*float*) – large transverse box size to emulate infinite plane

__repr__()

pycsgmesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

pyg4ometry.geant4.solid.**cubeNet**(*vecList*)

class pyg4ometry.geant4.solid.**Box**(*name*, *pX*, *pY*, *pZ*, *registry*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a box. Note the lengths are full lengths and not half lengths as in Geant4

Parameters

- **name** (*str*) – of solid for registry
- **pX** (*float*, Constant, Quantity, Variable, Expression) – length along x
- **pY** (*float*, Constant, Quantity, Variable, Expression) – length along y
- **pZ** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Tubs**(*name*, *pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cylindrical section.

Parameters

- **name** (*str*) – of solid for registry
- **pRMin** (*float*, Constant, Quantity, Variable) – inner radius
- **pRMax** (*float*, Constant, Quantity, Variable) – outer radius
- **pDz** (*float*, Constant, Quantity, Variable) – full length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment in phi
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**CutTubs**(*name*, *pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *pLowNorm*, *pHighNorm*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cylindrical section with end face cuts. Note pLowNorm and pHighNorm can be lists of floats, Constants, Quantities or Variables.

Parameters

- **name** (*str*) – of solid for registry
- **pRMin** (*float*, Constant, Quantity, Variable) – Inner radius

- **pRMax** (*float*, Constant, Quantity, Variable) – Outer radius
- **pDz** (*float*, Constant, Quantity, Variable) – length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment
- **pLowNorm** (*list*) – normal vector of the cut plane at -pDz/2
- **pHighNorm** (*list*) – normal vector of the cut plane at +pDz/2

__repr__()

__str__()

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Sphere**(*name*, *pRmin*, *pRmax*, *pSPhi*, *pDPhi*, *pSTheta*, *pDTheta*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a section of a spherical shell.

Parameters

- **name** (*str*) – of object in registry
- **pRmin** (*float*, Constant, Quantity, Variable) – inner radius of the shell
- **pRmax** (*float*, Constant, Quantity, Variable) – outer radius of the shell
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle in radians
- **pSTheta** (*float*, Constant, Quantity, Variable) – starting theta angle in radians
- **pDPhi** (*float*, Constant, Quantity, Variable) – delta phi angle in radians
- **pDTheta** (*float*, Constant, Quantity, Variable) – delta theta angle in radians
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

checkParameters()

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()
working off 0 < phi < 2pi 0 < theta < pi

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Cons**(*name*, *pRmin1*, *pRmax1*, *pRmin2*, *pRmax2*, *pDz*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)
Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a conical section.

Parameters

- **name** (*str*) – of the solid
- **pRMin1** (*float*, Constant, Quantity, Variable) – inner radius at -pDz/2
- **pRMax1** (*float*, Constant, Quantity, Variable) – outer radius at -pDz/2
- **pRMin2** (*float*, Constant, Quantity, Variable) – inner radius at +pDZ/2
- **pRMax2** (*float*, Constant, Quantity, Variable) – outer radius at +pDz/2
- **pDz** (*float*, Constant, Quantity, Variable) – length along z
- **pSPhi** (*float*, Constant, Quantity, Variable) – starting phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable) – angle of segment in radians
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

checkParameters()

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Ellipsoid**(*name*, *pxSemiAxis*, *pySemiAxis*, *pzSemiAxis*, *pzBottomCut*, *pzTopCut*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)
Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs an ellipsoid optinoally cut by planes perpendicular to the z-axis.

Parameters

- **name** (*str*) – of the solid
- **pxSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of x semi axis
- **pySemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of y semi axis
- **pzSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – length of z semi axis
- **pzBottomCut** (*float*, Constant, Quantity, Variable, Expression) – z-position of bottom cut plane
- **pzTopCut** (*float*, Constant, Quantity, Variable, Expression) – z-position of top cut plane
- **registry** (Registry) – for storing solid

- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Trd**(*name*, *pDx1*, *pDx2*, *pDy1*, *pDy2*, *pDz*, *registry*, *lunit='mm'*, *addRegistry=True*)
Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*
Constructs a trapezoid.

Parameters

- **name** (*str*) – of the solid
- **pDx1** (*float*, Constant, Quantity, Variable) – length along x at the surface positioned at -dz/2
- **pDx2** (*float*, Constant, Quantity, Variable) – length along x at the surface postitioned at +dz/2
- **pDy1** (*float*, Constant, Quantity, Variable) – length along y at the surface positioned at -dz/2
- **pDy2** (*float*, Constant, Quantity, Variable) – length along y at the surface positioned at +dz/2
- **dz** (*float*, Constant, Quantity, Variable) – length along the z axis
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Torus**(*name*, *pRmin*, *pRmax*, *pRtor*, *pSPhi*, *pDPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a torus.

Parameters

- **name** (*str*) – string, name of the volume
- **pRmin** (*float*, Constant, Quantity, Variable, Expression) – innder radius
- **pRmax** – outer radius
- **pRtor** (*float*, Constant, Quantity, Variable, Expression) – swept radius of torus
- **pSphi** (*float*, Constant, Quantity, Variable, Expression) – start phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – delta phi angle
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

__str__()

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**_GenericPolyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – name
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **numSide** (*float*, Constant, Quantity, Variable, Expression) – number of polygon sides
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate list
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate list

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid.**Polycone**(*name*, *pSPhi*, *pDPhi*, *pZpl*, *pRMin*, *pRMax*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface.

Parameters

- **name** (*str*) – of the solid

- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – starting rotation angle in radians
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – total rotation angle in radius
- **pZPlns** (*list of float*, Constant, Quantity, Variable, Expression) – z-positions of planes used
- **pRInr** (*list of float*, Constant, Quantity, Variable, Expression) – inner radii of surface at each z-plane
- **pROut** (*list of float*, Constant, Quantity, Variable, Expression) – outer radii of surface at each z-plane
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

Optional registration as this solid is used as a temporary solid in Polyhedra and needn't be always registered.

__repr__()

__str__()

mesh()

class pyg4ometry.geant4.solid.**_GenericPolyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – name
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **numSide** (*float*, Constant, Quantity, Variable, Expression) – number of polygon sides
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate list
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate list

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Polyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *numZPlanes*, *zPlane*, *rInner*, *rOuter*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a polyhedra.

Parameters

- **name** (*str*) – of solid
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – start phi angle
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – delta phi angle
- **numSide** (*int*) – number of sides
- **numZPlanes** (*int*) – number of planes along z
- **zPlane** (*list of float*, Constant, Quantity, Variable, Expression) – position of z planes
- **rInner** (*list of float*, Constant, Quantity, Variable, Expression) – tangent distance to inner surface per z plane
- **rOuter** (*list of float*, Constant, Quantity, Variable, Expression) – tangent distance to outer surface per z plane
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

__repr__()

__str__()

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.ExtrudedSolid(*name*, *pPolygon*, *pZslices*, *registry*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Construct an extruded solid

Parameters

- **name** (*str*) – of solid
- **pPolygon** (*list of lists*) – x-y coordinates of vertices for the polygon.
- **pZslices** (*list of lists*) – z-coordinates of a slice, slice offsets in x-y and scaling
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid

Example: Triangular prism with 2 slices pPoligon = [[x1,y1],[x2,y2],[x3,y3]] - vertices of polygon in clockwise order zSlices = [[z1,[offsx1, offsy1],scale1],[z2,[offsx2, offsy2],scale2]]

__repr__()

Return repr(self).

__str__()

Return str(self).

polygon_area(*vertices*)

evaluateParameterWithUnits(*varName*)

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Union**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Union between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra2
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]
- **registry** (Registry) – for storing solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.**rad2deg**(*rad*)

Convert rad in radians into degrees

Parameters
rad (*float*) – Input in radians

Returns
rad in degrees

Return type
float

pyg4ometry.geant4.solid.**deg2rad**(*deg*)

Convert deg in degrees into radians

Parameters
deg (*float*) – Input in degrees

Returns
deg in radians

Return type
float

pyg4ometry.geant4.solid.**grad2rad**(*gradians*)

Convert rad in gradians into radians

Parameters
gradians (*float*) – Input in gradians

Returns
gradians in radians

Return type
float

pyg4ometry.geant4.solid.**tbxyz2axisangle**(*rv*)

Tait-Bryan x-y-z rotation to axis-angle representation Algorithm from http://www.sedris.org/wg8home/Documents/WG80485.pdf

For converting rotation angles to an active axis/angle pair for use in pycsg. Order of rotation: x->y->z.

Parameters
rv (*float(3)*) – rotation angles

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**matrix2axisangle**(*matrix*)

Convert 3x3 transformation matrix to axis angle representation

Parameters
matrix (*array(3,3)*) – 3x3 rotation matrix array

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**axisangle2matrix**(*axis*, *angle*)

Convert axis angle to transformation matrix

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
transformation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix2tbxyz**(*matrix*)

Convert rotation matrix to Tait-Bryan angles. Order of rotation is x -> y -> z.

Parameters
matrix – active (positive angle = anti-clockwise rotation about that axis when looking at the axis) matrix.

Returns
[x, y, z] Tait-Bryan angles in a list.

Return type
list(3)

pyg4ometry.geant4.solid.**axisangle2tbxyz**(*axis*, *angle*)

Convert axis and angle to tait bryan angles

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
tait bryan angles (x-y-z)

Return type
array(3)

pyg4ometry.geant4.solid.**tbxyz2matrix**(*angles*)

Convert tait bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**tbzyx2matrix**(*angles*)

Convert Tait-Bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix_from**(*v_from*, *v_to*)

Returns the rotation matrix that rotates v_from to parallel to v_to.

Useful for ensuring a given face points in a certain direction.

v_from and v_to should be array-like three-vectors.

pyg4ometry.geant4.solid.**_rodrigues_anti_parallel**(*v_from*, *v_to*)

v_from = vector FROM v_to = vector TO

source: http://en.citizendium.org/wiki/Rotation_matrix#Case_that_.22from.22_and_.22to.22_vectors_are_anti-parallel

pyg4ometry.geant4.solid.**are_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are parallel

Return type
bool

pyg4ometry.geant4.solid.**are_anti_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are antiparallel

Return type
bool

pyg4ometry.geant4.solid.**reverse**(*angles*)

Invert the rotation represented by these angles.

pyg4ometry.geant4.solid.**two_fold_orientation**(*v1*, *v2*, *e1*, *e2*)

matrix_from will align one vector with another, but there are an infinite number of such matrices that align two vectors. This further contrains the rotation by introducing a second pair of vectors.

v1 start v v2 end v e1 start e e2 end v

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Intersection**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Intersection between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]
- **registry** (Registry) – for storing solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.**rad2deg**(*rad*)

Convert rad in radians into degrees

Parameters
rad (*float*) – Input in radians

Returns
rad in degrees

Return type
float

pyg4ometry.geant4.solid.**deg2rad**(*deg*)

Convert deg in degrees into radians

Parameters
deg (*float*) – Input in degrees

Returns
deg in radians

Return type
float

pyg4ometry.geant4.solid.**grad2rad**(*gradians*)

Convert rad in gradians into radians

Parameters
gradians (*float*) – Input in gradians

Returns
gradians in radians

Return type
float

pyg4ometry.geant4.solid.**tbxyz2axisangle**(*rv*)

Tait-Bryan x-y-z rotation to axis-angle representation Algorithm from http://www.sedris.org/wg8home/Documents/WG80485.pdf

For converting rotation angles to an active axis/angle pair for use in pycsg. Order of rotation: x->y->z.

Parameters
rv (*float(3)*) – rotation angles

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**matrix2axisangle**(*matrix*)

Convert 3x3 transformation matrix to axis angle representation

Parameters
matrix (*array(3,3)*) – 3x3 rotation matrix array

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**axisangle2matrix**(*axis*, *angle*)

Convert axis angle to transformation matrix

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
transformation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix2tbxyz**(*matrix*)

Convert rotation matrix to Tait-Bryan angles. Order of rotation is x -> y -> z.

Parameters
matrix – active (positive angle = anti-clockwise rotation about that axis when looking at the axis) matrix.

Returns
[x, y, z] Tait-Bryan angles in a list.

Return type
list(3)

pyg4ometry.geant4.solid.**axisangle2tbxyz**(*axis*, *angle*)

Convert axis and angle to tait bryan angles

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
tait bryan angles (x-y-z)

Return type
array(3)

pyg4ometry.geant4.solid.**tbxyz2matrix**(*angles*)

Convert tait bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**tbzyx2matrix**(*angles*)

Convert Tait-Bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix_from**(*v_from*, *v_to*)

Returns the rotation matrix that rotates v_from to parallel to v_to.

Useful for ensuring a given face points in a certain direction.

v_from and v_to should be array-like three-vectors.

pyg4ometry.geant4.solid.**_rodrigues_anti_parallel**(*v_from*, *v_to*)

v_from = vector FROM v_to = vector TO

source: http://en.citizendium.org/wiki/Rotation_matrix#Case_that_.22from.22_and_.22to.22_vectors_are_anti-parallel

pyg4ometry.geant4.solid.**are_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are parallel

Return type
bool

pyg4ometry.geant4.solid.**are_anti_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are antiparallel

Return type
bool

pyg4ometry.geant4.solid.**reverse**(*angles*)

Invert the rotation represented by these angles.

pyg4ometry.geant4.solid.**two_fold_orientation**(*v1*, *v2*, *e1*, *e2*)

matrix_from will align one vector with another, but there are an infinite number of such matrices that align two vectors. This further contrains the rotation by introducing a second pair of vectors.

v1 start v v2 end v e1 start e e2 end v

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Subtraction**(*name*, *obj1*, *obj2*, *tra2*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Subtraction between two solids

Parameters

- **name** (*str*) – of solid
- **obj1** (*pyg4ometry.geant4.solid*) – unrotated, untranslated solid
- **obj2** (*pyg4ometry.geant4.solid*) – solid rotated and translated according to tra2
- **tra2** (*list*) – [rot,tra] = [[a,b,g],[dx,dy,dz]]

- **registry** (Registry) – for storing solid

__repr__()

__str__()

mesh()

translation()

rotation()

object1()

object2()

pyg4ometry.geant4.solid.**rad2deg**(*rad*)

Convert rad in radians into degrees

Parameters
rad (*float*) – Input in radians

Returns
rad in degrees

Return type
float

pyg4ometry.geant4.solid.**deg2rad**(*deg*)

Convert deg in degrees into radians

Parameters
deg (*float*) – Input in degrees

Returns
deg in radians

Return type
float

pyg4ometry.geant4.solid.**grad2rad**(*gradians*)

Convert rad in gradians into radians

Parameters
gradians (*float*) – Input in gradians

Returns
gradians in radians

Return type
float

pyg4ometry.geant4.solid.**tbxyz2axisangle**(*rv*)

Tait-Bryan x-y-z rotation to axis-angle representation Algorithm from http://www.sedris.org/wg8home/Documents/WG80485.pdf

For converting rotation angles to an active axis/angle pair for use in pycsg. Order of rotation: x->y->z.

Parameters
rv (*float(3)*) – rotation angles

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**matrix2axisangle**(*matrix*)

Convert 3x3 transformation matrix to axis angle representation

Parameters
matrix (*array(3,3)*) – 3x3 rotation matrix array

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**axisangle2matrix**(*axis*, *angle*)

Convert axis angle to transformation matrix

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
transformation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix2tbxyz**(*matrix*)

Convert rotation matrix to Tait-Bryan angles. Order of rotation is x -> y -> z.

Parameters
matrix – active (positive angle = anti-clockwise rotation about that axis when looking at the axis) matrix.

Returns
[x, y, z] Tait-Bryan angles in a list.

Return type
list(3)

pyg4ometry.geant4.solid.**axisangle2tbxyz**(*axis*, *angle*)

Convert axis and angle to tait bryan angles

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
tait bryan angles (x-y-z)

Return type
array(3)

pyg4ometry.geant4.solid.**tbxyz2matrix**(*angles*)

Convert tait bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**tbzyx2matrix**(*angles*)

Convert Tait-Bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix_from**(*v_from*, *v_to*)

Returns the rotation matrix that rotates v_from to parallel to v_to.

Useful for ensuring a given face points in a certain direction.

v_from and v_to should be array-like three-vectors.

pyg4ometry.geant4.solid.**_rodrigues_anti_parallel**(*v_from*, *v_to*)

v_from = vector FROM v_to = vector TO

source: http://en.citizendium.org/wiki/Rotation_matrix#Case_that_.22from.22_and_.22to.22_vectors_are_anti-parallel

pyg4ometry.geant4.solid.**are_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are parallel

Return type
bool

pyg4ometry.geant4.solid.**are_anti_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are antiparallel

Return type
bool

pyg4ometry.geant4.solid.**reverse**(*angles*)

Invert the rotation represented by these angles.

pyg4ometry.geant4.solid.**two_fold_orientation**(*v1*, *v2*, *e1*, *e2*)

matrix_from will align one vector with another, but there are an infinite number of such matrices that align two vectors. This further contrains the rotation by introducing a second pair of vectors.

v1 start v v2 end v e1 start e e2 end v

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**OpticalSurface**(*name*, *finish*, *model*, *surf_type*, *value*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

allowed_models = ['glisur', 'unified', 'LUT', 'DAVIS', 'dichroic']

allowed_types = ['dielectric_metal', 'dielectric_dielectric', 'dielectric_LUT', 'dielectric_LUTDAVIS',...

allowed_finishes = ['polished', 'polishedfrontpainted', 'polishedbackpainted', 'ground', 'groundfrontpainted',...

__repr__()

__str__()

addProperty(*name*, *matrix*)

Add a property to this surface from a matrix.

Parameters

- **name** (*str*) – key of the surface property
- **matrix** (*Matrix*) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property

- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str,float,int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

pyg4ometry.geant4.solid.**_cubeNet**(*vecList*)

class pyg4ometry.geant4.solid.**Para**(*name*, *pDx*, *pDy*, *pDz*, *pAlpha*, *pTheta*, *pPhi*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a parallelepiped.

Parameters

- **name** (*str*) – of the volume
- **pX** (*float, Constant, Quantity, Variable*) – length along x
- **pY** (*float, Constant, Quantity, Variable*) – length along y
- **pZ** (*float, Constant, Quantity, Variable*) – length along z
- **pAlpha** (*float, Constant, Quantity, Variable*) – angle formed by the y axis and the plane joining the centres of the faces parallel tothe z-x plane at -dy/2 and +dy/2
- **pTheta** (*float, Constant, Quantity, Variable*) – polar angle of the line joining the centres of the faces at -dz/2 and +dz/2 in z

- **pPhi** (*float*, Constant, Quantity, Variable) – azimuthal angle of the line joining the centres of the faces at -dx/2 and +dz/2 in z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

pyg4ometry.geant4.solid.**_cubeNet**(*vecList*)

class pyg4ometry.geant4.solid.**Trap**(*name*, *pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pAlp1*, *pDy2*, *pDx3*, *pDx4*, *pAlp2*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a general trapezoid.

Parameters

- **name** (*str*) – name of the volume
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – half length along z
- **pTheta** (*float*, Constant, Quantity, Variable, Expression) – polar angle of the line joining the centres of the faces at -/+pDz
- **pPhi** (*float*, Constant, Quantity, Variable, Expression) – azimuthal angle of the line joining the centres of the faces at -/+pDz
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – half-length at -pDz
- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – half length along x of the side at y=-pDy1

- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – half length along x of the side at y=+pDy1
- **pAlp1** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axis from the centre of the side (lower endcap)
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – half-length at +pDz
- **pDx3** (*float*, Constant, Quantity, Variable, Expression) – half-length of the side at y=-pDy2 of the face at +pDz
- **pDx4** (*float*, Constant, Quantity, Variable, Expression) – half-length of the side at y=+pDy2 of the face at +pDz
- **pAlp2** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axis from the centre of the side (upper endcap)

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Orb**(*name*, *pRMax*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid sphere.

Parameters

- **name** (*str*) – of the sold
- **pRMax** (*float*, Constant, Quantity, Variable, Expression) – outer radius
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing

- **nstack** (*int*) – number of theta elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**EllipticalTube**(*name*, *pDx*, *pDy*, *pDz*, *registry*, *lunit='mm'*, *nstack=None*, *nslice=None*, *addRegistry=True*)
Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*
Constructs a tube of elliptical cross-section.

Parameters

- **name** (*str*) – name of the solid
- **pDx** (*float*, Constant, Quantity, Variable, Expression) – length in x
- **pDy** (*float*, Constant, Quantity, Variable, Expression) – length in y
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

__str__()

mesh()
new meshing based of Tubs meshing

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.EllipticalCone(*name*, *pxSemiAxis*, *pySemiAxis*, *zMax*, *pzTopCut*, *registry*, *lunit='mm'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a cone with elliptical cross-section and an optional cut. Both zMax and pzTopCut are half lengths extending from the centre of the cone, at z=0.

Parameters

- **name** (*str*) – name of the volume
- **pxSemiAxis** (*float*, Constant, Quantity, Variable, Expression) – semiaxis in x at z=0 as a fraction of zMax.
- **pySemiAxis** (*float*, Constant, Quantity, Variable, Expression) – semiaxis in y at z=0 as a fraction of zMax
- **zMax** (*float*, Constant, Quantity, Variable, Expression) – half length of the cone.
- **pzTopCut** (*float*, Constant, Quantity, Variable, Expression) – half length of the cut.

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Paraboloid**(*name*, *pDz*, *pR1*, *pR2*, *registry*, *lunit='mm'*, *nslice=16*, *nstack=8*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a paraboloid with possible cuts along the z axis.

Parameters

- **name** (*str*) – of solid
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **pR1** (*float*, Constant, Quantity, Variable, Expression) – radius at -Dz/2
- **pR2** (*float*, Constant, Quantity, Variable, Expression) – radius at +Dz/2 (pR2 > pR1)
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Hype**(*name*, *innerRadius*, *outerRadius*, *innerStereo*, *outerStereo*, *lenZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tube with hyperbolic profile.

Parameters

- **name** (*str*) – of solid
- **innerRadius** (*float*, Constant, Quantity, Variable, Expression) – inner radius
- **outerRadius** (*float*, Constant, Quantity, Variable, Expression) – outer radius
- **innerStereo** (*float*, Constant, Quantity, Variable, Expression) – inner stereo angle
- **outerStereo** (*float*, Constant, Quantity, Variable, Expression) – outer stereo angle
- **lenZ** – length along z
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing
- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.Tet(*name*, *anchor*, *p2*, *p3*, *p4*, *registry*, *lunit='mm'*, *degeneracyFlag=False*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tetrahedra.

Parameters

- **name** – of the solid
- **anchor** (*list*) – point 1 (anchor point)
- **p2** (*list*) – point 2
- **p3** (*list*) – point 3
- **p4** (*list*) – point 4
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **degeneracyFlag** – bool, indicates degeneracy of points

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid._TwistedSolid

makeFaceFromLayer(*layer*, *reverse=False*)

makeSide(*pal*, *pbl*, *pau*, *pbu*, *zl*, *zu*, *nsl*)

p = point a = first b = second u = upper l = lower

meshFromLayers(*layers*, *nsl*)

class pyg4ometry.geant4.solid.**_TwoVector**(*xIn*, *yIn*)

Rotated(*angle*)

__repr__()

Return repr(self).

__getitem__(*index*)

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__abs__()

dot(*other*)

cross(*other*)

class pyg4ometry.geant4.solid.**_Layer**(*p1*, *p2*, *p3*, *p4*, *z*)

__getitem__(*index*)

Rotated(*angle*)

__repr__()

Return repr(self).

class pyg4ometry.geant4.solid.**TwistedBox**(*name*, *twistedangle*, *pDx*, *pDy*, *pDz*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *refine=0*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a box that is twisted though angle twisted angle

Parameters

- **name** (*str*) – of the solid
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle, must be less than pi/2
- **pDx** (*float*, Constant, Quantity, Variable, Expression) – length in x
- **pDy** (*float*, Constant, Quantity, Variable, Expression) – length in y
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **refine** (*int*) – number of steps to iteratively smoothen the mesh by doubling the number of vertices at every step
- **registry** (Registry) – for storing solid

- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nstack** (*int*) – Not written

__repr__()
Return repr(self).

__str__()
Return str(self).

checkParameters()

makeLayers(*p1*, *p2*, *p3*, *p4*, *pDz*, *theta*, *nstack*)

mesh_old()

makeLayerEdge(*pDx*, *pDy*, *pTwistedAngle=0*, *nx=20*, *ny=20*)

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**_TwistedSolid**

makeFaceFromLayer(*layer*, *reverse=False*)

makeSide(*pal*, *pbl*, *pau*, *pbu*, *zl*, *zu*, *nsl*)
p = point a = first b = second u = upper l = lower

meshFromLayers(*layers*, *nsl*)

class pyg4ometry.geant4.solid.**_TwoVector**(*xIn*, *yIn*)

Rotated(*angle*)

__repr__()
Return repr(self).

__getitem__(*index*)

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__abs__()

dot(*other*)

cross(*other*)

class pyg4ometry.geant4.solid._Layer(*p1*, *p2*, *p3*, *p4*, *z*)

__getitem__(*index*)

Rotated(*angle*)

__repr__()

Return repr(self).

class pyg4ometry.geant4.solid.TwistedTrap(*name*, *twistedAngle*, *pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pDy2*, *pDx3*, *pDx4*, *pAlp*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a general trapezoid with a twist around one axis.

Parameters

- **name** (*str*) – of the solid
- **twistedAngle** – angle of twist (<90 deg)
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length along z
- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – length along x of the side at y=-pDy1/2
- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – length along x of the side at y=+pDy1/2
- **pTheta** (*float*, Constant, Quantity, Variable, Expression) – polar angle of the line joining the centres of the faces at -/+pDz/2
- **pPhi** (*float*, Constant, Quantity, Variable, Expression) – azimuthal angle of the line joining the centres of the faces at -/+pDz/2
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – length at -pDz/2
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – length at +pDz/2
- **pDx3** (*float*, Constant, Quantity, Variable, Expression) – length of the side at y=-pDy2 of the face at +pDz/2
- **pDx4** (*float*, Constant, Quantity, Variable, Expression) – length of the side at y=+pDy2 of the face at +pDz/2

- **pAlp** (*float*, Constant, Quantity, Variable, Expression) – angle wrt the y axi from the centre of the side
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid

checkParameters()

__repr__()
Return repr(self).

__str__()
Return str(self).

makeLayers(*pl1*, *pl2*, *pl3*, *pl4*, *pu1*, *pu2*, *pu3*, *pu4*, *pDz*, *twist*, *theta*, *nsl*)

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**_TwoVector**(*xIn*, *yIn*)

Rotated(*angle*)

__repr__()
Return repr(self).

__getitem__(*index*)

__add__(*other*)

__sub__(*other*)

__mul__(*other*)

__rmul__(*other*)

__truediv__(*other*)

__abs__()

dot(*other*)

cross(*other*)

class pyg4ometry.geant4.solid._Layer(*p1*, *p2*, *p3*, *p4*, *z*)

__getitem__(*index*)

Rotated(*angle*)

__repr__()

Return repr(self).

class pyg4ometry.geant4.solid._TwistedSolid

makeFaceFromLayer(*layer*, *reverse=False*)

makeSide(*pal*, *pbl*, *pau*, *pbu*, *zl*, *zu*, *nsl*)

p = point a = first b = second u = upper l = lower

meshFromLayers(*layers*, *nsl*)

class pyg4ometry.geant4.solid.TwistedTrd(*name*, *twistedangle*, *pDx1*, *pDx2*, *pDy1*, *pDy2*, *pDz*, *registry*, *lunit='mm'*, *aunit='rad'*, *nstack=None*, *refine=0*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*, *pyg4ometry.geant4.solid.TwistedSolid.TwistedSolid*

Constructs a twisted general trapezoid.

Parameters

- **name** (*str*) – of solid
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle, must be less than 0.5*pi
- **pDx1** (*float*, Constant, Quantity, Variable, Expression) – length in x at surface positioned at -pDz/2
- **pDx2** (*float*, Constant, Quantity, Variable, Expression) – length in x at surface positioned at +pDz/2
- **pDy1** (*float*, Constant, Quantity, Variable, Expression) – length in y at surface positioned at -pDz/2
- **pDy2** (*float*, Constant, Quantity, Variable, Expression) – length in y at surface positioned at +pDz/2
- **pDz** (*float*, Constant, Quantity, Variable, Expression) – length in z
- **refine** (*int*) – number of steps to iteratively smoothen the mesh by doubling the number of vertices at every step
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nstack** (*int*) – number of theta elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

checkParameters()

makeLayers(*pl1*, *pl2*, *pl3*, *pl4*, *pu1*, *pu2*, *pu3*, *pu4*, *pDz*, *theta*, *nsl*)

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)
Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.TwistedTubs(*name*, *endinnerrad*, *endouterrad*, *zlen*, *phi*, *twistedangle*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *nstack=None*, *addRegistry=True*)
Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Creates a twisted tube segement. Note that only 1 constructor is supprted.

Parameters

- **name** (*str*) – of solid
- **endinnerrad** (*float*, Constant, Quantity, Variable, Expression) – inner radius at the end of the segment
- **endinnerrad** – outer radius at the end of the segment
- **zlen** (*float*, Constant, Quantity, Variable, Expression) – length of the tube segement
- **twistedangle** (*float*, Constant, Quantity, Variable, Expression) – twist angle
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** – number of phi elements for meshing

- **nstack** (*int*) – number of theta elements for meshing

__repr__()

Return repr(self).

__str__()

Return str(self).

makeLayers(*verts_bot*, *verts_top*)

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**_GenericPolyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – name
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **numSide** (*float*, Constant, Quantity, Variable, Expression) – number of polygon sides
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate list
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate list

__repr__()

Return repr(self).

__str__()
Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid.GenericPolycone(*name*, *pSPhi*, *pDPhi*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *nslice=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – of solid
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate
- **registry** (Registry) – for storing solid
- **lunit** (*str*) – length unit (nm,um,mm,m,km) for solid
- **aunit** (*str*) – angle unit (rad,deg) for solid
- **nslice** (*int*) – number of phi elements for meshing

__repr__()
Return repr(self).

__str__()
Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid._SolidBase(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**GenericPolyhedra**(*name*, *pSPhi*, *pDPhi*, *numSide*, *pR*, *pZ*, *registry*, *lunit='mm'*, *aunit='rad'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a solid of rotation using an arbitrary 2D surface defined by a series of (r,z) coordinates.

Parameters

- **name** (*str*) – name
- **pSPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at start of rotation
- **pDPhi** (*float*, Constant, Quantity, Variable, Expression) – angle Phi at end of rotation
- **numSide** (*float*, Constant, Quantity, Variable, Expression) – number of polygon sides
- **pR** (*list of float*, Constant, Quantity, Variable, Expression) – r coordinate list
- **pZ** (*list of float*, Constant, Quantity, Variable, Expression) – z coordinate list

__repr__()

Return repr(self).

__str__()

Return str(self).

checkParameters()

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**GenericTrap**(*name*, *v1x*, *v1y*, *v2x*, *v2y*, *v3x*, *v3y*, *v4x*, *v4y*, *v5x*, *v5y*, *v6x*, *v6y*, *v7x*, *v7y*, *v8x*, *v8y*, *dz*, *registry*, *nstack=20*, *lunit='mm'*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs an arbitrary trapezoid using two quadrilaterals sitting on two parallel planes. Vertices 1-4 define the quadrilateral at -dz and vertices 5-8 define the quadrilateral at +dz. This solid is called Arb8 in GDML notation.

Parameters

- **name** – string, name of the volume
- **v1x** – vertex 1 x position
- **v1y** – vertex 1 y position
- **v2x** – vertex 2 x position
- **v2y** – vertex 2 y position
- **v3x** – vertex 3 x position
- **v3y** – vertex 3 y position
- **v4x** – vertex 4 x position
- **v4y** – vertex 4 y position
- **v5x** – vertex 5 x position
- **v5y** – vertex 5 y position
- **v6x** – vertex 6 x position
- **v6y** – vertex 6 y position
- **v7x** – vertex 7 x position
- **v7y** – vertex 7 y position
- **v8x** – vertex 8 x position
- **v8y** – vertex 8 y position
- **dz** – half length along z
- **registry** (Registry) – for storing solid

__repr__()

__str__()

polygon_area(*vertices*)

get_vertex(*index*)

makeLayers(*verts_bot*, *verts_top*)

mesh()

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**TessellatedSolid**(*name*, *meshTess*, *registry*, *meshtype=MeshType.Freecad*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a tessellated solid

Parameters

- **name** (*str*) – of solid
- **mesh** (*Freecad, Gdml or Stl*) – mesh
- **registry** (Registry) – for storing solid
- **meshtype** (*MeshType.Freecad*) – type of mesh

class MeshType

Freecad = 1

Gdml = 2

Stl = 3

__repr__()

Return repr(self).

__str__()

Return str(self).

addVertex(*vertex*)

addTriangle(*triangle*)

removeDuplicateVertices()

mesh()

pyg4ometry.geant4.solid.**createTessellatedSolid**(*name*, *polygons*, *reg*)

Args:

name: Name of the tessallated solid polygons: list of polygons (list of points given in clockwise order). All polygons should have the same number of points. reg: registry

Returns: TessellatedSolid

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**MultiUnion**(*name*, *objects*, *transformations*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Union between two or more solids.

Parameters

- **name** (*str*) – of solid
- **objects** – unrotated, untranslated solid objects to form union
- **transformations** – [[rot1,tra1],[rot2,tra2], [rot3,tra3] ..] or [[[a,b,g],[dx,dy,dz]], [[a,b,g],[dx,dy,dz]], [[a,b,g],[dx,dy,dz]], . . .]
- **registry** (Registry) – for storing solid
- **addRegistry** – Add solid to registry

__repr__()

Return repr(self).

__str__()

Return str(self).

mesh()

pyg4ometry.geant4.solid.**rad2deg**(*rad*)

Convert rad in radians into degrees

Parameters
rad (*float*) – Input in radians

Returns
rad in degrees

Return type
float

pyg4ometry.geant4.solid.**deg2rad**(*deg*)

Convert deg in degrees into radians

Parameters
deg (*float*) – Input in degrees

Returns
deg in radians

Return type
float

pyg4ometry.geant4.solid.**grad2rad**(*gradians*)

Convert rad in gradians into radians

Parameters
gradians (*float*) – Input in gradians

Returns
gradians in radians

Return type
float

pyg4ometry.geant4.solid.**tbxyz2axisangle**(*rv*)

Tait-Bryan x-y-z rotation to axis-angle representation Algorithm from http://www.sedris.org/wg8home/Documents/WG80485.pdf

For converting rotation angles to an active axis/angle pair for use in pycsg. Order of rotation: x->y->z.

Parameters
rv (*float(3)*) – rotation angles

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**matrix2axisangle**(*matrix*)

Convert 3x3 transformation matrix to axis angle representation

Parameters
matrix (*array(3,3)*) – 3x3 rotation matrix array

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.geant4.solid.**axisangle2matrix**(*axis*, *angle*)

Convert axis angle to transformation matrix

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
transformation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix2tbxyz**(*matrix*)

Convert rotation matrix to Tait-Bryan angles. Order of rotation is x -> y -> z.

Parameters
matrix – active (positive angle = anti-clockwise rotation about that axis when looking at the axis) matrix.

Returns
[x, y, z] Tait-Bryan angles in a list.

Return type
list(3)

pyg4ometry.geant4.solid.**axisangle2tbxyz**(*axis*, *angle*)

Convert axis and angle to tait bryan angles

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
tait bryan angles (x-y-z)

Return type
array(3)

pyg4ometry.geant4.solid.**tbxyz2matrix**(*angles*)

Convert tait bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**tbzyx2matrix**(*angles*)

Convert Tait-Bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.geant4.solid.**matrix_from**(*v_from*, *v_to*)

Returns the rotation matrix that rotates v_from to parallel to v_to.

Useful for ensuring a given face points in a certain direction.

v_from and v_to should be array-like three-vectors.

pyg4ometry.geant4.solid.**_rodrigues_anti_parallel**(*v_from*, *v_to*)

v_from = vector FROM v_to = vector TO

source: http://en.citizendium.org/wiki/Rotation_matrix#Case_that_.22from.22_and_.22to.22_vectors_are_anti-parallel

pyg4ometry.geant4.solid.**are_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are parallel

Return type
bool

pyg4ometry.geant4.solid.**are_anti_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are antiparallel

Return type
bool

pyg4ometry.geant4.solid.**reverse**(*angles*)

Invert the rotation represented by these angles.

pyg4ometry.geant4.solid.**two_fold_orientation**(*v1*, *v2*, *e1*, *e2*)

matrix_from will align one vector with another, but there are an infinite number of such matrices that align two vectors. This further contrains the rotation by introducing a second pair of vectors.

v1 start v v2 end v e1 start e e2 end v

class pyg4ometry.geant4.solid.**_SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)

Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()

return a TessellatedSolid instance based on the mesh of this solid.

class pyg4ometry.geant4.solid.**Scaled**(*name*, *solid*, *pX*, *pY*, *pZ*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.solid.SolidBase.SolidBase*

Constructs a scaled sold.

Parameters

- **name** (*str*) – of solid for registry
- **pX** (*float*, Constant, Quantity, Variable, Expression) – scale in x
- **pY** (*float*, Constant, Quantity, Variable, Expression) – scale in y
- **pZ** (*float*, Constant, Quantity, Variable, Expression) – scale in z
- **registry** (Registry) – for storing solid

Poram solid
reference for scaling

__repr__()
Return repr(self).

__str__()
Return str(self).

mesh()

class pyg4ometry.geant4.solid.**SolidBase**(*name*, *type*, *registry=None*)

Base class for all solids

property name

evaluateParameter(*obj*)

evaluateParameterWithUnits(*varName*)

_addProperty(*attribute*)

_setProperty(*attribute*, *value*)

_getProperty(*attribute*)

_twoPiValueCheck(*attribute*, *aunit='rad'*)
Raises a ValueError if the attribute is over pyg4ometry.config.twoPiComparisonTolerance **over** 2 x pi.

conver2Tessellated()
return a TessellatedSolid instance based on the mesh of this solid.

Submodules

pyg4ometry.geant4.AssemblyVolume

Module Contents

Classes

AssemblyVolume	AssemblyVolume : similar to a logical volume but does not have a sense of

class pyg4ometry.geant4.AssemblyVolume.**AssemblyVolume**(*name*, *registry=None*, *addRegistry=True*)

AssemblyVolume : similar to a logical volume but does not have a sense of shape, material or field :param name: of assembly volume :type name: str :param registry: :type registry: :param addRegistry: :type addRegistry: bool

__repr__()

Return repr(self).

add(*physicalVolume*)

_getDaughterMeshesByName(*name*)

_getDaughterMeshesByIndex(*index*)

_getDaughterMeshes()

Get daughter meshes for overlap checking. return [daughterMesh,..],[daughterBoundingMesh,..][daughterName,...]

_getPVMeshes(*pv*)

Can technically return more than one mesh if the daughter is also an assembly.

_getPhysicalDaughterMesh(*pv*, *warn=True*)

Return a (cloned from the lv) mesh of a given pv with rotation,scale, translation evaluated.

clipGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*, *replace=False*, *depth=0*, *solidUsageCount=_defaultdict(int)*, *lvUsageCount=_defaultdict(int)*)

Clip the geometry to newSolid, placed with rotation and position.

extent(*includeBoundingSolid=True*)

depth(*depth=0*)

Depth for LV-PV tree

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

logicalVolume(*material='G4_Galactic'*, *solidName='worldSolid'*)

Return an logical volume of this this assembly volume, in effect adding a cuboid solid and material of this logical volume, retaining all of the relative daughter placements.

makeWorldVolume(*material='G4_Galactic'*)

dumpStructure(*depth=0*)

pyg4ometry.geant4.BorderSurface

Module Contents

Classes

BorderSurface

class pyg4ometry.geant4.BorderSurface.**BorderSurface**(*name*, *physref1*, *physref2*, *surface_property*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.SurfaceBase.SurfaceBase*

__repr__()

pyg4ometry.geant4.DivisionVolume

Module Contents

Classes

DivisionVolume	DivisionVolume: G4PVDivision

class pyg4ometry.geant4.DivisionVolume.**DivisionVolume**(*name*, *logicalVolume*, *motherVolume*, *axis*, *ndivisions=-1*, *width=-1*, *offset=0*, *registry=None*, *addRegistry=True*, *unit='mm'*)

Bases: *pyg4ometry.geant4.PhysicalVolume.PhysicalVolume*

DivisionVolume: G4PVDivision

Parameters

- **name** – of physical volume
- **logical** – volume to be placed
- **mother** – logical volume,
- **axis** – kXAxis,kYAxis,kZAxis,kRho,kPhi
- **ncopies** – number of replicas
- **width** – spacing between replicas along axis
- **offset** – of grid

class Axis

kXAxis = 1

kYAxis = 2

kZAxis = 3

kRho = 4

kPhi = 5

getMotherSize()

checkAxis(*allowed_axes*)

divideBox(*offset*, *width*, *ndiv*)

divideTubs(*offset*, *width*, *ndiv*)

divideCons(*offset*, *width*, *ndiv*)

dividePara(*offset*, *width*, *ndiv*)

divideTrd(*offset*, *width*, *ndiv*)

dividePolycone(*offset*, *width*, *ndiv*)

dividePolyhedra(*offset*, *width*, *ndiv*)

createDivisionMeshes()

__repr__()

extent(*includeBoundingSolid=True*)

pyg4ometry.geant4.LogicalVolume

Module Contents

Classes

LogicalVolume	LogicalVolume : G4LogicalVolume

Functions

_solid2tessellated(solid)

pyg4ometry.geant4.LogicalVolume.**_solid2tessellated**(*solid*)

class pyg4ometry.geant4.LogicalVolume.**LogicalVolume**(*solid*, *material*, *name*, *registry=None*, *addRegistry=True*, ***kwargs*)

LogicalVolume : G4LogicalVolume

Parameters

- **solid** –
- **material** –
- **name** (*str*) –
- **registry** –
- **addRegistry** (*bool*) –

__repr__()

Return repr(self).

reMesh(*recursive=False*)

Regenerate the visualisation for this logical volume. Required if the geometry is modified and overlap checking is subsequently required or revisualisation.

add(*physicalVolume*)

Add physical volume to this logicalVolume

Parameters
physicalVolume (PhysicalVolume, ReplicaVolume, ParameterisedVolume, DivisionVolume) – physical volume to add

addBDSIMObject(*bdsimobject*)

_getPhysicalDaughterMesh(*pv*, *warn=True*)

Return a (cloned from the lv) mesh of a given pv with rotation,scale, translation evaluated.

cullDaughtersOutsideSolid(*solid*, *rotation=None*, *position=None*)

Given a solid with a placement rotation and position inside this logical volume, remove (cull) any daughters that would not lie entirely within it. The rotation and position are applied to the solid w.r.t. the frame of this logical volume.

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv

transformDaughters(*rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Transform the daugter volumes (without clipping)

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

replaceSolid(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Replace the outer solid with optional position and rotation

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

clipGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*, *replace=False*, *depth=0*, *solidUsageCount=_defaultdict(int)*, *lvUsageCount=_defaultdict(int)*)

Clip the geometry to newSolid, placed with rotation and position.

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)
- **replace** (*bool*) – replace the outer solid or not
- **depth** (*int*) – recursion depth (DO NOT USE)
- **solidUsageCount** (*defaultdict*) – solid name dictionary for replacement recursion (DO NOT USE)
- **lvUsageCount** (*defaultdict*) – lv name dictionary for replacement recursion (DO NOT USE)

changeSolidAndTrimGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Change the solid of this logical volume, remove any daughters that will lie outside it, and form new Boolean intersection solids for any daughters that cross the boundary (intersect) it. The rotation and translation are with respect to the original frame and all daughters will now be replaced with respect to the new frame. Therefore, that same rotation and translation should be use to re-place this logical volume if desired. The default is none though, so the frame would nominally remain the same.

Parameters

- **newSolid** (*any of pyg4ometry.geant4.solid*) – new solid to use for this logical volume
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan rotation for the new solid w.r.t. old frame (i.e. current lv)
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation for the new solid w.r.t. old frame (i.e. current lv)

checkOverlaps(*recursive=False*, *coplanar=False*, *debugIO=False*, *printOut=True*, *nOverlapsDetected=[0]*)

Check based on the meshes in each logical volume if there are any geometrical overlaps. By default, overlaps are checked between daughter volumes and with the mother volume itself (protrusion). Coplanar overlaps may also be checked (default on).

Print out will be given for any overlaps detected and the visualiser will show the colour coded overlaps.

Parameters

- **recursive** – bool - Whether to descend into the daughter volumes and check their contents also.
- **coplanar** – bool - Whether to check for coplanar overlaps
- **debugIO** – bool - Print out for every check made
- **printOut** – bool - (internal) Whether to print out a summary of N overlaps detected
- **nOverlapsDetected** – [int] - (internal) counter for recursion - ignore

setSolid(*solid*)

Set (replace) the outer solid. Does not change the placement of the daughters in the volume. If there is a transformation then use replaceSolid

makeSolidTessellated()

Make solid tesselated. Sometimes useful when a boolean cannot be visualised in Geant4

addAuxiliaryInfo(*auxiliary*)

Add auxilary information to logical volume :param auxiliary: auxiliary information for the logical volume :type auxiliary: tuple or list

extent(*includeBoundingSolid=False*)

Compute the axis aligned extent of the logical volume.

Parameters
includeBoundingSolid (*bool*) – Include the bounding solid or not

depth(*depth=0*)

Depth for LV-PV tree

clipSolid(*lengthSafety=1e-06*)

Assuming the solid of this LV is a Box, reduce its dimensions and re-placement all daughters to reduce the box to the minimum (axis-aligned) bounding box. This updates the dimensions of the box and the translation of each daughter physical volume.

Parameters
lengthSafety (*float*) – safety length

makeLogicalPhysicalNameSets()

Return a set of logical names and physical names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc.

findLogicalByName(*name*)

Return a list of LogicalVolume instances used inside this logical volume as daughters (at any level inside) with the given name.

Parameters
name (*str*) – lv name

makeMaterialNameSet()

Return a set of material names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc etc.

assemblyVolume(*materialName='G4_AIR0x7f8441173ac0'*)

Return an assembly volume of this logical volume, in effect removing the solid and material of this logical volume, but retaining all of the relative daughter placements.

makeWorldVolume(*worldMaterial='G4_Galactic'*)

This will create a container box according to the extents of this logical volume: an axis-aligned bounding-box. It will be filled with the given material (predefined by name) and assigned as the world volume (outermost) of the registry according to this logical volume.

dumpStructure(*depth=0*)

pyg4ometry.geant4.Loop

Module Contents

Classes

Loop	

class pyg4ometry.geant4.Loop.**Loop**(*variable*, *loopFrom*, *loopTo*, *loopStep*, *objectsToLoop*)

expandLoop()

pyg4ometry.geant4.ParameterisedVolume

Module Contents

Classes

ParameterisedVolume	ParametrisedVolume

class pyg4ometry.geant4.ParameterisedVolume.**ParameterisedVolume**(*name*, *logicalVolume*, *motherVolume*, *ncopies*, *paramData*, *transforms*, *registry=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.ReplicaVolume.ReplicaVolume*

ParametrisedVolume :param name: of parametrised volume :type name: str :param logical: volume to be placed :type logical: logicalVolume :param mother: volume logical volume :type mother: logicalVolume :param ncopies: number of parametrised volumes :type ncopies: int

class BoxDimensions(*pX*, *pY*, *pZ*, *lunit='mm'*)

class TubeDimensions(*pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class ConeDimensions(*pRMin1*, *pRMax1*, *pRMin2*, *pRMax2*, *pDz*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class OrbDimensions(*pRMax*, *lunit='mm'*)

class SphereDimensions(*pRMin*, *pRMax*, *pSPhi*, *pDPhi*, *pSTheta*, *pDTheta*, *lunit='mm'*, *aunit='rad'*)

class TorusDimensions(*pRMin*, *pRMax*, *pRTor*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class HypeDimensions(*innerRadius*, *outerRadius*, *innerStereo*, *outerStereo*, *lenZ*, *lunit='mm'*, *aunit='rad'*)

class ParaDimensions(*pX*, *pY*, *pZ*, *pAlpha*, *pTheta*, *pPhi*, *lunit='mm'*, *aunit='rad'*)

class TrdDimensions(*pX1*, *pX2*, *pY1*, *pY2*, *pZ*, *lunit='mm'*)

class TrapDimensions(*pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pAlp1*, *pDy2*, *pDx3*, *pDx4*, *pAlp2*, *lunit='mm'*, *aunit='rad'*)

class PolyconeDimensions(*pSPhi*, *pDPhi*, *pZpl*, *pRMin*, *pRMax*, *lunit='mm'*, *aunit='rad'*)

class PolyhedraDimensions(*pSPhi*, *pDPhi*, *numSide*, *pZpl*, *pRMin*, *pRMax*, *lunit='mm'*, *aunit='rad'*)

class EllipsoidDimensions(*pxSemiAxis*, *pySemiAxis*, *pzSemiAxis*, *pzBottomCut*, *pzTopCut*, *lunit='mm'*)

createParameterisedMeshes()

__repr__()

extent(*includeBoundingSolid=True*)

pyg4ometry.geant4.PhysicalVolume

Module Contents

Classes

PhysicalVolume	PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

class pyg4ometry.geant4.PhysicalVolume.**PhysicalVolume**(*rotation*, *position*, *logicalVolume*, *name*, *motherVolume*, *registry=None*, *copyNumber=0*, *addRegistry=True*, *scale=None*)

PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

Parameters

- **rotation** – [float,float,float] - rotations about x,y,z axes of mother volume
- **position** – [float,float,float] - translation with respect to mother volume
- **logicalVolume** – *pyg4ometry.geant4.LogicalVolume* - instance to place
- **name** – str - name of this placement
- **motherVolume** – *pyg4ometry.geant4.LogicalVolume* - mother volume to place into
- **registry** – *pyg4ometry.geant4.Registry* - registry to register to
- **copyNumber** – int - copy number of the placement that can be used for sensitivity
- **addRegistry** – bool - whether to add to the registry or not

__repr__()

Return repr(self).

extent(*includeBoundingSolid=True*)

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

pyg4ometry.geant4.Registry

Module Contents

Classes

Registry	Object to store geometry for input and output.
GeometryComplexityInformation	

Functions

solidName(var)	
removeprefix(string, prefix, /)	
AnalyseGeometryComplexity(logicalVolume)	Analyse a geometry tree starting from a logical volume.
_UpdateComplexity(lv, info)	
AnalyseGeometryStructure(registry[, lv_name, debug, ...])	Produce a pandas dataframe representing the structure of the geometry.
DumpGeometryStructureTree(lv[, depth])	

pyg4ometry.geant4.Registry.**solidName**(*var*)

pyg4ometry.geant4.Registry.**removeprefix**(*string*, *prefix*, */*)

Parameters

- **string** (*str*) –
- **prefix** (*str*) –

Return type
str

class pyg4ometry.geant4.Registry.**Registry**

Object to store geometry for input and output. All of the pyg4ometry classes can be used without storing them in the Registry. The registry is used to write the GDML output file. A registry needs to be used in conjunction with GDML Define objects for evaluation of expressions.

clear()

Empty all internal structures

getExpressionParser()

registerSolidEdit(*solid*)

addMaterial(*material*, *dontWarnIfAlreadyAdded=False*)

Register a material with this registry.

Parameters
material (Material) – Material object for storage

transferMaterial(*material*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a material to this registry. This can operate on a Material, an Isotope and an Element instance.

addSolid(*solid*)

Register a solid with this registry.

Parameters
solid (*One of the geant4 solids*) – Solid object for storage

transferSolid(*solid*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a solid to this registry. Doesn't handle any members' transferal - only the solid itself.

Parameters
solid (*One of the geant4 solids*) – Solid object for storage

addLogicalVolume(*volume*)

Register a logical volume with this registry. Also accepts Assembly Volumes.

Parameters
volume (LogicalVolume) – LogicalVolume object for storage

transferLogicalVolume(*volume*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a logical volume to this registry. Doesn't handle any members' transferal - only the logical volume itself.

addPhysicalVolume(*volume*)

Registry a physical volume with this registry.

Parameters
volume (PhysicalVolume) – PhysicalVolume object for storage

transferPhysicalVolume(*volume*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a physical volume to this registry. Doesn't handle any members' transferal - only the physical volume itself.

addSurface(*surface*)

Register a surface with this registry.

Parameters
surface (pyg4ometry.geant4.BorderSurface *or* pyg4ometry.geant4.SkinSurface) – Surface

transferSurface(*surface*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a surface to this registry.

addAuxiliary(*auxiliary*)

addDefine(*define*)

Register a define with this registry.

Parameters
define (Constant, Quantity, Variable, Matrix) – Definition object for storage

transferDefine(*define*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a single define from another registry to this one. No checking on previous registry or not.

transferDefines(*var*, *otherRegistry*, *incrementRenameDict={}*, *userRenameDict=None*)

This function tolerates all types of defines including vector ones.

Transfer defines from one registry to another recursively. A define may not be part of the old registry so won't be added to this one. A define may be a vector or composite and its 'bits' may be in the (old) registry so each part should be checked.

In "3x + 2", "x" would be a variable". In "3.5*2" there would be no variables.

setWorld(*worldIn*)

The argument can either be the name of logical volume of the world or the pyg4ometry.geant4.LogicalVolume instance of the world volume. The term world is used to refer to the outermost volume of the hierarchy.

setWorldVolume(*worldIn*)

An alias for some of us who can't remember.

_orderMaterialList(*materials*, *materials_ordered=[]*)

orderMaterials()

Need to have a ordered list of all material entities for writing to GDML. GDML needs to have the isotopes/elements/materials defined in use order

orderLogicalVolumes(*lvName*, *first=True*)

Need to have an ordered list from most basic (solid) object upto physical/logical volumes for writing to GDML. GDML needs to have the solids/booleans/volumes defined in order

addVolumeRecursive(*volume*, *collapseAssemblies=False*, *incrementRenameDict=None*, *userRenameDict=None*)

Transfer a volume hierarchy to this registry. Any objects that had a registry set to another will be set to this one and will be owned by it effectively. :param volume: PhysicalVolume or LogicalVolume or AssemblyVolume. :type volume: pyg4ometry.geant4.PhysicalVolume, pyg4ometry.geant4.LogicalVolume, pyg4ometry.geant4.AssemblyVolume. :param collapseAssemblies: if True, daughters of AssemblyVolume's will be attached directly to the mother of the assembly and the AssemblyVolume itself will be eliminated from the geometry tree :param incrementRenameDict: ignore - dictionary used internally for potentially incrementing names :param userRenameDict: a dictionary of find/replace regex strings to be used to rename volumes/materials/etc.

In the case where some object or variable has a name (e.g. 'X') that already exists in this registry, it will be incremented to 'X_1'.

addAndCollapseAssemblyVolumeRecursive(*assemblyPV*, *motherVol*, *positions*, *rotations*, *scales*, *names*, *incrementRenameDict*, *userRenameDict*)

Transfer and collapse an AssemblyVolume hierarchy to this registry. Daughter volumes are copied, renamed, and attached to the supplied mother LogicalVolume with the correct position/rotation. Any objects that had a registry set to another will be set to this one and will be owned by it effectively. :param assemblyPV: the PhysicalVolume that places an AssemblyVolume :param motherVol: the LogicalVolume to which all daughters of the AssemblyVolume (including any daughters of nested AssemblyVolume's) should be attached :param positions: a list (initially empty) to hold position objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param rotations: a list (initially empty) to hold rotation objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param scales: a list (initially empty) to hold scale objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param names: a list (initially empty) to hold the names of each AssemblyVolume in a hierarchy of nested assemblies (used to set unique names for the daughters within the motherVol) :param incrementRenameDict: ignore - dictionary

used internally for potentially incrementing names :param userRenameDict: a dictionary of find/replace regex strings to be used to rename volumes/materials/etc.

transferSolidDefines(*solid*, *incrementRenameDict={}*, *userRenameDict=None*)

For each parameter in a given solid (unique to each) check if it's a define and transfer that over.

volumeTree(*lvName*)

Not sure what this method is used for

solidTree(*solidName*)

Not sure what this method is used for

getWorldVolume()

printStats()

structureAnalysis(*lv_name=None*, *debug=False*, *level=0*, *df=None*)

_findDictByName(*dic*, *nameFragment*)

Find a object which name matches (or partially matches) nameFragment, returns a list of objects

findSolidByName(*nameFragment='box'*)

Find a solid which name matches (or partially matches) nameFragment, returns a list of solids

findMaterialByName(*nameFragment='G4_AIR'*)

Find a material which name matches (or partially matches) nameFragment, returns a list of materials

findLogicalVolumeByName(*nameFragment='World'*)

Find a logical volume which name matches (or partially matches) nameFragment, returns a list of LogicalVolumes

findPhysicalVolumeByName(*nameFragment*)

Find a physical volume which name matches (or partially matches) nameFragment, returns a list of LogicalVolumes

class pyg4ometry.geant4.Registry.**GeometryComplexityInformation**

printSummary(*boolDepthLimit=3*)

pyg4ometry.geant4.Registry.**AnalyseGeometryComplexity**(*logicalVolume*)

Analyse a geometry tree starting from a logical volume. Produces an instance of *GeometryComplexityInformation* with summary information. Provides:

- count per solid type
- number of daughters per logical volume
- dictionary of N daughters for each logical volume name
- depth count of Boolean solids

ie a Boolean of a Boolean returns 2, a Boolean of two primitives returns 1

- a dictionary of boolean depth for each logical volume name

Example:

```
info = AnalyseGeometryComplexity(lv)
info.printSummary()
```

pyg4ometry.geant4.Registry.**_UpdateComplexity**(*lv*, *info*)

pyg4ometry.geant4.Registry.**AnalyseGeometryStructure**(*registry*, *lv_name=None*, *debug=False*, *level=0*, *df=None*)

Produce a pandas dataframe representing the structure of the geometry.

pyg4ometry.geant4.Registry.**DumpGeometryStructureTree**(*lv*, *depth=0*)

pyg4ometry.geant4.ReplicaVolume

Module Contents

Classes

ReplicaVolume	ReplicaVolume: G4PVReplica

class pyg4ometry.geant4.ReplicaVolume.**ReplicaVolume**(*name*, *logicalVolume*, *motherVolume*, *axis*, *nreplicas*, *width*, *offset=0*, *registry=None*, *addRegistry=True*, *wunit='mm'*, *ounit='mm'*)

Bases: *pyg4ometry.geant4.PhysicalVolume.PhysicalVolume*

ReplicaVolume: G4PVReplica

Parameters

- **name** – of physical volume
- **logical** – volume to be placed
- **mother** – logical volume,
- **axis** – kXAxis,kYAxis,kZAxis,kRho,kPhi
- **ncopies** – number of replicas
- **width** – spacing between replicas along axis
- **offset** – of grid

class Axis

kXAxis = 1

kYAxis = 2

kZAxis = 3

kRho = 4

kPhi = 5

GetAxisName()

_checkInternalOverlaps(*debugIO=False*, *nOverlapsDetected=[0]*)

Check if there are overlaps with the nominal mother volume. ie it possible to provide an incorrect mother volume / logical volume and parameterisation.

createReplicaMeshes()

getPhysicalVolumes()

return a list of temporary (ie not added to the relevant registry) PhysicalVolume instances with appropriate transforms including any daughter ReplicaVolumes.

The exception is for kRho axis where new unique solids and logical volumes are required. Therefore, these are added to the registry and inadvertently to the mother LV as PVS.

__repr__()

extent(*includeBoundingSolid=True*)

pyg4ometry.geant4.SkinSurface

Module Contents

Classes

SkinSurface

class pyg4ometry.geant4.SkinSurface.**SkinSurface**(*name*, *volumeref*, *surface_property*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.SurfaceBase.SurfaceBase*

__repr__()

pyg4ometry.geant4.SurfaceBase

Module Contents

Classes

SurfaceBase

class pyg4ometry.geant4.SurfaceBase.**SurfaceBase**(*name*, *type*, *surface_property*, *registry*, *addRegistry*)

_chkType(*obj*, *t*, *prop*)

pyg4ometry.geant4._Material

Module Contents

Classes

MaterialBase	
Material	This class provides an interface to GDML material definitions.
Element	This class provides an interface to GDML material definitions. Because of the different options
Isotope	This class that handles isotopes as components of composite materials. An element can be

Functions

getNistMaterialDict()	
getNistMaterialList()	
getNistElementZToName()	
_getClassVariables(obj)	
_makeNISTCompoundList()	
_safeName(name)	
loadNISTMaterialDict()	
nist_materials_name_lookup(name)	
nist_materials_z_lookup(z)	
nist_element_2geant4Element(name[, reg])	This returns and instance of either ElementSimple or ElementIsotopeMixture.
nist_material_2geant4Material(name[, reg])	
MaterialPredefined(name[, registry])	Proxy method to construct a NIST compound material - this is just a handle as nothing
MaterialArbitrary(name[, registry])	Just a name of a material. WARNING: It is left to the
MaterialSingleElement(name, atomic_number, ...[, ...])	Proxy method to construct a simple material - full description of the element contained is contained in one definition
MaterialCompound(name, density, number_of_components)	Proxy method to construct a composite material - can be any mixture of Elements and/or Materials
ElementSimple(name, symbol, Z, A[, registry])	Proxy method to construct a simple element - full description of the element contained is contained in one definition
ElementIsotopeMixture(name, symbol, n_comp[, ...])	Proxy method to construct a composite element - a mixture of predefined isotopes

Attributes

_nistMaterialDict
_nistMaterialList
_nistElementZToName

pyg4ometry.geant4._Material.**_nistMaterialDict**

pyg4ometry.geant4._Material.**_nistMaterialList**

pyg4ometry.geant4._Material.**_nistElementZToName**

pyg4ometry.geant4._Material.**getNistMaterialDict**()

pyg4ometry.geant4._Material.**getNistMaterialList**()

pyg4ometry.geant4._Material.**getNistElementZToName**()

pyg4ometry.geant4._Material.**_getClassVariables**(*obj*)

pyg4ometry.geant4._Material.**_makeNISTCompoundList**()

pyg4ometry.geant4._Material.**_safeName**(*name*)

pyg4ometry.geant4._Material.**loadNISTMaterialDict**()

pyg4ometry.geant4._Material.**nist_materials_name_lookup**(*name*)

pyg4ometry.geant4._Material.**nist_materials_z_lookup**(*z*)

pyg4ometry.geant4._Material.**nist_element_2geant4Element**(*name*, *reg=None*)

This returns and instance of either ElementSimple or ElementIsotopeMixture.

pyg4ometry.geant4._Material.**nist_material_2geant4Material**(*name*, *reg=None*)

pyg4ometry.geant4._Material.**MaterialPredefined**(*name*, *registry=None*)

Proxy method to construct a NIST compound material - this is just a handle as nothing needs to be additionally defined for a NIST compound. A check is performed on the name to ensure it is a valid NIST specifier.

Inputs:

name - string

pyg4ometry.geant4._Material.**MaterialArbitrary**(*name*, *registry=None*)

Just a name of a material. WARNING: It is left to the user to ensure that the name is valid.

Inputs:

name - string

pyg4ometry.geant4._Material.**MaterialSingleElement**(*name*, *atomic_number*, *atomic_weight*, *density*, *registry=None*, *tolerateZeroDensity=False*)

Proxy method to construct a simple material - full description of the element contained is contained in one definition

Inputs:

name - string atomic_number - int, number of protons, commonly known as 'Z' atomic_weight - molar weight in g/mole, commonly known as 'A' density - float, material density in g/cm3

pyg4ometry.geant4._Material.**MaterialCompound**(*name*, *density*, *number_of_components*, *registry=None*, *tolerateZeroDensity=False*, *state=None*)

Proxy method to construct a composite material - can be any mixture of Elements and/or Materials

Inputs:

name - string density - float, material density in g/cm3 number_of_components - int, number of components in the mixture

pyg4ometry.geant4._Material.**ElementSimple**(*name*, *symbol*, *Z*, *A*, *registry=None*)

Proxy method to construct a simple element - full description of the element contained is contained in one definition

Inputs:

name - string symbol - string, chemical formula of the compound Z - int, Atomic number A - float, mass number

pyg4ometry.geant4._Material.**ElementIsotopeMixture**(*name*, *symbol*, *n_comp*, *registry=None*, *state=None*)

Proxy method to construct a composite element - a mixture of predefined isotopes

Inputs:

name - string symbol - string, chemical formula of the compound n_comp - int, number of isotope components

class pyg4ometry.geant4._Material.**MaterialBase**(*name*, *state=None*, *registry=None*)

_addToRegistry()

get_material_oject(*material*)

set_registry(*registry*, *dontWarnIfAlreadyAdded=False*)

set_state(*state*)

__repr__()

Return repr(self).

class pyg4ometry.geant4._Material.**Material**(***kwargs*)

Bases: *MaterialBase*

This class provides an interface to GDML material definitions.

Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

MaterialSingleElement MaterialCompound MaterialPredefined

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string density - float atomic_number - int atomic_weight - float number_of_components - int state - string pressure - float pressure_unit - string temperature - float temperature_unit - string

property state_variables

add_element_massfraction(*element*, *massfraction*)

Add an element as a component to a material as a fraction of the material mass. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance massfraction - float, 0.0 < massfraction <= 1.0

add_element_natoms(*element*, *natoms*)

Add an element as a component to a material as a number of atoms in the material molecule. Can only add elements to materials defined as composite.

Inputs:

element - pyg4ometry.geant4.Material.Element instance natoms - int, number of atoms in the compound molecule

add_material(*material*, *fractionmass*)

Add a material as a component to another material (mixture) as a fraction of the mixture mass. Can only add new materials to materials defined as composite.

Inputs:
material - pyg4ometry.geant4.Material.Material instance massfraction - float, 0.0 < massfraction <= 1.0

set_pressure(*value*, *unit='pascal'*)

set_temperature(*value*, *unit='K'*)

__str__()

Return str(self).

addProperty(*name*, *matrix*)

Add a material property from a matrix.

Parameters

- **name** (*str*) – key of the material property
- **matrix** (Matrix) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str,float,int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.geant4._Material.**Element**(***kwargs*)

Bases: *MaterialBase*

This class provides an interface to GDML material definitions. Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

ElementSimple ElementIsotopeMixture

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string symbol - string Z - int A - int n_comp - int

add_isotope(*isotope*, *abundance*)

Add an isotope as a component to an element as an abundance fraction in the element.

Inputs:
element - pyg4ometry.geant4.Material.Isotope instance abundance - float, 0.0 < abundance <= 1.0

class pyg4ometry.geant4._Material.**Isotope**(*name*, *Z*, *N*, *a*, *registry=None*)

Bases: *MaterialBase*

This class that handles isotopes as components of composite materials. An element can be defined as a mixture of isotopes.

Inputs:
name - string Z - int, atomic number N - int, mass number a - float, molar weight in g/mole

Package Contents

Classes

_OverlapType	
LogicalVolume	LogicalVolume : G4LogicalVolume
PhysicalVolume	
AssemblyVolume	AssemblyVolume : similar to a logical volume but does not have a sense of
_PhysicalVolume	PhysicalVolume : G4VPhysicalVolume, G4PVPlacement
_OverlapType	
ReplicaVolume	ReplicaVolume: G4PVReplica
_ReplicaVolume	ReplicaVolume: G4PVReplica
ParameterisedVolume	
_PhysicalVolume	PhysicalVolume : G4VPhysicalVolume, G4PVPlacement
DivisionVolume	DivisionVolume: G4PVDivision
LogicalVolume	LogicalVolume : G4LogicalVolume
SurfaceBase	
SkinSurface	
PhysicalVolume	
SurfaceBase	
BorderSurface	
Registry	
GeometryComplexityInformation	
MaterialBase	
Material	This class provides an interface to GDML material definitions.
Element	This class provides an interface to GDML material definitions. Because of the different options
Isotope	This class that handles isotopes as components of composite materials. An element can be

Functions

IsAReplica(logicalVolume)	Utility function to test if an LV is really a replica volume. A replica is a special case
_solid2tessellated(solid)	
solidName(var)	
removeprefix(string, prefix, /)	
AnalyseGeometryComplexity(logicalVolume)	Analyse a geometry tree starting from a logical volume.
_UpdateComplexity(lv, info)	
AnalyseGeometryStructure(registry[, lv_name, debug, ...])	Produce a pandas dataframe representing the structure of the geometry.
DumpGeometryStructureTree(lv[, depth])	
getNistMaterialDict()	
getNistMaterialList()	
getNistElementZToName()	
_getClassVariables(obj)	
_makeNISTCompoundList()	
_safeName(name)	
loadNISTMaterialDict()	
nist_materials_name_lookup(name)	
nist_materials_z_lookup(z)	
nist_element_2geant4Element(name[, reg])	This returns and instance of either ElementSimple or ElementIsotopeMixture.
nist_material_2geant4Material(name[, reg])	
MaterialPredefined(name[, registry])	Proxy method to construct a NIST compound material - this is just a handle as nothing
MaterialArbitrary(name[, registry])	Just a name of a material. WARNING: It is left to the
MaterialSingleElement(name, atomic_number, ...[, ...])	Proxy method to construct a simple material - full description of the element contained is contained in one definition
MaterialCompound(name, density, number_of_components)	Proxy method to construct a composite material - can be any mixture of Elements and/or Materials
ElementSimple(name, symbol, Z, A[, registry])	Proxy method to construct a simple element - full description of the element contained is contained in one definition
ElementIsotopeMixture(name, symbol, n_comp[, ...])	Proxy method to construct a composite element - a mixture of predefined isotopes

Attributes

_nistMaterialDict
_nistMaterialList
_nistElementZToName

pyg4ometry.geant4.**IsAReplica**(*logicalVolume*)

Utility function to test if an LV is really a replica volume. A replica is a special case where we have an in-effect dummy mother and is detectable by there only being 1 daughter and it being a ReplicaVolume instance.

class pyg4ometry.geant4.**_OverlapType**

protrusion = 1

overlap = 2

coplanar = 3

pyg4ometry.geant4.**_solid2tessellated**(*solid*)

class pyg4ometry.geant4.**LogicalVolume**(*solid*, *material*, *name*, *registry=None*, *addRegistry=True*, ***kwargs*)

LogicalVolume : G4LogicalVolume

Parameters

- **solid** –
- **material** –
- **name** (*str*) –
- **registry** –
- **addRegistry** (*bool*) –

__repr__()

Return repr(self).

reMesh(*recursive=False*)

Regenerate the visualisation for this logical volume. Required if the geometry is modified and overlap checking is subsequently required or revisualisation.

add(*physicalVolume*)

Add physical volume to this logicalVolume

Parameters

physicalVolume (PhysicalVolume, ReplicaVolume, ParameterisedVolume, DivisionVolume) – physical volume to add

addBDSIMObject(*bdsimobject*)

_getPhysicalDaughterMesh(*pv*, *warn=True*)

Return a (cloned from the lv) mesh of a given pv with rotation,scale, translation evaluated.

cullDaughtersOutsideSolid(*solid*, *rotation=None*, *position=None*)

Given a solid with a placement rotation and position inside this logical volume, remove (cull) any daughters that would not lie entirely within it. The rotation and position are applied to the solid w.r.t. the frame of this logical volume.

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv

transformDaughters(*rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Transform the daugter volumes (without clipping)

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

replaceSolid(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Replace the outer solid with optional position and rotation

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

clipGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*, *replace=False*, *depth=0*, *solidUsageCount=_defaultdict(int)*, *lvUsageCount=_defaultdict(int)*)

Clip the geometry to newSolid, placed with rotation and position.

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)
- **replace** (*bool*) – replace the outer solid or not

- **depth** (*int*) – recursion depth (DO NOT USE)
- **solidUsageCount** (*defaultdict*) – solid name dictionary for replacement recursion (DO NOT USE)
- **lvUsageCount** (*defaultdict*) – lv name dictionary for replacement recursion (DO NOT USE)

changeSolidAndTrimGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Change the solid of this logical volume, remove any daughters that will lie outside it, and form new Boolean intersection solids for any daughters that cross the boundary (intersect) it. The rotation and translation are with respect to the original frame and all daughters will now be replaced with respect to the new frame. Therefore, that same rotation and translation should be use to re-place this logical volume if desired. The default is none though, so the frame would nominally remain the same.

Parameters

- **newSolid** (*any of pyg4ometry.geant4.solid*) – new solid to use for this logical volume
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan rotation for the new solid w.r.t. old frame (i.e. current lv)
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation for the new solid w.r.t. old frame (i.e. current lv)

checkOverlaps(*recursive=False*, *coplanar=False*, *debugIO=False*, *printOut=True*, *nOverlapsDetected=[0]*)

Check based on the meshes in each logical volume if there are any geometrical overlaps. By default, overlaps are checked between daughter volumes and with the mother volume itself (protrusion). Coplanar overlaps may also be checked (default on).

Print out will be given for any overlaps detected and the visualiser will show the colour coded overlaps.

Parameters

- **recursive** – bool - Whether to descend into the daughter volumes and check their contents also.
- **coplanar** – bool - Whether to check for coplanar overlaps
- **debugIO** – bool - Print out for every check made
- **printOut** – bool - (internal) Whether to print out a summary of N overlaps detected
- **nOverlapsDetected** – [int] - (internal) counter for recursion - ignore

setSolid(*solid*)

Set (replace) the outer solid. Does not change the placement of the daughters in the volume. If there is a transformation then use replaceSolid

makeSolidTessellated()

Make solid tesselated. Sometimes useful when a boolean cannot be visualised in Geant4

addAuxiliaryInfo(*auxiliary*)

Add auxilary information to logical volume :param auxiliary: auxiliary information for the logical volume :type auxiliary: tuple or list

extent(*includeBoundingSolid=False*)

Compute the axis aligned extent of the logical volume.

Parameters

includeBoundingSolid (*bool*) – Include the bounding solid or not

depth(*depth=0*)

Depth for LV-PV tree

clipSolid(*lengthSafety=1e-06*)

Assuming the solid of this LV is a Box, reduce its dimensions and re-placement all daughters to reduce the box to the minimum (axis-aligned) bounding box. This updates the dimensions of the box and the translation of each daughter physical volume.

Parameters
lengthSafety (*float*) – safety length

makeLogicalPhysicalNameSets()

Return a set of logical names and physical names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc.

findLogicalByName(*name*)

Return a list of LogicalVolume instances used inside this logical volume as daughters (at any level inside) with the given name.

Parameters
name (*str*) – lv name

makeMaterialNameSet()

Return a set of material names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc etc.

assemblyVolume(*materialName='G4_AIR0x7f8441173ac0'*)

Return an assembly volume of this logical volume, in effect removing the solid and material of this logical volume, but retaining all of the relative daughter placements.

makeWorldVolume(*worldMaterial='G4_Galactic'*)

This will create a container box according to the extents of this logical volume: an axis-aligned bounding-box. It will be filled with the given material (predefined by name) and assigned as the world volume (outermost) of the registry according to this logical volume.

dumpStructure(*depth=0*)

class pyg4ometry.geant4.**PhysicalVolume**(*rotation*, *position*, *logicalVolume*, *name*, *motherVolume*, *registry=None*, *copyNumber=0*, *addRegistry=True*, *scale=None*)

PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

Parameters

- **rotation** – [float,float,float] - rotations about x,y,z axes of mother volume
- **position** – [float,float,float] - translation with respect to mother volume
- **logicalVolume** – *pyg4ometry.geant4.LogicalVolume* - instance to place
- **name** – str - name of this placement
- **motherVolume** – *pyg4ometry.geant4.LogicalVolume* - mother volume to place into
- **registry** – *pyg4ometry.geant4.Registry* - registry to register to
- **copyNumber** – int - copy number of the placement that can be used for sensitivity
- **addRegistry** – bool - whether to add to the registry or not

__repr__()

Return repr(self).

extent(*includeBoundingSolid=True*)

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

class pyg4ometry.geant4.**AssemblyVolume**(*name*, *registry=None*, *addRegistry=True*)

AssemblyVolume : similar to a logical volume but does not have a sense of shape, material or field :param name: of assembly volume :type name: str :param registry: :type registry: :param addRegistry: :type addRegistry: bool

__repr__()

Return repr(self).

add(*physicalVolume*)

_getDaughterMeshesByName(*name*)

_getDaughterMeshesByIndex(*index*)

_getDaughterMeshes()

Get daughter meshes for overlap checking. return [daughterMesh,..],[daughterBoundingMesh,..][daughterName,...]

_getPVMeshes(*pv*)

Can technically return more than one mesh if the daughter is also an assembly.

_getPhysicalDaughterMesh(*pv*, *warn=True*)

Return a (cloned from the lv) mesh of a given pv with rotation,scale, translation evaluated.

clipGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*, *replace=False*, *depth=0*, *solidUsageCount=_defaultdict(int)*, *lvUsageCount=_defaultdict(int)*)

Clip the geometry to newSolid, placed with rotation and position.

extent(*includeBoundingSolid=True*)

depth(*depth=0*)

Depth for LV-PV tree

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

logicalVolume(*material='G4_Galactic'*, *solidName='worldSolid'*)

Return an logical volume of this this assembly volume, in effect adding a cuboid solid and material of this logical volume, retaining all of the relative daughter placements.

makeWorldVolume(*material='G4_Galactic'*)

dumpStructure(*depth=0*)

class pyg4ometry.geant4.**_PhysicalVolume**(*rotation*, *position*, *logicalVolume*, *name*, *motherVolume*, *registry=None*, *copyNumber=0*, *addRegistry=True*, *scale=None*)

PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

Parameters

- **rotation** – [float,float,float] - rotations about x,y,z axes of mother volume
- **position** – [float,float,float] - translation with respect to mother volume
- **logicalVolume** – *pyg4ometry.geant4.LogicalVolume* - instance to place

- **name** – str - name of this placement
- **motherVolume** – *pyg4ometry.geant4.LogicalVolume* - mother volume to place into
- **registry** – *pyg4ometry.geant4.Registry* - registry to register to
- **copyNumber** – int - copy number of the placement that can be used for sensitivity
- **addRegistry** – bool - whether to add to the registry or not

__repr__()

Return repr(self).

extent(*includeBoundingSolid=True*)

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

class pyg4ometry.geant4.**_OverlapType**

protrusion = 1

overlap = 2

coplanar = 3

class pyg4ometry.geant4.**ReplicaVolume**(*name*, *logicalVolume*, *motherVolume*, *axis*, *nreplicas*, *width*, *offset=0*, *registry=None*, *addRegistry=True*, *wunit='mm'*, *ounit='mm'*)

Bases: *pyg4ometry.geant4.PhysicalVolume.PhysicalVolume*

ReplicaVolume: G4PVReplica

Parameters

- **name** – of physical volume
- **logical** – volume to be placed
- **mother** – logical volume,
- **axis** – kXAxis,kYAxis,kZAxis,kRho,kPhi
- **ncopies** – number of replicas
- **width** – spacing between replicas along axis
- **offset** – of grid

class Axis

kXAxis = 1

kYAxis = 2

kZAxis = 3

kRho = 4

kPhi = 5

GetAxisName()

_checkInternalOverlaps(*debugIO=False*, *nOverlapsDetected=[0]*)

Check if there are overlaps with the nominal mother volume. ie it possible to provide an incorrect mother volume / logical volume and parameterisation.

createReplicaMeshes()

getPhysicalVolumes()

return a list of temporary (ie not added to the relevant registry) PhysicalVolume instances with appropriate transforms including any daughter ReplicaVolumes.

The exception is for kRho axis where new unique solids and logical volumes are required. Therefore, these are added to the registry and inadvertently to the mother LV as PVS.

__repr__()

extent(*includeBoundingSolid=True*)

class pyg4ometry.geant4.**_ReplicaVolume**(*name*, *logicalVolume*, *motherVolume*, *axis*, *nreplicas*, *width*, *offset=0*, *registry=None*, *addRegistry=True*, *wunit='mm'*, *ounit='mm'*)

Bases: *pyg4ometry.geant4.PhysicalVolume.PhysicalVolume*

ReplicaVolume: G4PVReplica

Parameters

- **name** – of physical volume
- **logical** – volume to be placed
- **mother** – logical volume,
- **axis** – kXAxis,kYAxis,kZAxis,kRho,kPhi
- **ncopies** – number of replicas
- **width** – spacing between replicas along axis
- **offset** – of grid

class Axis

kXAxis = 1

kYAxis = 2

kZAxis = 3

kRho = 4

kPhi = 5

GetAxisName()

_checkInternalOverlaps(*debugIO=False*, *nOverlapsDetected=[0]*)

Check if there are overlaps with the nominal mother volume. ie it possible to provide an incorrect mother volume / logical volume and parameterisation.

createReplicaMeshes()

getPhysicalVolumes()

return a list of temporary (ie not added to the relevant registry) PhysicalVolume instances with appropriate transforms including any daughter ReplicaVolumes.

The exception is for kRho axis where new unique solids and logical volumes are required. Therefore, these are added to the registry and inadvertently to the mother LV as PVS.

__repr__()

extent(*includeBoundingSolid=True*)

class pyg4ometry.geant4.**ParameterisedVolume**(*name*, *logicalVolume*, *motherVolume*, *ncopies*, *paramData*, *transforms*, *registry=None*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.ReplicaVolume.ReplicaVolume*

ParametrisedVolume :param name: of parametrised volume :type name: str :param logical: volume to be placed :type logical: logicalVolume :param mother: volume logical volume :type mother: logicalVolume :param ncopies: number of parametrised volumes :type ncopies: int

class BoxDimensions(*pX*, *pY*, *pZ*, *lunit='mm'*)

class TubeDimensions(*pRMin*, *pRMax*, *pDz*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class ConeDimensions(*pRMin1*, *pRMax1*, *pRMin2*, *pRMax2*, *pDz*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class OrbDimensions(*pRMax*, *lunit='mm'*)

class SphereDimensions(*pRMin*, *pRMax*, *pSPhi*, *pDPhi*, *pSTheta*, *pDTheta*, *lunit='mm'*, *aunit='rad'*)

class TorusDimensions(*pRMin*, *pRMax*, *pRTor*, *pSPhi*, *pDPhi*, *lunit='mm'*, *aunit='rad'*)

class HypeDimensions(*innerRadius*, *outerRadius*, *innerStereo*, *outerStereo*, *lenZ*, *lunit='mm'*, *aunit='rad'*)

class ParaDimensions(*pX*, *pY*, *pZ*, *pAlpha*, *pTheta*, *pPhi*, *lunit='mm'*, *aunit='rad'*)

class TrdDimensions(*pX1*, *pX2*, *pY1*, *pY2*, *pZ*, *lunit='mm'*)

class TrapDimensions(*pDz*, *pTheta*, *pDPhi*, *pDy1*, *pDx1*, *pDx2*, *pAlp1*, *pDy2*, *pDx3*, *pDx4*, *pAlp2*, *lunit='mm'*, *aunit='rad'*)

class PolyconeDimensions(*pSPhi*, *pDPhi*, *pZpl*, *pRMin*, *pRMax*, *lunit='mm'*, *aunit='rad'*)

class PolyhedraDimensions(*pSPhi*, *pDPhi*, *numSide*, *pZpl*, *pRMin*, *pRMax*, *lunit='mm'*, *aunit='rad'*)

class EllipsoidDimensions(*pxSemiAxis*, *pySemiAxis*, *pzSemiAxis*, *pzBottomCut*, *pzTopCut*, *lunit='mm'*)

createParameterisedMeshes()

__repr__()

extent(*includeBoundingSolid=True*)

class pyg4ometry.geant4.**_PhysicalVolume**(*rotation*, *position*, *logicalVolume*, *name*, *motherVolume*, *registry=None*, *copyNumber=0*, *addRegistry=True*, *scale=None*)

PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

Parameters

- **rotation** – [float,float,float] - rotations about x,y,z axes of mother volume

- **position** – [float,float,float] - translation with respect to mother volume
- **logicalVolume** – *pyg4ometry.geant4.LogicalVolume* - instance to place
- **name** – str - name of this placement
- **motherVolume** – *pyg4ometry.geant4.LogicalVolume* - mother volume to place into
- **registry** – *pyg4ometry.geant4.Registry* - registry to register to
- **copyNumber** – int - copy number of the placement that can be used for sensitivity
- **addRegistry** – bool - whether to add to the registry or not

__repr__()

Return repr(self).

extent(*includeBoundingSolid=True*)

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

class pyg4ometry.geant4.**DivisionVolume**(*name*, *logicalVolume*, *motherVolume*, *axis*, *ndivisions=-1*, *width=-1*, *offset=0*, *registry=None*, *addRegistry=True*, *unit='mm'*)

Bases: *pyg4ometry.geant4.PhysicalVolume.PhysicalVolume*

DivisionVolume: G4PVDivision

Parameters

- **name** – of physical volume
- **logical** – volume to be placed
- **mother** – logical volume,
- **axis** – kXAxis,kYAxis,kZAxis,kRho,kPhi
- **ncopies** – number of replicas
- **width** – spacing between replicas along axis
- **offset** – of grid

class Axis

kXAxis = 1

kYAxis = 2

kZAxis = 3

kRho = 4

kPhi = 5

getMotherSize()

checkAxis(*allowed_axes*)

divideBox(*offset*, *width*, *ndiv*)

divideTubs(*offset*, *width*, *ndiv*)

divideCons(*offset*, *width*, *ndiv*)

dividePara(*offset*, *width*, *ndiv*)

divideTrd(*offset*, *width*, *ndiv*)

dividePolycone(*offset*, *width*, *ndiv*)

dividePolyhedra(*offset*, *width*, *ndiv*)

createDivisionMeshes()

__repr__()

extent(*includeBoundingSolid=True*)

class pyg4ometry.geant4.**LogicalVolume**(*solid*, *material*, *name*, *registry=None*, *addRegistry=True*, ***kwargs*)

LogicalVolume : G4LogicalVolume

Parameters

- **solid** –
- **material** –
- **name** (*str*) –
- **registry** –
- **addRegistry** (*bool*) –

__repr__()

Return repr(self).

reMesh(*recursive=False*)

Regenerate the visualisation for this logical volume. Required if the geometry is modified and overlap checking is subsequently required or revisualisation.

add(*physicalVolume*)

Add physical volume to this logicalVolume

Parameters

physicalVolume (PhysicalVolume, ReplicaVolume, ParameterisedVolume, DivisionVolume) – physical volume to add

addBDSIMObject(*bdsimobject*)

_getPhysicalDaughterMesh(*pv*, *warn=True*)

Return a (cloned from the lv) mesh of a given pv with rotation,scale, translation evaluated.

cullDaughtersOutsideSolid(*solid*, *rotation=None*, *position=None*)

Given a solid with a placement rotation and position inside this logical volume, remove (cull) any daughters that would not lie entirely within it. The rotation and position are applied to the solid w.r.t. the frame of this logical volume.

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv

transformDaughters(*rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Transform the daugter volumes (without clipping)

Parameters

- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

replaceSolid(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Replace the outer solid with optional position and rotation

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)

clipGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*, *replace=False*, *depth=0*, *solidUsageCount=_defaultdict(int)*, *lvUsageCount=_defaultdict(int)*)

Clip the geometry to newSolid, placed with rotation and position.

Parameters

- **newSolid** (*pyg4ometry.geant4.solid*) – object to clip the geometry to
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan angles for rotation of the solid w.r.t. this lv
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation of the solid w.r.t. this lv
- **runit** (*str*) – angular unit for rotation (rad,deg)
- **punit** (*str*) – length unit for position (m,mm,km)
- **replace** (*bool*) – replace the outer solid or not
- **depth** (*int*) – recursion depth (DO NOT USE)
- **solidUsageCount** (*defaultdict*) – solid name dictionary for replacement recursion (DO NOT USE)
- **lvUsageCount** (*defaultdict*) – lv name dictionary for replacement recursion (DO NOT USE)

changeSolidAndTrimGeometry(*newSolid*, *rotation=(0, 0, 0)*, *position=(0, 0, 0)*, *runit='rad'*, *punit='mm'*)

Change the solid of this logical volume, remove any daughters that will lie outside it, and form new Boolean intersection solids for any daughters that cross the boundary (intersect) it. The rotation and translation are with respect to the original frame and all daughters will now be replaced with respect to the new frame.

Therefore, that same rotation and translation should be use to re-place this logical volume if desired. The default is none though, so the frame would nominally remain the same.

Parameters

- **newSolid** (*any of pyg4ometry.geant4.solid*) – new solid to use for this logical volume
- **rotation** (*list(float, float, float) or None - 3 values in radians*) – Tait-Bryan rotation for the new solid w.r.t. old frame (i.e. current lv)
- **position** (*list(float, float, float) or None - 3 values in mm*) – translation for the new solid w.r.t. old frame (i.e. current lv)

checkOverlaps(*recursive=False*, *coplanar=False*, *debugIO=False*, *printOut=True*, *nOverlapsDetected=[0]*)

Check based on the meshes in each logical volume if there are any geometrical overlaps. By default, overlaps are checked between daughter volumes and with the mother volume itself (protrusion). Coplanar overlaps may also be checked (default on).

Print out will be given for any overlaps detected and the visualiser will show the colour coded overlaps.

Parameters

- **recursive** – bool - Whether to descend into the daughter volumes and check their contents also.
- **coplanar** – bool - Whether to check for coplanar overlaps
- **debugIO** – bool - Print out for every check made
- **printOut** – bool - (internal) Whether to print out a summary of N overlaps detected
- **nOverlapsDetected** – [int] - (internal) counter for recursion - ignore

setSolid(*solid*)

Set (replace) the outer solid. Does not change the placement of the daughters in the volume. If there is a transformation then use replaceSolid

makeSolidTessellated()

Make solid tesselated. Sometimes useful when a boolean cannot be visualised in Geant4

addAuxiliaryInfo(*auxiliary*)

Add auxilary information to logical volume :param auxiliary: auxiliary information for the logical volume :type auxiliary: tuple or list

extent(*includeBoundingSolid=False*)

Compute the axis aligned extent of the logical volume.

Parameters
includeBoundingSolid (*bool*) – Include the bounding solid or not

depth(*depth=0*)

Depth for LV-PV tree

clipSolid(*lengthSafety=1e-06*)

Assuming the solid of this LV is a Box, reduce its dimensions and re-placement all daughters to reduce the box to the minimum (axis-aligned) bounding box. This updates the dimensions of the box and the translation of each daughter physical volume.

Parameters
lengthSafety (*float*) – safety length

makeLogicalPhysicalNameSets()

Return a set of logical names and physical names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc.

findLogicalByName(*name*)

Return a list of LogicalVolume instances used inside this logical volume as daughters (at any level inside) with the given name.

Parameters
name (*str*) – lv name

makeMaterialNameSet()

Return a set of material names used in this logical volume and any daughters. This is built up recursively by checking all daughters etc etc.

assemblyVolume(*materialName='G4_AIR0x7f8441173ac0'*)

Return an assembly volume of this logical volume, in effect removing the solid and material of this logical volume, but retaining all of the relative daughter placements.

makeWorldVolume(*worldMaterial='G4_Galactic'*)

This will create a container box according to the extents of this logical volume: an axis-aligned bounding-box. It will be filled with the given material (predefined by name) and assigned as the world volume (outermost) of the registry according to this logical volume.

dumpStructure(*depth=0*)

class pyg4ometry.geant4.**SurfaceBase**(*name*, *type*, *surface_property*, *registry*, *addRegistry*)

_chkType(*obj*, *t*, *prop*)

class pyg4ometry.geant4.**SkinSurface**(*name*, *volumeref*, *surface_property*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.SurfaceBase.SurfaceBase*

__repr__()

class pyg4ometry.geant4.**PhysicalVolume**(*rotation*, *position*, *logicalVolume*, *name*, *motherVolume*, *registry=None*, *copyNumber=0*, *addRegistry=True*, *scale=None*)

PhysicalVolume : G4VPhysicalVolume, G4PVPlacement

Parameters

- **rotation** – [float,float,float] - rotations about x,y,z axes of mother volume
- **position** – [float,float,float] - translation with respect to mother volume
- **logicalVolume** – *pyg4ometry.geant4.LogicalVolume* - instance to place
- **name** – str - name of this placement
- **motherVolume** – *pyg4ometry.geant4.LogicalVolume* - mother volume to place into
- **registry** – *pyg4ometry.geant4.Registry* - registry to register to
- **copyNumber** – int - copy number of the placement that can be used for sensitivity
- **addRegistry** – bool - whether to add to the registry or not

__repr__()

Return repr(self).

extent(*includeBoundingSolid=True*)

getAABBMesh()

return CSG.core (symmetric around the origin) axis aligned bounding box mesh

class pyg4ometry.geant4.**SurfaceBase**(*name*, *type*, *surface_property*, *registry*, *addRegistry*)

_chkType(*obj*, *t*, *prop*)

class pyg4ometry.geant4.**BorderSurface**(*name*, *physref1*, *physref2*, *surface_property*, *registry*, *addRegistry=True*)

Bases: *pyg4ometry.geant4.SurfaceBase.SurfaceBase*

__repr__()

pyg4ometry.geant4.**solidName**(*var*)

pyg4ometry.geant4.**removeprefix**(*string*, *prefix*, */*)

Parameters

- **string** (*str*) –
- **prefix** (*str*) –

Return type

str

class pyg4ometry.geant4.**Registry**

Object to store geometry for input and output. All of the pyg4ometry classes can be used without storing them in the Registry. The registry is used to write the GDML output file. A registry needs to be used in conjunction with GDML Define objects for evaluation of expressions.

clear()

Empty all internal structures

getExpressionParser()

registerSolidEdit(*solid*)

addMaterial(*material*, *dontWarnIfAlreadyAdded=False*)

Register a material with this registry.

Parameters

material (*Material*) – Material object for storage

transferMaterial(*material*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a material to this registry. This can operate on a Material, an Isotope and an Element instance.

addSolid(*solid*)

Register a solid with this registry.

Parameters

solid (*One of the geant4 solids*) – Solid object for storage

transferSolid(*solid*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a solid to this registry. Doesn't handle any members' transferal - only the solid itself.

Parameters

solid (*One of the geant4 solids*) – Solid object for storage

addLogicalVolume(*volume*)

Register a logical volume with this registry. Also accepts Assembly Volumes.

Parameters
volume (LogicalVolume) – LogicalVolume object for storage

transferLogicalVolume(*volume*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a logical volume to this registry. Doesn't handle any members' transferal - only the logical volume itself.

addPhysicalVolume(*volume*)

Registry a physical volume with this registry.

Parameters
volume (PhysicalVolume) – PhysicalVolume object for storage

transferPhysicalVolume(*volume*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a physical volume to this registry. Doesn't handle any members' transferal - only the physical volume itself.

addSurface(*surface*)

Register a surface with this registry.

Parameters
surface (pyg4ometry.geant4.BorderSurface *or* pyg4ometry.geant4.SkinSurface) – Surface

transferSurface(*surface*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a surface to this registry.

addAuxiliary(*auxiliary*)

addDefine(*define*)

Register a define with this registry.

Parameters
define (Constant, Quantity, Variable, Matrix) – Definition object for storage

transferDefine(*define*, *incrementRenameDict={}*, *userRenameDict=None*)

Transfer a single define from another registry to this one. No checking on previous registry or not.

transferDefines(*var*, *otherRegistry*, *incrementRenameDict={}*, *userRenameDict=None*)

This function tolerates all types of defines including vector ones.

Transfer defines from one registry to another recursively. A define may not be part of the old registry so won't be added to this one. A define may be a vector or composite and its 'bits' may be in the (old) registry so each part should be checked.

In "3x + 2", "x" would be a variable". In "3.5*2" there would be no variables.

setWorld(*worldIn*)

The argument can either be the name of logical volume of the world or the pyg4ometry.geant4.LogicalVolume instance of the world volume. The term world is used to refer to the outermost volume of the hierarchy.

setWorldVolume(*worldIn*)

An alias for some of us who can't remember.

_orderMaterialList(*materials*, *materials_ordered=[]*)

orderMaterials()

Need to have a ordered list of all material entities for writing to GDML. GDML needs to have the isotopes/elements/materials defined in use order

orderLogicalVolumes(*lvName*, *first=True*)

Need to have an ordered list from most basic (solid) object upto physical/logical volumes for writing to GDML. GDML needs to have the solids/booleans/volumes defined in order

addVolumeRecursive(*volume*, *collapseAssemblies=False*, *incrementRenameDict=None*, *userRenameDict=None*)

Transfer a volume hierarchy to this registry. Any objects that had a registry set to another will be set to this one and will be owned by it effectively. :param volume: PhysicalVolume or LogicalVolume or AssemblyVolume. :type volume: pyg4ometry.geant4.PhysicalVolume, pyg4ometry.geant4.LogicalVolume, pyg4ometry.geant4.AssemblyVolume. :param collapseAssemblies: if True, daughters of AssemblyVolume's will be attached directly to the mother of the assembly and the AssemblyVolume itself will be eliminated from the geometry tree :param incrementRenameDict: ignore - dictionary used internally for potentially incrementing names :param userRenameDict: a dictionary of find/replace regex strings to be used to rename volumes/materials/etc.

In the case where some object or variable has a name (e.g. 'X') that already exists in this registry, it will be incremented to 'X_1'.

addAndCollapseAssemblyVolumeRecursive(*assemblyPV*, *motherVol*, *positions*, *rotations*, *scales*, *names*, *incrementRenameDict*, *userRenameDict*)

Transfer and collapse an AssemblyVolume hierarchy to this registry. Daughter volumes are copied, renamed, and attached to the supplied mother LogicalVolume with the correct position/rotation. Any objects that had a registry set to another will be set to this one and will be owned by it effectively. :param assemblyPV: the PhysicalVolume that places an AssemblyVolume :param motherVol: the LogicalVolume to which all daughters of the AssemblyVolume (including any daughters of nested AssemblyVolume's) should be attached :param positions: a list (initially empty) to hold position objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param rotations: a list (initially empty) to hold rotation objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param scales: a list (initially empty) to hold scale objects for each AssemblyVolume in a hierarchy of nested assemblies (used to correctly set the transformation of daughters within the motherVol) :param names: a list (initially empty) to hold the names of each AssemblyVolume in a hierarchy of nested assemblies (used to set unique names for the daughters within the motherVol) :param incrementRenameDict: ignore - dictionary used internally for potentially incrementing names :param userRenameDict: a dictionary of find/replace regex strings to be used to rename volumes/materials/etc.

transferSolidDefines(*solid*, *incrementRenameDict={}*, *userRenameDict=None*)

For each parameter in a given solid (unique to each) check if it's a define and transfer that over.

volumeTree(*lvName*)

Not sure what this method is used for

solidTree(*solidName*)

Not sure what this method is used for

getWorldVolume()

printStats()

structureAnalysis(*lv_name=None*, *debug=False*, *level=0*, *df=None*)

_findDictByName(*dic*, *nameFragment*)

Find a object which name matches (or partially matches) nameFragment, returns a list of objects

findSolidByName(*nameFragment='box'*)

Find a solid which name matches (or partially matches) nameFragment, returns a list of solids

findMaterialByName(*nameFragment='G4_AIR'*)

Find a material which name matches (or partially matches) nameFragment, returns a list of materials

findLogicalVolumeByName(*nameFragment='World'*)

Find a logical volume which name matches (or partially matches) nameFragment, returns a list of LogicalVolumes

findPhysicalVolumeByName(*nameFragment*)

Find a physical volume which name matches (or partially matches) nameFragment, returns a list of LogicalVolumes

class pyg4ometry.geant4.**GeometryComplexityInformation**

printSummary(*boolDepthLimit=3*)

pyg4ometry.geant4.**AnalyseGeometryComplexity**(*logicalVolume*)

Analyse a geometry tree starting from a logical volume. Produces an instance of *GeometryComplexityInformation* with summary information. Provides:

- count per solid type
- number of daughters per logical volume
- dictionary of N daughters for each logical volume name
- depth count of Boolean solids

ie a Boolean of a Boolean returns 2, a Boolean of two primitives returns 1

- a dictionary of boolean depth for each logical volume name

Example:

```
info = AnalyseGeometryComplexity(lv)
info.printSummary()
```

pyg4ometry.geant4.**_UpdateComplexity**(*lv*, *info*)

pyg4ometry.geant4.**AnalyseGeometryStructure**(*registry*, *lv_name=None*, *debug=False*, *level=0*, *df=None*)

Produce a pandas dataframe representing the structure of the geometry.

pyg4ometry.geant4.**DumpGeometryStructureTree**(*lv*, *depth=0*)

pyg4ometry.geant4.**_nistMaterialDict**

pyg4ometry.geant4.**_nistMaterialList**

pyg4ometry.geant4.**_nistElementZToName**

pyg4ometry.geant4.**getNistMaterialDict**()

pyg4ometry.geant4.**getNistMaterialList**()

pyg4ometry.geant4.**getNistElementZToName**()

pyg4ometry.geant4._getClassVariables(*obj*)

pyg4ometry.geant4._makeNISTCompoundList()

pyg4ometry.geant4._safeName(*name*)

pyg4ometry.geant4.loadNISTMaterialDict()

pyg4ometry.geant4.nist_materials_name_lookup(*name*)

pyg4ometry.geant4.nist_materials_z_lookup(*z*)

pyg4ometry.geant4.nist_element_2geant4Element(*name*, *reg=None*)

This returns and instance of either ElementSimple or ElementIsotopeMixture.

pyg4ometry.geant4.nist_material_2geant4Material(*name*, *reg=None*)

pyg4ometry.geant4.MaterialPredefined(*name*, *registry=None*)

Proxy method to construct a NIST compound material - this is just a handle as nothing needs to be additionally defined for a NIST compound. A check is performed on the name to ensure it is a valid NIST specifier.

Inputs:

name - string

pyg4ometry.geant4.MaterialArbitrary(*name*, *registry=None*)

Just a name of a material. WARNING: It is left to the user to ensure that the name is valid.

Inputs:

name - string

pyg4ometry.geant4.MaterialSingleElement(*name*, *atomic_number*, *atomic_weight*, *density*, *registry=None*, *tolerateZeroDensity=False*)

Proxy method to construct a simple material - full description of the element contained is contained in one definition

Inputs:

name - string atomic_number - int, number of protons, commonly known as 'Z' atomic_weight - molar weight in g/mole, commonly known as 'A' density - float, material density in g/cm3

pyg4ometry.geant4.MaterialCompound(*name*, *density*, *number_of_components*, *registry=None*, *tolerateZeroDensity=False*, *state=None*)

Proxy method to construct a composite material - can be any mixture of Elements and/or Materials

Inputs:

name - string density - float, material density in g/cm3 number_of_components - int, number of components in the mixture

pyg4ometry.geant4.ElementSimple(*name*, *symbol*, *Z*, *A*, *registry=None*)

Proxy method to construct a simple element - full description of the element contained is contained in one definition

Inputs:

name - string symbol - string, chemical formula of the compound Z - int, Atomic number A - float, mass number

pyg4ometry.geant4.ElementIsotopeMixture(*name*, *symbol*, *n_comp*, *registry=None*, *state=None*)

Proxy method to construct a composite element - a mixture of predefined isotopes

Inputs:
name - string symbol - string, chemical formula of the compound n_comp - int, number of isotope components

class pyg4ometry.geant4.**MaterialBase**(*name*, *state=None*, *registry=None*)

_addToRegistry()

get_material_oject(*material*)

set_registry(*registry*, *dontWarnIfAlreadyAdded=False*)

set_state(*state*)

__repr__()
Return repr(self).

class pyg4ometry.geant4.**Material**(***kwargs*)
Bases: *MaterialBase*

This class provides an interface to GDML material definitions.

Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

MaterialSingleElement MaterialCompound MaterialPredefined

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string density - float atomic_number - int atomic_weight - float number_of_components - int state - string pressure - float pressure_unit - string temperature - float temperature_unit - string

property state_variables

add_element_massfraction(*element*, *massfraction*)
Add an element as a component to a material as a fraction of the material mass. Can only add elements to materials defined as composite.

Inputs:
element - pyg4ometry.geant4.Material.Element instance massfraction - float, 0.0 < massfraction <= 1.0

add_element_natoms(*element*, *natoms*)
Add an element as a component to a material as a number of atoms in the material molecule. Can only add elements to materials defined as composite.

Inputs:
element - pyg4ometry.geant4.Material.Element instance natoms - int, number of atoms in the compound molecule

add_material(*material*, *fractionmass*)
Add a material as a component to another material (mixture) as a fraction of the mixture mass. Can only add new materials to materials defined as composite.

Inputs:
material - pyg4ometry.geant4.Material.Material instance massfraction - float, 0.0 < massfraction <= 1.0

set_pressure(*value*, *unit='pascal'*)

set_temperature(*value*, *unit='K'*)

__str__()

Return str(self).

addProperty(*name*, *matrix*)

Add a material property from a matrix.

Parameters

- **name** (*str*) – key of the material property
- **matrix** (Matrix) – matrix defining the value(s) of the property

addVecProperty(*name*, *e*, *v*, *eunit='eV'*, *vunit=''*)

Add a property from an energy and a value vector to this object.

Parameters

- **name** (*str*) – key of property
- **e** (*list or numpy.array - shape (1,)*) – energy list/vector in units of eunit
- **v** (*list or numpy.array - shape (1,)*) – value list/vector in units of vunit
- **eunit** (*str*) – unit for the energy vector (default: eV)
- **vunit** (*str*) – unit for the value vector (default: unitless)

addConstProperty(*name*, *value*, *vunit=''*)

Add a constant scalar property to this object.

Parameters

- **name** (*str*) – key of property
- **value** (*str,float,int*) – constant value for this property
- **vunit** (*str*) – unit for the value vector (default: unitless)

class pyg4ometry.geant4.**Element**(***kwargs*)

Bases: *MaterialBase*

This class provides an interface to GDML material definitions. Because of the different options for constructing a material instance the constructor is kwarg only. Proxy methods are provided to instantiate particular types of material. Those proxy methods are:

ElementSimple ElementIsotopeMixture

It is possible to instantiate a material directly through kwargs. The possible kwargs are (but note some are mutually exclusive): name - string symbol - string Z - int A - int n_comp - int

add_isotope(*isotope*, *abundance*)

Add an isotope as a component to an element as an abundance fraction in the element.

Inputs:

element - pyg4ometry.geant4.Material.Isotope instance abundance - float, 0.0 < abundance <= 1.0

class pyg4ometry.geant4.**Isotope**(*name*, *Z*, *N*, *a*, *registry=None*)

Bases: *MaterialBase*

This class that handles isotopes as components of composite materials. An element can be defined as a mixture of isotopes.

Inputs:

name - string Z - int, atomic number N - int, mass number a - float, molar weight in g/mole

pyg4ometry.gui

Submodules

pyg4ometry.gui.GeometryModel

Module Contents

Classes

GeometryModel

Functions

extensionFromPath(fileName)
nameFromPath(fileName)

pyg4ometry.gui.GeometryModel.**extensionFromPath**(*fileName*)

pyg4ometry.gui.GeometryModel.**nameFromPath**(*fileName*)

class pyg4ometry.gui.GeometryModel.**GeometryModel**

createNewRegistry(*name*, *type*)

loadNewRegistry(*fileName*)

pyg4ometry.gui.MainWindow

Module Contents

Classes

MainWindow

Functions

main()	

class pyg4ometry.gui.MainWindow.**MainWindow**

Bases: PyQt5.QtWidgets.QMainWindow

initModel()

initUI()

slotModelChanged(*signal*)

setDisplayRenderer(*iRenderer*)

openFileNameDialog()

saveFileDialog()

pyg4ometry.gui.MainWindow.**main**()

pyg4ometry.gui.QVTKRenderWindowInteractor

Module Contents

Classes

QVTKRenderWindowInteractor	A QVTKRenderWindowInteractor for Python and Qt. Uses a

Functions

QVTKRenderWidgetConeExample()	A simple example that uses the QVTKRenderWindow-Interactor class.
_qt_key_to_key_sym(key)	Convert a Qt key into a vtk keysym.

Attributes

PyQtImpl
QVTKRWIBase
_keysyms

pyg4ometry.gui.QVTKRenderWindowInteractor.**PyQtImpl**

pyg4ometry.gui.QVTKRenderWindowInteractor.**QVTKRWIBase**

class pyg4ometry.gui.QVTKRenderWindowInteractor.**QVTKRenderWindowInteractor**(*parent=None*, ***kw*)

Bases: PyQt5.QtWidgets.QWidget

A QVTKRenderWindowInteractor for Python and Qt. Uses a vtkGenericRenderWindowInteractor to handle the interactions. Use GetRenderWindow() to get the vtkRenderWindow. Create with the keyword stereo=1 in order to generate a stereo-capable window.

The user interface is summarized in vtkInteractorStyle.h:

- Keypress j / Keypress t: toggle between joystick (position

sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.

- Keypress c / Keypress o: toggle between camera and object

(actor) modes. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.

- Button 1: rotate the camera around its focal point (if camera

mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.

- Button 2: pan the camera (if camera mode) or translate the actor

(if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with 2-button mice, pan is defined as <Shift>-Button 1.)

- Button 3: zoom the camera (if camera mode) or scale the actor

(if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.

- Keypress 3: toggle the render window into and out of stereo

mode. By default, red-blue stereo pairs are created. Some systems support Crystal Eyes LCD stereo glasses; you have to invoke SetStereoTypeToCrystalEyes() on the rendering window. Note: to use stereo you also need to pass a stereo=1 keyword argument to the constructor.

- Keypress e: exit the application.

- Keypress f: fly to the picked point

- Keypress p: perform a pick operation. The render window interactor

has an internal instance of vtkCellPicker that it uses to pick.

- Keypress r: reset the camera view along the current view

direction. Centers the actors and moves the camera so that all actors are visible.

- Keypress s: modify the representation of all actors so that they

are surfaces.

- Keypress u: invoke the user-defined function. Typically, this

keypress will bring up an interactor that you can type commands in.

- Keypress w: modify the representation of all actors so that they

are wireframe.

_CURSOR_MAP

__getattr__(*attr*)

Makes the object behave like a vtkGenericRenderWindowInteractor

Finalize()

Call internal cleanup method on VTK objects

CreateTimer(*obj*, *evt*)

DestroyTimer(*obj*, *evt*)

TimerEvent()

CursorChangedEvent(*obj*, *evt*)

Called when the CursorChangedEvent fires on the render window.

HideCursor()

Hides the cursor.

ShowCursor()

Shows the cursor.

closeEvent(*evt*)

sizeHint()

paintEngine()

paintEvent(*ev*)

resizeEvent(*ev*)

_GetCtrlShift(*ev*)

static _getPixelRatio()

_setEventInformation(*x*, *y*, *ctrl*, *shift*, *key*, *repeat=0*, *keysum=None*)

enterEvent(*ev*)

leaveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

mouseMoveEvent(*ev*)

keyPressEvent(*ev*)

keyReleaseEvent(*ev*)

wheelEvent(*ev*)

GetRenderWindow()

Render()

pyg4ometry.gui.QVTKRenderWindowInteractor.**QVTKRenderWidgetConeExample**()

A simple example that uses the QVTKRenderWindowInteractor class.

pyg4ometry.gui.QVTKRenderWindowInteractor.**_keysyms**

pyg4ometry.gui.QVTKRenderWindowInteractor.**_qt_key_to_key_sym**(*key*)

Convert a Qt key into a vtk keysym.

This is essentially copied from the c++ implementation in GUISupport/Qt/QVTKInteractorAdapter.cxx.

pyg4ometry.gui.example1

Module Contents

Classes

QVTKRenderWindowInteractor	A QVTKRenderWindowInteractor for Python and Qt. Uses a

Functions

QVTKRenderWidgetConeExample()	A simple example that uses the QVTKRenderWindow-Interactor class.
_qt_key_to_key_sym(key)	Convert a Qt key into a vtk keysym.

Attributes

PyQtImpl
QVTKRWIBase
QVTKRWIBase
PyQtImpl
QVTKRWIBaseClass
_keysyms

pyg4ometry.gui.example1.**PyQtImpl**

pyg4ometry.gui.example1.**QVTKRWIBase = 'QWidget'**

pyg4ometry.gui.example1.**QVTKRWIBase**

pyg4ometry.gui.example1.**PyQtImpl = 'PyQt5'**

pyg4ometry.gui.example1.**QVTKRWIBaseClass**

class pyg4ometry.gui.example1.**QVTKRenderWindowInteractor**(*parent=None*, ***kw*)

Bases: *QVTKRWIBaseClass*

A QVTKRenderWindowInteractor for Python and Qt. Uses a vtkGenericRenderWindowInteractor to handle the interactions. Use GetRenderWindow() to get the vtkRenderWindow. Create with the keyword stereo=1 in order to generate a stereo-capable window.

The user interface is summarized in vtkInteractorStyle.h:

- Keypress j / Keypress t: toggle between joystick (position

sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.

- Keypress c / Keypress o: toggle between camera and object

(actor) modes. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.

- Button 1: rotate the camera around its focal point (if camera

mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.

- Button 2: pan the camera (if camera mode) or translate the actor

(if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with 2-button mice, pan is defined as <Shift>-Button 1.)

- Button 3: zoom the camera (if camera mode) or scale the actor

(if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.

- Keypress 3: toggle the render window into and out of stereo

mode. By default, red-blue stereo pairs are created. Some systems support Crystal Eyes LCD stereo glasses; you have to invoke SetStereoTypeToCrystalEyes() on the rendering window. Note: to use stereo you also need to pass a stereo=1 keyword argument to the constructor.

- Keypress e: exit the application.

- Keypress f: fly to the picked point

- Keypress p: perform a pick operation. The render window interactor

has an internal instance of vtkCellPicker that it uses to pick.

- Keypress r: reset the camera view along the current view

direction. Centers the actors and moves the camera so that all actors are visible.

- Keypress s: modify the representation of all actors so that they

are surfaces.

- Keypress u: invoke the user-defined function. Typically, this

keypress will bring up an interactor that you can type commands in.

- Keypress w: modify the representation of all actors so that they

are wireframe.

_CURSOR_MAP

__getattr__(*attr*)
Makes the object behave like a vtkGenericRenderWindowInteractor

Finalize()
Call internal cleanup method on VTK objects

CreateTimer(*obj*, *evt*)

DestroyTimer(*obj*, *evt*)

TimerEvent()

CursorChangedEvent(*obj*, *evt*)
Called when the CursorChangedEvent fires on the render window.

HideCursor()
Hides the cursor.

ShowCursor()
Shows the cursor.

closeEvent(*evt*)

sizeHint()

paintEngine()

paintEvent(*ev*)

resizeEvent(*ev*)

_GetCtrlShift(*ev*)

static _getPixelRatio()

_setEventInformation(*x*, *y*, *ctrl*, *shift*, *key*, *repeat=0*, *keysum=None*)

enterEvent(*ev*)

leaveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

mouseMoveEvent(*ev*)

keyPressEvent(*ev*)

keyReleaseEvent(*ev*)

wheelEvent(*ev*)

GetRenderWindow()

Render()

pyg4ometry.gui.example1.**QVTKRenderWidgetConeExample**()

A simple example that uses the QVTKRenderWindowInteractor class.

pyg4ometry.gui.example1.**_keysyms**

pyg4ometry.gui.example1.**_qt_key_to_key_sym**(*key*)

Convert a Qt key into a vtk keysym.

This is essentially copied from the c++ implementation in GUISupport/Qt/QVTKInteractorAdapter.cxx.

Package Contents

Classes

QVTKRenderWindowInteractor	
GeometryModel	
MainWindow	
QVTKRenderWindowInteractor	
GeometryModel	

Functions

main()	
QVTKRenderWidgetConeExample()	A simple example that uses the QVTKRenderWindowInteractor class.
_qt_key_to_key_sym(key)	Convert a Qt key into a vtk keysym.
extensionFromPath(fileName)	
nameFromPath(fileName)	

Attributes

PyQtImpl
QVTKRWIBase
_keysyms

class pyg4ometry.gui.**QVTKRenderWindowInteractor**(*parent=None*, ***kw*)

Bases: PyQt5.QtWidgets.QWidget

A QVTKRenderWindowInteractor for Python and Qt. Uses a vtkGenericRenderWindowInteractor to handle the interactions. Use GetRenderWindow() to get the vtkRenderWindow. Create with the keyword stereo=1 in order to generate a stereo-capable window.

The user interface is summarized in vtkInteractorStyle.h:

- Keypress j / Keypress t: toggle between joystick (position

sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.

- Keypress c / Keypress o: toggle between camera and object

(actor) modes. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.

- Button 1: rotate the camera around its focal point (if camera

mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.

- Button 2: pan the camera (if camera mode) or translate the actor

(if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with 2-button mice, pan is defined as <Shift>-Button 1.)

- Button 3: zoom the camera (if camera mode) or scale the actor

(if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.

- Keypress 3: toggle the render window into and out of stereo

mode. By default, red-blue stereo pairs are created. Some systems support Crystal Eyes LCD stereo glasses; you have to invoke SetStereoTypeToCrystalEyes() on the rendering window. Note: to use stereo you also need to pass a stereo=1 keyword argument to the constructor.

- Keypress e: exit the application.
- Keypress f: fly to the picked point
- Keypress p: perform a pick operation. The render window interactor

has an internal instance of vtkCellPicker that it uses to pick.

- Keypress r: reset the camera view along the current view

direction. Centers the actors and moves the camera so that all actors are visible.

- Keypress s: modify the representation of all actors so that they

are surfaces.

- Keypress u: invoke the user-defined function. Typically, this

keypress will bring up an interactor that you can type commands in.

- Keypress w: modify the representation of all actors so that they

are wireframe.

_CURSOR_MAP

__getattr__(*attr*)

Makes the object behave like a vtkGenericRenderWindowInteractor

Finalize()

Call internal cleanup method on VTK objects

CreateTimer(*obj*, *evt*)

DestroyTimer(*obj*, *evt*)

TimerEvent()

CursorChangedEvent(*obj*, *evt*)

Called when the CursorChangedEvent fires on the render window.

HideCursor()

Hides the cursor.

ShowCursor()

Shows the cursor.

closeEvent(*evt*)

sizeHint()

paintEngine()

paintEvent(*ev*)

resizeEvent(*ev*)

_GetCtrlShift(*ev*)

static _getPixelRatio()

_setEventInformation(*x*, *y*, *ctrl*, *shift*, *key*, *repeat=0*, *keysum=None*)

enterEvent(*ev*)

leaveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

mouseMoveEvent(*ev*)

keyPressEvent(*ev*)

keyReleaseEvent(*ev*)

wheelEvent(*ev*)

GetRenderWindow()

Render()

class pyg4ometry.gui.**GeometryModel**

createNewRegistry(*name*, *type*)

loadNewRegistry(*fileName*)

class pyg4ometry.gui.**MainWindow**

Bases: PyQt5.QtWidgets.QMainWindow

initModel()

initUI()

slotModelChanged(*signal*)

setDisplayRenderer(*iRenderer*)

openFileNameDialog()

saveFileDialog()

pyg4ometry.gui.**main**()

pyg4ometry.gui.**PyQtImpl**

pyg4ometry.gui.**QVTKRWIBase**

class pyg4ometry.gui.**QVTKRenderWindowInteractor**(*parent=None*, ***kw*)

Bases: PyQt5.QtWidgets.QWidget

A QVTKRenderWindowInteractor for Python and Qt. Uses a vtkGenericRenderWindowInteractor to handle the interactions. Use GetRenderWindow() to get the vtkRenderWindow. Create with the keyword stereo=1 in order to generate a stereo-capable window.

The user interface is summarized in vtkInteractorStyle.h:

- Keypress j / Keypress t: toggle between joystick (position

sensitive) and trackball (motion sensitive) styles. In joystick style, motion occurs continuously as long as a mouse button is pressed. In trackball style, motion occurs when the mouse button is pressed and the mouse pointer moves.

- Keypress c / Keypress o: toggle between camera and object

(actor) modes. In camera mode, mouse events affect the camera position and focal point. In object mode, mouse events affect the actor that is under the mouse pointer.

- Button 1: rotate the camera around its focal point (if camera

mode) or rotate the actor around its origin (if actor mode). The rotation is in the direction defined from the center of the renderer's viewport towards the mouse position. In joystick mode, the magnitude of the rotation is determined by the distance the mouse is from the center of the render window.

- Button 2: pan the camera (if camera mode) or translate the actor

(if object mode). In joystick mode, the direction of pan or translation is from the center of the viewport towards the mouse position. In trackball mode, the direction of motion is the direction the mouse moves. (Note: with 2-button mice, pan is defined as <Shift>-Button 1.)

- Button 3: zoom the camera (if camera mode) or scale the actor

(if object mode). Zoom in/increase scale if the mouse position is in the top half of the viewport; zoom out/decrease scale if the mouse position is in the bottom half. In joystick mode, the amount of zoom is controlled by the distance of the mouse pointer from the horizontal centerline of the window.

- Keypress 3: toggle the render window into and out of stereo

mode. By default, red-blue stereo pairs are created. Some systems support Crystal Eyes LCD stereo glasses; you have to invoke SetStereoTypeToCrystalEyes() on the rendering window. Note: to use stereo you also need to pass a stereo=1 keyword argument to the constructor.

- Keypress e: exit the application.
- Keypress f: fly to the picked point
- Keypress p: perform a pick operation. The render window interactor

has an internal instance of vtkCellPicker that it uses to pick.

- Keypress r: reset the camera view along the current view

direction. Centers the actors and moves the camera so that all actors are visible.

- Keypress s: modify the representation of all actors so that they

are surfaces.

- Keypress u: invoke the user-defined function. Typically, this

keypress will bring up an interactor that you can type commands in.

- Keypress w: modify the representation of all actors so that they

are wireframe.

_CURSOR_MAP

__getattr__(*attr*)

Makes the object behave like a vtkGenericRenderWindowInteractor

Finalize()

Call internal cleanup method on VTK objects

CreateTimer(*obj*, *evt*)

DestroyTimer(*obj*, *evt*)

TimerEvent()

CursorChangedEvent(*obj*, *evt*)

Called when the CursorChangedEvent fires on the render window.

HideCursor()

Hides the cursor.

ShowCursor()

Shows the cursor.

closeEvent(*evt*)

sizeHint()

paintEngine()

paintEvent(*ev*)

resizeEvent(*ev*)

_GetCtrlShift(*ev*)

static _getPixelRatio()

_setEventInformation(*x*, *y*, *ctrl*, *shift*, *key*, *repeat=0*, *keysum=None*)

enterEvent(*ev*)

leaveEvent(*ev*)

mousePressEvent(*ev*)

mouseReleaseEvent(*ev*)

mouseMoveEvent(*ev*)

keyPressEvent(*ev*)

keyReleaseEvent(*ev*)

wheelEvent(*ev*)

GetRenderWindow()

Render()

pyg4ometry.gui.**QVTKRenderWidgetConeExample**()

A simple example that uses the QVTKRenderWindowInteractor class.

pyg4ometry.gui.**_keysyms**

pyg4ometry.gui.**_qt_key_to_key_sym**(*key*)

Convert a Qt key into a vtk keysym.

This is essentially copied from the c++ implementation in GUISupport/Qt/QVTKInteractorAdapter.cxx.

pyg4ometry.gui.**extensionFromPath**(*fileName*)

pyg4ometry.gui.**nameFromPath**(*fileName*)

class pyg4ometry.gui.**GeometryModel**

createNewRegistry(*name*, *type*)

loadNewRegistry(*fileName*)

pyg4ometry.io

Submodules

pyg4ometry.io.ROOTTGeo

Module Contents

Classes

Reader	

Functions

_isClose(a, b[, relativeTolerance, absoluteTolerance])	return True if values are close to each other. As judged by the
rootMatrix2pyg4ometry(matrix, reader)	
rootShape2pyg4ometry(shape, reader[, warnAboutBadShapes])	

pyg4ometry.io.ROOTTGeo.**_isClose**(*a*, *b*, *relativeTolerance=1e-05*, *absoluteTolerance=0.0*)

return True if values are close to each other. As judged by the absolute difference between a and b is greater than relative_tolerance times that difference. Implementation from PEP 485 (python > 3.5)

pyg4ometry.io.ROOTTGeo.**rootMatrix2pyg4ometry**(*matrix*, *reader*)

pyg4ometry.io.ROOTTGeo.**rootShape2pyg4ometry**(*shape*, *reader*, *warnAboutBadShapes=True*)

class pyg4ometry.io.ROOTTGeo.**Reader**(*fileName*, *upgradeVacuumToG4Galactic=True*, *solidsToTessellate=None*, *suffixSeparator='__'*, *warnAboutBadShapes=True*, *maximumDepth=1000000.0*, *dontLoadOverlapNodes=True*)

getRegistry()

load(*upgradeVacuumToG4Galactic=True*, *warnAboutBadShapes=True*, *dontLoadOverlapNodes=True*)

_ROOTMatStateToGeant4MatState(*rootMaterialState*)

Based on https://root.cern.ch/doc/master/classTGeoMaterial.html#a8b69c72f90711a29726087e029e39c61 enum TGeoMaterial::EGeoMaterialState

loadMaterials(*upgradeVacuumToG4Galactic*)

_makeG4Element(*rootElement*)

_makeG4Isotope(*rootIsotope*)

recurseVolumeTree(*volume*, *thisDepth*, *maximumDepth*, *warnAboutBadShapes=True*, *dontLoadOverlapNodes=True*)

pyg4ometry.misc

Submodules

pyg4ometry.misc.NestedSolids

Module Contents

Functions

NestedBoxes(nameBase, bx, by, bz, registry[, lunit, ...])	Creates a list of geant4.Box starting the bx, by, bz in size and each element is dx, dy, dz smaller

pyg4ometry.misc.NestedSolids.**NestedBoxes**(*nameBase*, *bx*, *by*, *bz*, *registry*, *lunit='mm'*, *dx=0*, *dy=0*, *dz=0*, *N=0*)

Creates a list of geant4.Box starting the bx, by, bz in size and each element is dx, dy, dz smaller for each iteration

Parameters

- **nameBase** (*str*) – name stub for solid
- **bx** (*float*) – box max x size
- **by** (*float*) – box max y size
- **bz** (*float*) – box max z size
- **registry** (Registry) – Registry object
- **lunit** (*str*) – length unit (mm/m etc)
- **dx** (*float*) – decrement in x for each iteration
- **dy** (*float*) – decrement in y for each iteration
- **dz** (*float*) – decrement in z for each iteration
- **N** (*int*) – number of iterations

pyg4ometry.misc.TestUtils

Module Contents

Functions

md5_file(fname)	
diffFiles(file1, file2)	
compareFilesWithAssert(file1, file2)	
compareNumericallyWithAssert(file1, file2)	
compareMeshInfo(meshInfo1, meshInfo2)	
compareGdmlNumericallyWithAssert(file1, file2)	

pyg4ometry.misc.TestUtils.**md5_file**(*fname*)

pyg4ometry.misc.TestUtils.**diffFiles**(*file1*, *file2*)

pyg4ometry.misc.TestUtils.**compareFilesWithAssert**(*file1*, *file2*)

pyg4ometry.misc.TestUtils.**compareNumericallyWithAssert**(*file1*, *file2*)

pyg4ometry.misc.TestUtils.**compareMeshInfo**(*meshInfo1*, *meshInfo2*)

pyg4ometry.misc.TestUtils.**compareGdmlNumericallyWithAssert**(*file1*, *file2*)

Package Contents

Functions

NestedBoxes(nameBase, bx, by, bz, registry[, lunit, ...])	Creates a list of geant4.Box starting the bx, by, bz in size and each element is dx, dy, dz smaller
md5_file(fname)	
diffFiles(file1, file2)	
compareFilesWithAssert(file1, file2)	
compareNumericallyWithAssert(file1, file2)	
compareMeshInfo(meshInfo1, meshInfo2)	
compareGdmlNumericallyWithAssert(file1, file2)	

pyg4ometry.misc.**NestedBoxes**(*nameBase*, *bx*, *by*, *bz*, *registry*, *lunit='mm'*, *dx=0*, *dy=0*, *dz=0*, *N=0*)

Creates a list of geant4.Box starting the bx, by, bz in size and each element is dx, dy, dz smaller for each iteration

Parameters

- **nameBase** (*str*) – name stub for solid
- **bx** (*float*) – box max x size
- **by** (*float*) – box max y size
- **bz** (*float*) – box max z size
- **registry** (Registry) – Registry object
- **lunit** (*str*) – length unit (mm/m etc)
- **dx** (*float*) – decrement in x for each iteration
- **dy** (*float*) – decrement in y for each iteration
- **dz** (*float*) – decrement in z for each iteration
- **N** (*int*) – number of iterations

pyg4ometry.misc.**md5_file**(*fname*)

pyg4ometry.misc.**diffFiles**(*file1*, *file2*)

pyg4ometry.misc.**compareFilesWithAssert**(*file1*, *file2*)

pyg4ometry.misc.**compareNumericallyWithAssert**(*file1*, *file2*)

pyg4ometry.misc.**compareMeshInfo**(*meshInfo1*, *meshInfo2*)

pyg4ometry.misc.**compareGdmlNumericallyWithAssert**(*file1*, *file2*)

pyg4ometry.montecarlo

Submodules

pyg4ometry.montecarlo.beam

Module Contents

Classes

Beam

class pyg4ometry.montecarlo.beam.**Beam**(*pos=[0, 0, 0]*, *dir=[0, 0, 1]*, *energy=1*, *particleType='e-'*)

flukaString()

bdsimString()

pyg4ometry.montecarlo.boundary

Module Contents

Classes

Boundary

class pyg4ometry.montecarlo.boundary.**Boundary**(*name='boundary1'*, *pos=[0, 0, 0]*, *rot=[0, 0, 0]*, *type='plane'*, *size=1*, *length=1*)

flukaString()

bdsimString()

pyg4ometry.montecarlo.config

Module Contents

Classes

Config

class pyg4ometry.montecarlo.config.**Config**(*nHistory=1*, *seed=1*)

flukaString()

bdsimString()

pyg4ometry.montecarlo.scoring

Module Contents

Classes

Scoring

class pyg4ometry.montecarlo.scoring.**Scoring**(*n1=10*, *low1=-10*, *high1=10*, *n2=10*, *low2=-10*, *high2=10*, *n3=10*, *low3=-10*, *high3=10*, *coordType='cartesian'*, *type=''*)

flukaString()

bdsimString()

Package Contents

Classes

Beam
Scoring
Boundary
Config

class pyg4ometry.montecarlo.**Beam**(*pos=[0, 0, 0]*, *dir=[0, 0, 1]*, *energy=1*, *particleType='e-'*)

flukaString()

bdsimString()

class pyg4ometry.montecarlo.**Scoring**(*n1=10*, *low1=-10*, *high1=10*, *n2=10*, *low2=-10*, *high2=10*, *n3=10*, *low3=-10*, *high3=10*, *coordType='cartesian'*, *type=''*)

flukaString()

bdsimString()

class pyg4ometry.montecarlo.**Boundary**(*name='boundary1'*, *pos=[0, 0, 0]*, *rot=[0, 0, 0]*, *type='plane'*, *size=1*, *length=1*)

flukaString()

bdsimString()

class pyg4ometry.montecarlo.**Config**(*nHistory=1*, *seed=1*)

flukaString()

bdsimString()

pyg4ometry.pycgal

Submodules

pyg4ometry.pycgal.HalfPlane

Module Contents

Functions

halfPlaneMesh([planes])

pyg4ometry.pycgal.HalfPlane.**halfPlaneMesh**(*planes=None*)

pyg4ometry.pycgal.core

Module Contents

Classes

CSG
PolygonProcessing
PolyhedronProcessing

Functions

do_intersect(csg1, csg2)
intersecting_meshes(csgList)

class pyg4ometry.pycgal.core.**CSG**

classmethod fromPolygons(*polygons*, ***kwargs*)

toVerticesAndPolygons()

clone()

rotate(*axisIn*, *angleDeg*)

translate(*disp*)

scale(**args*)

getNumberPolys()

getNumberVertices()

vertexCount()

polygonCount()

intersect(*csg2*)

union(*csg2*)

subtract(*csg2*)

inverse()

coplanarIntersection(*csg*)

Compute the coplanar surfaces between self and csg

classmethod cube(*center=[0, 0, 0]*, *radius=[1, 1, 1]*)

Construct an axis-aligned solid cuboid. Optional parameters are *center* and *radius*, which default to *[0, 0, 0]* and *[1, 1, 1]*. The radius can be specified using a single number or a list of three numbers, one for each axis.

Example code:

```
cube = CSG.cube(
  center=[0, 0, 0],
  radius=1
)
```

volume()

area()

isNull()

isClosed()

isTriangleMesh()

isOutwardOriented()

info()

pyg4ometry.pycgal.core.**do_intersect**(*csg1*, *csg2*)

pyg4ometry.pycgal.core.**intersecting_meshes**(*csgList*)

class pyg4ometry.pycgal.core.**PolygonProcessing**

classmethod windingNumber(*pgon*)

return the winding number of pgon :param pgon: list of points [[x1,y1], [x2,y2], …] :type pgon: List[List[x1,y1], …] returns: Integer winding number

classmethod reversePolygon(*pgon*)

return reversed polygon :param pgon: list of points [[x1,y1], [x2,y2], …] :type pgon: List[List[x1,y1], …] returns: List[List[x1,y1], …]

classmethod makePolygonFromList(*pgon*, *type=''*)

Convert list of points [[x1,y1], [x2,y2], …] to cgal Polygon_2

Parameters

- **pgon** (*List[List[x,y], ..]*) – list of points [[x1,y1], [x2,y2], …]
- **type** – Class of polygon (Polygon_2_EPICK, Polygon_2_EPECK, Partition_traits_2_Polygon_2_EPECK)
- **type** – str

returns: Polygon_2

classmethod makeListFromPolygon(*pgon*)

Convert 2D polygon to list of points [[x1,y1], [x2,y2], …]

Parameters
pgon (*Polygon_2_EPECK or Polygon_2_EPICK*) – cgal Polygon_2 input

returns: [[x1,y1], [x2,y2], …]

classmethod decomposePolygon2d(*pgon*)

Decompose general 2D polygon (pgon) to convex 2D polygons

Parameters
pgon (*List(List[2])*) – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of polgons [pgon1, pgon2, …]

classmethod decomposePolygon2dWithHoles(*pgonOuter*, *pgonHoles*)

Decompose general 2D polygon with holes (pgon) to convex 2D polygons

Parameters

- **pgonOuter** – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]
- **pgonHoles** – List of polgons [pgon1, pgon2, …]

returns: List of polgons [pgon1, pgon2, …]

classmethod triangulatePolygon2d(*pgon*)

Triangulate general 2D polygon

Parameters
pgonOuter – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of triangles [[[x1,y1], [x2,y2], [x3,y3]], [[x1,y1], [x2,y2], [x3,y3]], …]

class pyg4ometry.pycgal.core.**PolyhedronProcessing**

classmethod surfaceMesh_to_Polyhedron(*sm*)

classmethod nefPolyhedron_to_convexPolyhedra(*np*)

classmethod polyhedron_to_numpyArrayPlanes(*p*)

pyg4ometry.pycgal.pythonHelpers

Module Contents

Functions

polygon_to_numpy(polygon)	
_draw_polygon_2(p2)	
draw_polygon_2_list(pl)	
draw_polygon_2(p2)	Draw a CGAL polygon for debugging etc

pyg4ometry.pycgal.pythonHelpers.**polygon_to_numpy**(*polygon*)

pyg4ometry.pycgal.pythonHelpers.**_draw_polygon_2**(*p2*)

pyg4ometry.pycgal.pythonHelpers.**draw_polygon_2_list**(*pl*)

pyg4ometry.pycgal.pythonHelpers.**draw_polygon_2**(*p2*)

Draw a CGAL polygon for debugging etc

Parameters

p2 (*Polygon_2, Polygon_with_holes_2, List_Polygon_with_holes_2*) – Polygon_2 for plotting

Package Contents

Classes

CSG
PolygonProcessing
PolyhedronProcessing

Functions

do_intersect(csg1, csg2)
intersecting_meshes(csgList)

class pyg4ometry.pycgal.**CSG**

classmethod fromPolygons(*polygons*, ***kwargs*)

toVerticesAndPolygons()

clone()

rotate(*axisIn*, *angleDeg*)

translate(*disp*)

scale(**args*)

getNumberPolys()

getNumberVertices()

vertexCount()

polygonCount()

intersect(*csg2*)

union(*csg2*)

subtract(*csg2*)

inverse()

coplanarIntersection(*csg*)

Compute the coplanar surfaces between self and csg

classmethod cube(*center=[0, 0, 0]*, *radius=[1, 1, 1]*)

Construct an axis-aligned solid cuboid. Optional parameters are *center* and *radius*, which default to *[0, 0, 0]* and *[1, 1, 1]*. The radius can be specified using a single number or a list of three numbers, one for each axis.

Example code:

```
cube = CSG.cube(
  center=[0, 0, 0],
  radius=1
)
```

volume()

area()

isNull()

isClosed()

isTriangleMesh()

isOutwardOriented()

info()

pyg4ometry.pycgal.**do_intersect**(*csg1*, *csg2*)

pyg4ometry.pycgal.**intersecting_meshes**(*csgList*)

class pyg4ometry.pycgal.**PolygonProcessing**

classmethod windingNumber(*pgon*)

return the winding number of pgon :param pgon: list of points [[x1,y1], [x2,y2], …] :type pgon: List[List[x1,y1], …] returns: Integer winding number

classmethod reversePolygon(*pgon*)

return reversed polygon :param pgon: list of points [[x1,y1], [x2,y2], …] :type pgon: List[List[x1,y1], …] returns: List[List[x1,y1], …]

classmethod makePolygonFromList(*pgon*, *type=''*)

Convert list of points [[x1,y1], [x2,y2], …] to cgal Polygon_2

Parameters

- **pgon** (*List[List[x,y], ..]*) – list of points [[x1,y1], [x2,y2], …]

- **type** – Class of polygon (Polygon_2_EPICK, Polygon_2_EPECK, Partition_traits_2_Polygon_2_EPECK)
- **type** – str

returns: Polygon_2

classmethod makeListFromPolygon(*pgon*)

Convert 2D polygon to list of points [[x1,y1], [x2,y2], …]

Parameters
pgon (*Polygon_2_EPECK or Polygon_2_EPICK*) – cgal Polygon_2 input

returns: [[x1,y1], [x2,y2], …]

classmethod decomposePolygon2d(*pgon*)

Decompose general 2D polygon (pgon) to convex 2D polygons

Parameters
pgon (*List(List[2])*) – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of polgons [pgon1, pgon2, …]

classmethod decomposePolygon2dWithHoles(*pgonOuter*, *pgonHoles*)

Decompose general 2D polygon with holes (pgon) to convex 2D polygons

Parameters

- **pgonOuter** – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]
- **pgonHoles** – List of polgons [pgon1, pgon2, …]

returns: List of polgons [pgon1, pgon2, …]

classmethod triangulatePolygon2d(*pgon*)

Triangulate general 2D polygon

Parameters
pgonOuter – list of pgon points (which are lists) [[x1,y1], [x2,y2], …]

returns: List of triangles [[[x1,y1], [x2,y2], [x3,y3]], [[x1,y1], [x2,y2], [x3,y3]], …]

class pyg4ometry.pycgal.**PolyhedronProcessing**

classmethod surfaceMesh_to_Polyhedron(*sm*)

classmethod nefPolyhedron_to_convexPolyhedra(*np*)

classmethod polyhedron_to_numpyArrayPlanes(*p*)

pyg4ometry.pycgal_old

Submodules

pyg4ometry.pycgal_old.cgal

Module Contents

Functions

pycsgmesh2NefPolyhedron(mesh)	
pycsgmeshWritePolygon(mesh[, fileName])	
numpyPolygonConvex(polygonnp)	

Attributes

g
f
vertexfacet_to_polyhedron
convexpolyhedron_to_planes
surfacemesh_print
polyhedron_write
nefpolyhedron_write
surfacemesh_write
polyhedron_print
nefpolyhedron_print
delete_polyhedron
delete_nefpolyhedron
delete_surfacemesh
polyhedron_to_nefpolyhedron
nefpolyhedron_to_polyhedron
nefpolyhedron_to_surfacemesh
nefpolyhedron_to_convexpolyhedra
vertex_to_polygon
polygon_to_vertex
delete_polygon
polygon_to_convexpolygons
delete_polygonlist

pyg4ometry.pycgal_old.cgal.**g**

pyg4ometry.pycgal_old.cgal.**f**

pyg4ometry.pycgal_old.cgal.**vertexfacet_to_polyhedron**

pyg4ometry.pycgal_old.cgal.**convexpolyhedron_to_planes**

pyg4ometry.pycgal_old.cgal.**surfacemesh_print**

pyg4ometry.pycgal_old.cgal.**polyhedron_write**

pyg4ometry.pycgal_old.cgal.**nefpolyhedron_write**

pyg4ometry.pycgal_old.cgal.**surfacemesh_write**

pyg4ometry.pycgal_old.cgal.**polyhedron_print**

pyg4ometry.pycgal_old.cgal.**nefpolyhedron_print**

pyg4ometry.pycgal_old.cgal.**delete_polyhedron**

pyg4ometry.pycgal_old.cgal.**delete_nefpolyhedron**

pyg4ometry.pycgal_old.cgal.**delete_surfacemesh**

pyg4ometry.pycgal_old.cgal.**polyhedron_to_nefpolyhedron**

pyg4ometry.pycgal_old.cgal.**nefpolyhedron_to_polyhedron**

pyg4ometry.pycgal_old.cgal.**nefpolyhedron_to_surfacemesh**

pyg4ometry.pycgal_old.cgal.**nefpolyhedron_to_convexpolyhedra**

pyg4ometry.pycgal_old.cgal.**vertex_to_polygon**

pyg4ometry.pycgal_old.cgal.**polygon_to_vertex**

pyg4ometry.pycgal_old.cgal.**delete_polygon**

pyg4ometry.pycgal_old.cgal.**polygon_to_convexpolygons**

pyg4ometry.pycgal_old.cgal.**delete_polygonlist**

pyg4ometry.pycgal_old.cgal.**pycsgmesh2NefPolyhedron**(*mesh*)

pyg4ometry.pycgal_old.cgal.**pycsgmeshWritePolygon**(*mesh*, *fileName='mesh.pol'*)

pyg4ometry.pycgal_old.cgal.**numpyPolygonConvex**(*polygonnp*)

Package Contents

Functions

pycsgmesh2NefPolyhedron(mesh)	
pycsgmeshWritePolygon(mesh[, fileName])	
numpyPolygonConvex(polygonnp)	

Attributes

g
f
vertexfacet_to_polyhedron
convexpolyhedron_to_planes
surfacemesh_print
polyhedron_write
nefpolyhedron_write
surfacemesh_write
polyhedron_print
nefpolyhedron_print
delete_polyhedron
delete_nefpolyhedron
delete_surfacemesh
polyhedron_to_nefpolyhedron
nefpolyhedron_to_polyhedron
nefpolyhedron_to_surfacemesh
nefpolyhedron_to_convexpolyhedra
vertex_to_polygon
polygon_to_vertex
delete_polygon
polygon_to_convexpolygons
delete_polygonlist

pyg4ometry.pycgal_old.**g**

pyg4ometry.pycgal_old.**f**

pyg4ometry.pycgal_old.**vertexfacet_to_polyhedron**

pyg4ometry.pycgal_old.**convexpolyhedron_to_planes**

pyg4ometry.pycgal_old.**surfacemesh_print**

pyg4ometry.pycgal_old.**polyhedron_write**

pyg4ometry.pycgal_old.**nefpolyhedron_write**

pyg4ometry.pycgal_old.**surfacemesh_write**

pyg4ometry.pycgal_old.**polyhedron_print**

pyg4ometry.pycgal_old.**nefpolyhedron_print**

pyg4ometry.pycgal_old.**delete_polyhedron**

pyg4ometry.pycgal_old.**delete_nefpolyhedron**

pyg4ometry.pycgal_old.**delete_surfacemesh**

pyg4ometry.pycgal_old.**polyhedron_to_nefpolyhedron**

pyg4ometry.pycgal_old.**nefpolyhedron_to_polyhedron**

pyg4ometry.pycgal_old.**nefpolyhedron_to_surfacemesh**

pyg4ometry.pycgal_old.**nefpolyhedron_to_convexpolyhedra**

pyg4ometry.pycgal_old.**vertex_to_polygon**

pyg4ometry.pycgal_old.**polygon_to_vertex**

pyg4ometry.pycgal_old.**delete_polygon**

pyg4ometry.pycgal_old.**polygon_to_convexpolygons**

pyg4ometry.pycgal_old.**delete_polygonlist**

pyg4ometry.pycgal_old.**pycsgmesh2NefPolyhedron**(*mesh*)

pyg4ometry.pycgal_old.**pycsgmeshWritePolygon**(*mesh*, *fileName='mesh.pol'*)

pyg4ometry.pycgal_old.**numpyPolygonConvex**(*polygonnp*)

pyg4ometry.pycsg

Package Contents

pyg4ometry.pycsg.**__version__** = '0.3.3'

pyg4ometry.pyoce

Submodules

pyg4ometry.pyoce.Reader

Module Contents

Classes

Reader

class pyg4ometry.pyoce.Reader.**Reader**(*fileName*)

readStepFile(*fileName*)

freeShapes()

traverse(*label=None*)

pyg4ometry.pyoce.pythonHelpers

Module Contents

Functions

shapeTopologyCount(shape[, countType, ignoreType])	
shapeTopology(shape)	
findOCCShapeByName(shapeTool, shapeName)	Find a shape by its name
findOCCShapeByTreeNode(label, shapeTreeNode)	
get_TDataStd_Name_From_Label(label)	
get_TDataStd_TreeNode_From_Label(label)	
get_XCAFDoc_Location_From_Label(label)	
get_shapeTypeString(st, label)	
gp_XYZ_numpy(xyz)	
test(fileName)	

pyg4ometry.pyoce.pythonHelpers.**shapeTopologyCount**(*shape*, *countType=_TopAbs.TopAbs_FACE*, *ignoreType=_TopAbs.TopAbs_VERTEX*)

pyg4ometry.pyoce.pythonHelpers.**shapeTopology**(*shape*)

pyg4ometry.pyoce.pythonHelpers.**findOCCShapeByName**(*shapeTool*, *shapeName*)

Find a shape by its name

Parameters

- **shapeTool** (*ShapeTool*) – OpenCascade ShapeTool
- **shapeName** (*str*) – Name of the shape

pyg4ometry.pyoce.pythonHelpers.**findOCCShapeByTreeNode**(*label*, *shapeTreeNode*)

pyg4ometry.pyoce.pythonHelpers.**get_TDataStd_Name_From_Label**(*label*)

pyg4ometry.pyoce.pythonHelpers.**get_TDataStd_TreeNode_From_Label**(*label*)

pyg4ometry.pyoce.pythonHelpers.**get_XCAFDoc_Location_From_Label**(*label*)

pyg4ometry.pyoce.pythonHelpers.**get_shapeTypeString**(*st*, *label*)

pyg4ometry.pyoce.pythonHelpers.**gp_XYZ_numpy**(*xyz*)

pyg4ometry.pyoce.pythonHelpers.**test**(*fileName*)

Package Contents

Classes

Reader

class pyg4ometry.pyoce.**Reader**(*fileName*)

readStepFile(*fileName*)

freeShapes()

traverse(*label=None*)

pyg4ometry.stl

Submodules

pyg4ometry.stl.Reader

Module Contents

Classes

_Facet	
Reader	STL file reader

class pyg4ometry.stl.Reader.**_Facet**(*normal=(0, 0, 0)*)

add_vertex(*xyztup*)

dump()

class pyg4ometry.stl.Reader.**Reader**(*filename*, *solidname='stl_tessellated'*, *scale=1*, *centre=False*, *registry=None*, *forcebinary=False*)

STL file reader

Parameters

- **filename** (*str*) – Input STL filename
- **solidname** (*str*) – Name of the solid to be created
- **scale** (*float*) – Scaling of STL (e.g. for units)
- **centre** (*boolean*) – Flag to centre STL solid
- **registry** (Registry) – Registry to add solid to
- **forcebinary** (*boolean*) – Forces to load this STL file in binary format, otherwise the file format is determined from whether it starts with the string 'solid'

_load_ascii(*data*)

Load ASCII STL file from bytes instance

_load_binary(*data*)

Load binary STL file from bytes instance

extent()

Compute the axis aligned extent of the STL solid.

Returns
list of minima and maxima in 3 axes

Return type
[[xmin,ymin,zmin],[xmax, ymax, zmax]]

extentCentre()

Translate STL mesh to centre of the extent

translate(*translation=[0, 0, 0]*)

Translate STL mesh by translation

Parameters
translation (*list(3) or array(3)*) – Vector to translate mesh

getSolid()

Get geant4.solid

Returns
G4Tesselated for STL

Return type
TessellatedSolid

getRegistry()

Return registry

visualise()

View solid directly by using a dummy world

writeDefaultGDML(*filename='default'*, *gmad_tester=False*)

Write the tessellated solid loaded from STL to GDML. The placement has no rotation or translation. The world material is G4_Galactic, the solid material is G4_Fe.

Parameters

- **filename** (*str*) – Output file name
- **gmad_tester** (*boolean*) – Flag for writing BDSIM gmad tester

Package Contents

Classes

_Facet
Reader

class pyg4ometry.stl.**_Facet**(*normal=(0, 0, 0)*)

add_vertex(*xyztup*)

dump()

class pyg4ometry.stl.**Reader**(*filename*, *solidname='stl_tessellated'*, *scale=1*, *centre=False*, *registry=None*, *forcebinary=False*)

STL file reader

Parameters

- **filename** (*str*) – Input STL filename
- **solidname** (*str*) – Name of the solid to be created
- **scale** (*float*) – Scaling of STL (e.g. for units)
- **centre** (*boolean*) – Flag to centre STL solid
- **registry** (*Registry*) – Registry to add solid to
- **forcebinary** (*boolean*) – Forces to load this STL file in binary format, otherwise the file format is determined from whether it starts with the string 'solid'

_load_ascii(*data*)

Load ASCII STL file from bytes instance

_load_binary(*data*)

Load binary STL file from bytes instance

extent()

Compute the axis aligned extent of the STL solid.

Returns
list of minima and maxima in 3 axes

Return type
[[xmin,ymin,zmin],[xmax, ymax, zmax]]

extentCentre()

Translate STL mesh to centre of the extent

translate(*translation=[0, 0, 0]*)

Translate STL mesh by translation

Parameters
translation (*list(3) or array(3)*) – Vector to translate mesh

getSolid()

Get geant4.solid

Returns
G4Tesselated for STL

Return type
TessellatedSolid

getRegistry()

Return registry

visualise()

View solid directly by using a dummy world

writeDefaultGDML(*filename='default'*, *gmad_tester=False*)

Write the tessellated solid loaded from STL to GDML. The placement has no rotation or translation. The world material is G4_Galactic, the solid material is G4_Fe.

Parameters

- **filename** (*str*) – Output file name
- **gmad_tester** (*boolean*) – Flag for writing BDSIM gmad tester

pyg4ometry.visualisation

Submodules

pyg4ometry.visualisation.Convert

Module Contents

Functions

mkVtkIdList(it)	
pycsgMeshToVtkPolyData(mesh)	
vtkPolyDataToNumpy(data)	
pycsgMeshToObj(mesh, fileName)	
pyg42VtkTransformation(mtra, tra)	
vtkTransformation2PyG4(vt)	
pycsgMeshToStl(mesh, fileName)	

pyg4ometry.visualisation.Convert.**mkVtkIdList**(*it*)

pyg4ometry.visualisation.Convert.**pycsgMeshToVtkPolyData**(*mesh*)

pyg4ometry.visualisation.Convert.**vtkPolyDataToNumpy**(*data*)

pyg4ometry.visualisation.Convert.**pycsgMeshToObj**(*mesh*, *fileName*)

pyg4ometry.visualisation.Convert.**pyg42VtkTransformation**(*mtra*, *tra*)

pyg4ometry.visualisation.Convert.**vtkTransformation2PyG4**(*vt*)

pyg4ometry.visualisation.Convert.**pycsgMeshToStl**(*mesh*, *fileName*)

pyg4ometry.visualisation.Mesh

Module Contents

Classes

OverlapType	
Mesh	

Functions

_getBoundingBox(aMesh[, rotationMatrix, translation, ...])	Axes aligned bounding box. Can also provide a rotation and
_getBoundingBoxMesh(boundingBox)	

class pyg4ometry.visualisation.Mesh.**OverlapType**

protrusion = 1

overlap = 2

coplanar = 3

class pyg4ometry.visualisation.Mesh.**Mesh**(*solid*)

remesh()

addOverlapMesh(*mesh*)

getLocalMesh()

getBoundingBox(*rotationMatrix=None*, *translation=None*)

Axes aligned bounding box. Can also provide a rotation and a translation (applied in that order) to the vertices.

getBoundingBoxMesh()

pyg4ometry.visualisation.Mesh.**_getBoundingBox**(*aMesh*, *rotationMatrix=None*, *translation=None*, *nameForError=''*)

Axes aligned bounding box. Can also provide a rotation and a translation (applied in that order) to the vertices.

pyg4ometry.visualisation.Mesh.**_getBoundingBoxMesh**(*boundingBox*)

pyg4ometry.visualisation.Plot

Module Contents

Functions

AddCutterDataToPlot(filename[, projection, ax, ...])	Add cutter data to a plot or draw in a new figure if no axis object given.

pyg4ometry.visualisation.Plot.**AddCutterDataToPlot**(*filename*, *projection='zx'*, *ax=None*, *unitsFactor=1.0*, *colour='k'*, *linewidth=0.5*, *alpha=1.0*)

Add cutter data to a plot or draw in a new figure if no axis object given.

Parameters

- **filename** (*str*) – file name of exported cutter data from visualiser.

- **projection** (*str*) – which two axes to plot in a 2D plot from the 3D data. e.g. zx, zy.
- **ax** (*matplotlib.axes._axes.Axes*) – matplotlib axes instance.
- **unitsFactor** (*float*) – numerical scaling factor for coordinates (e.g. 0.001 for plotting in meters).
- **colour** (*str*) – matplotlib.pyplot.plot colour argument
- **linewidth** (*float*) – matplotlib.pyplot.plot linewidth argument
- **alpha** (*float*) – matplotlib.pyplot.plot alpha argument for transparency

If used without an axes, a new figure and axes will be created.

The project specifies the order of plotting, so 'zx' would give z in the plot's x axis and 'x' in plot's y axis. Any combination of 'x', 'y' and 'z' are accepted and must be different. e.g. any of 'xy', 'xz', 'yx', 'yz', 'zx', 'zy'.

pyg4ometry.visualisation.RenderWriter

Module Contents

Classes

RenderWriter	

class pyg4ometry.visualisation.RenderWriter.**RenderWriter**

addLogicalVolumeRecursive(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*)

addMesh(*logical*)

addInstance(*logical*, *transformation*, *translation*)

write(*outputDirectory*)

pyg4ometry.visualisation.ViewerBase

Module Contents

Classes

ViewerBase	Base class for all viewers and exporters. Handles unique meshes and their instances

Functions

_daughterSubtractedMesh(lv)	

pyg4ometry.visualisation.ViewerBase.**_daughterSubtractedMesh**(*lv*)

class pyg4ometry.visualisation.ViewerBase.**ViewerBase**

Base class for all viewers and exporters. Handles unique meshes and their instances

clear()

setSubtractDaughters(*subtractDaughters=True*)

addLogicalVolume(*lv*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *visOptions=_VisOptions(representation='wireframe')*, *depth=0*, *name=None*)

Add a logical volume to viewer (recursively)

Parameters

- **mtra** (*matrix(3,3)*) – Transformation matrix for logical volume
- **tra** (*array(3)*) – Displacement for logical volume
- **visOptions** (VisualisationOptions) – VisualisationOptions for the lv mesh

addFlukaRegions(*fluka_registry*, *max_region=1000000*, *debugIO=False*)

addMesh(*name*, *mesh*)

Add a single mesh

Parameters

- **name** (*str*) – Name of mesh (e.g logical volume name)
- **mesh** (CSG) – Mesh to be added

addInstance(*name*, *transformation*, *translation*, *instanceName=''*)

Add a new instance for mesh with name

Parameters

- **name** (*str*) – name of mesh to add instance
- **transformation** (*array(3,3)*) – Transformation matrix for instance
- **translation** (*array(3)*) – Translation for instance
- **instanceName** (*str*) – Name of the instance e.g PV

addVisOptions(*name*, *visOption*)

Add vis options to mesh with name

Parameters

- **name** (*str*) – name of mesh
- **visOptions** (VisualisationOptions) –

addPbrOptions(*name*, *pbrOption*)

Add pbr options to mesh with name

Parameters

- **name** (*str*) – name of mesh
- **pbrOptions** (*PbrOptions*) –

addMaterialVisOption(*materialName*, *visOption*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** (*str*) – material name to match
- **visOption** (VisualisationOptions) – instance

addMaterialPbrOption(*materialName*, *visOption*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** (*str*) – material name to match
- **visOption** (VisualisationOptions) – instance

setDefaultVisOptions(*visOption*)

getDefaultVisOptions()

getMaterialVisOptions(*material*)

setOverlapVisOption(*visOption*)

setCoplanarVisOption(*visOption*)

setProtusionVisOption(*visOption*)

getOverlapVisOptions(*overlaptype*)

removeInvisible()

Remove wireframe or transparent instances from self

scaleScene(*scaleFactor*)

exportGLTFScene(*gltfFileName='test.gltf'*, *singleInstance=False*)

Export entire scene as gltf file, filename extension dictates binary (glb) or readable json (gltf) singleInstance is a Boolean flag to supress all but one instance

exportGLTFAssets(*gltfFileName='test.gltf'*)

Export all the assets (meshes) without all the instances. The position of the asset is the position of the first instance

exportThreeJSScene(*fileNameBase='test'*, *lightBoxHDR='concrete_tunnel_02_4k.hdr'*)

html based on https://threejs.org/examples/#webgl_loader_gltf HRDI https://polyhaven.com/a/concrete_tunnel_02

npm install node wget https://dl.polyhaven.org/file/ph-assets/HDRIs/exr/4k/concrete_tunnel_02_4k.exr conver EXR to HDR e.g. https://convertio.co/exr-hdr/ python -m http.server 8000 open test.html

dumpMeshQuality()

__repr__()
Return repr(self).

pyg4ometry.visualisation.VisualisationOptions

Module Contents

Classes

VisualisationOptions	Basic holder for visualisation parameters, i.e. colour and opacity.

Functions

randomColour()	Return random RGB tuple
hexRGBToRGBTriplet(value)	
loadPredefined()	Construct a ColorMap initialised with default colours for various materials.
getPredefinedMaterialVisOptions()	
makeVisualisationOptionsDictFromPredefined(C	

Attributes

_predefinedMaterialVisOptions

pyg4ometry.visualisation.VisualisationOptions._predefinedMaterialVisOptions

pyg4ometry.visualisation.VisualisationOptions.randomColour()
Return random RGB tuple

pyg4ometry.visualisation.VisualisationOptions.hexRGBToRGBTriplet(*value*)

class pyg4ometry.visualisation.VisualisationOptions.VisualisationOptions(*representation='surface'*, *colour=[0.5, 0.5, 0.5]*, *alpha=0.5*, *visible=True*, *lineWidth=1*, *randomColour=False*, *depth=0*)
Basic holder for visualisation parameters, i.e. colour and opacity. Construct an instance then modify members.

__repr__()

Return repr(self).

getColour()

Return the colour and generate a random colour if flagged.

_generateRandomColour()

pyg4ometry.visualisation.VisualisationOptions.**loadPredefined**()

Construct a ColorMap initialised with default colours for various materials.

Lods from package resource files.

pyg4ometry.visualisation.VisualisationOptions.**getPredefinedMaterialVisOptions**()

pyg4ometry.visualisation.VisualisationOptions.**makeVisualisationOptionsDictFromPredefined**(*ColourMap*)

pyg4ometry.visualisation.VtkExporter

Module Contents

Classes

VtkExporter

Attributes

_WITH_PARAVIEW
_WITH_PARAVIEW

pyg4ometry.visualisation.VtkExporter.**_WITH_PARAVIEW = True**

pyg4ometry.visualisation.VtkExporter.**_WITH_PARAVIEW = False**

class pyg4ometry.visualisation.VtkExporter.**VtkExporter**(*path='.'*)

export_to_Paraview(*reg*, *fileName='Paraview_model.pvsm'*, *model=True*, *df_model=None*, *df_color=None*)

Method that exports the visible logical volumes of the registry reg into Paraview VTK files (.vtm) and creates a Paraview State file (.pvsm) for the Paraview Model.

df_model and df_color are optional: if they are not given, the Paraview model will take the materials colors

Parameters

- **reg** – Registry
- **fileName** – Name of the Paraview State file (.pvsm)
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param df_model: (optional) pandas.DataFrame linking the NAME of the element and their TYPE :param df_color: (optional) pandas.DataFrame linking the TYPE with its specific R G B color

export_to_VTK(*reg*, *model=True*, *df_model=None*, *df_color=None*)

Method that exports the visible logical volumes of the registry reg into Paraview VTK files (.vtm).

Parameters

- **reg** – Registry
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param df_model: (optional) pandas.DataFrame linking the NAME of the element and their TYPE :param df_color: (optional) pandas.DataFrame linking the TYPE with its specific R G B color

fill_color_dico(*df_gdml*, *df_model*, *df_color*)

Method that fills a dictionary linking the logical volumes names and their respective color based on the pandas.DataFrame df_color linking the TYPE of the elements and their respective RGB color values.

Parameters

- **df_gdml** – pandas.DataFrame that represents the GDML Structure
- **df_model** – pandas.DataFrame linking the NAME of the element and their TYPE
- **df_color** – pandas.DataFrame linking the TYPE with its specific R G B color

Returns: A dictionary with the keys R, G and B whose item is a dictionary with as keys the logical volumes names and as item the respective RGB value

add_logical_world_volume(*lv*, *model*, *color_dico={'R': {}, 'G': {}, 'B': {}}*, *rotation=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *translation=_np.array([0, 0, 0])*)

Method that receives the logical world volume and calls the necessary methods to write the .vtm file(s).

Parameters

- **lv** – Logical world volume
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param color_dico: A dictionary with the keys R, G and B whose item is a dictionary with as keys the logical volumes names and as item the respective RGB value :param rotation: (optional) numpy.matrix :param translation: (optional) numpy.array

add_logical_volume_recursive(*lv*, *rotation*, *translation*, *color_dico*, *first_level=True*)

Method that receives a logical volume and add calls addMesh() on its mesh. The method is recursive and will be called on all the daughter logical volumes of the logical volume.

Parameters

- **lv** – Logical volume
- **rotation** – numpy.matrix
- **ranslation** – numpy.array
- **color_dico** – A dictionary with the keys R, G and B whose item is a dictionary

with as keys the logical volumes names and as item the respective RGB value :param first_level: Boolean indicating if we are at the first level of recursivity

addMesh(*solid_name*, *mesh*, *rotation*, *translation*, *visOptions=None*)

Method that converts the mesh into VtkPolyData (.vtp file) and gives it the correct rotation, translation and color.

Parameters

- **solid_name** – Name of the logical volume solid
- **mesh** – Mesh of the logical volume
- **rotation** – numpy.matrix
- **translation** – numpy.array
- **visOptions** – VisualisationOptions instance

getMaterialVisOptions(*name*)

Method that "cleans" the logical volume material string.

Parameters
name (*str*) – raw name of the logical volume material

Returns: clean name of the logical volume material

getElementName(*logicalVolumeName*)

Method that "cleans" the logical volume name string.

Parameters
logicalVolumeName (*str*) – raw name of the logical volume

Returns: clean name of the logical volume

countVisibleDaughters(*lv*, *element_name*, *n=0*)

Method that counts the number of "visible" daughter logical volumes of the mother logical volume lv.

Parameters

- **lv** (pyg4ometry.geant4.LogicalVolume) – logical volume
- **element_name** (*str*) – name of the element
- **n** (*int*) – number of "visible" daughter volumes

Returns: n

pyg4ometry.visualisation.VtkViewer

Module Contents

Classes

VtkViewer	Visualiser.
VtkViewerColoured	Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.
VtkViewerColouredMaterial	Extension of VtkViewerColoured that uses a default material dictionary for
MouseInteractorNamePhysicalVolume	

Functions

axesFromExtents(extent)	
viewLogicalVolumeDifference(referenceLV, otherLV[, ...])	**param referenceLV** LogicalVolume instance to view viewed in red.

Attributes

PubViewer

class pyg4ometry.visualisation.VtkViewer.**VtkViewer**(*size=(1024, 1024)*, *interpolation='none'*, ***kwargs*)

Visualiser.

Parameters

- **size** – (int,int) - (nPixelsHorizontal, nPixelsVeritcal), default (1024,1024)
- **interpolation** – (str) - one of "none", "flat", "gouraud", "phong"

Examples

```
>>> v = VtkViewer()
>>> v.addLogicalVolume(someLV)
>>> v.view()
```

addAxes(*length=20.0*, *origin=(0, 0, 0)*)

Add x,y,z axis to the scene.

Parameters

- **length** – float - length of each axis in mm
- **origin** – (float,float,float) - (x,y,z) of origin in mm

addAxesWidget()

setOpacity(*v*, *iActor=-1*)

setWireframe(*iActor=-1*)

setSurface(*iActor=-1*)

setOpacityOverlap(*v*, *iActor=-1*)

setWireframeOverlap(*iActor=-1*)

setSurfaceOverlap(*iActor=-1*)

setRandomColours(*seed=0*)

setCutterOrigin(*dimension*, *origin*)

Parameters

- **dimension** – str - 'x', 'y', or 'z'
- **origin** – list([x,y,z])

setCutterNormal(*dimension*, *normal*)

Parameters

- **dimension** – str - 'x', 'y', or 'z'
- **normal** – list([x,y,z]) - should be unit vector

addMaterialVisOption(*materialName*, *visOptionInstance*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** – str - material name to match
- **visOptionInstance** – `VisualisationOptions` instance

setMaterialVisOptions(*materialDict*)

Replace the (by default None) dictionary for materials to colours :param materialDict: {"materialName": VisualisationOptions}

See also `VisualisationOptions`.

setCameraFocusPosition(*focalPoint=[0, 0, 0]*, *position=[100, 100, 100]*)

start()

exportOBJScene(*fileName='scene'*)

exportVRMLScene(*fileName='scene'*)

exportGLTFScene(*fileName='scene.gltf'*)

exportScreenShot(*fileName='screenshot.png'*, *rgba=True*)

Write the render window view to an image file.

Image types supported are: BMP, JPEG, PNM, PNG, PostScript, TIFF. The default parameters are used for all writers, change as needed.

Parameters

- **fileName** – The file name, if no extension then PNG is assumed.
- **renWin** – The render window.
- **rgba** – Used to set the buffer type.

Returns

addLogicalVolume(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *recursive=True*, *addWorld=True*)

addLogicalVolumeBounding(*logical*)

addSolid(*solid*, *rotation=[0, 0, 0]*, *position=[0, 0, 0]*, *representation='surface'*, *colour=[0.5, 0.5, 0.5]*, *opacity=0.2*)

Add a solid to the view with an optional rotation and translation.

Parameters

- **solid** (*any solid in pyg4ometry.geant4.solid*) – solid to add to the view
- **rotation** (*list(float, float, float) - 3 values*) – list of TB rotation angles in radians
- **position** (*list(float, float, float) - 3 values*) – translation in global from from centre in mm
- **representation** (*str*) – the way to visualise it, e.g. 'surface' or 'wireframe'
- **colour** (*list(float, float, float) - 3 values ranging from 0 - 1*) – normalised rgb colour to use for the solid mesh
- **opacity** (*float, from 0 to 1*) – the opacity of the solid if surface style

addBooleanSolidRecursive(*solid*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *first=True*)

Parameters

solid (pyg4ometry.geant4.solid.SolidBase) – pyg4ometry.geant4.solid instance.

Other parameters are for internal recursion and don't need to be provided.

Render only a Boolean solid. If one of the constituent solids is also a Boolean, visualise those too. The resultant Boolean is shown in solid form and each constituent in a wireframe. In the case of a null mesh, only the constituents can be shown.

addMeshSimple(*csgMesh*, *visOptions=_VisOptions()*, *clip=False*, *name='mesh'*)

addLogicalVolumeRecursive(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*)

addMesh(*pv_name*, *solid_name*, *mesh*, *mtra*, *tra*, *localmeshes*, *filters*, *mappers*, *mapperMap*, *actors*, *actorMap*, *visOptions=None*, *overlap=False*, *cutters=True*, *clippers=False*)

view(*interactive=True*, *resetCamera=True*)

_getCutterData(*axis='x'*, *scaling=1.0*)

exportCutterSection(*filename*, *normal='x'*, *scaling=1.0*)

Export the section lines in plane perpendicular to normal. Exported as json text.

Parameters

- **filename** – (str) - name of file to export to

- **normal** – (str) - one of “x”, “y” or “z”
- **scaling** – (float) - multiplier for all cutter line coordinates on export

Examples

```
>>> v.exportCutterSection("xz-section.dat", normal="y", scaling=1000)
```

viewSection(*dir='x'*)

addCutterPlane(*position*, *normal*, *colour=None*)

Add a cutting plane at position=[x,y,z] with normal [nx,ny,nz].

Parameters

- **position** – [float, float, float] - (x,y,z) position in scene (mm)
- **normal** – [float, float, float] - (nx,ny,z) normal unit vector
- **colour** – None or [float, float, float] - [r,g,b] in range [0:1]

Cutters are stored in self.usercutters.

addActor(*actor*)

getOverlapVisOptions(*overlaptype*)

getMaterialVisOptions(*pv*)

_getDefaultVis(*pv*)

printViewParameters()

class pyg4ometry.visualisation.VtkViewer.**VtkViewerColoured**(**args*, *defaultColour=None*, *materialVisOptions=None*, ***kwargs*)

Bases: *VtkViewer*

Visualiser that extends VtkViewer. Uses “flat” interpolation and introduces control over colours.

Keyword Arguments

- **materialVisOptions**: {“materialName”: `VisualisationOptions` or list or tuple, ...}
- **interpolation** (str): see *VtkViewer*
- **defaultColour** (str): “random” or [r,g,b]

Examples

```
>>> vMaterialMap = VtkViewerColoured(materialVisOptions={"G4_WATER":[0,0,1]})
>>> vRandom = VtkViewerColoured(defaultColour="random")
>>> vColoured = VtkViewerColoured(defaultColour=[0.1,0.1,0.1])
>>> vColourAlpha = VtkViewerColoured(defaultColour=[0.1,0.1,0.1,0.5])
```

of use visualisation options instances

```
>>> vo = pyg4ometry.visualisation.VisualisationOptions()
>>> vo.colour = [0.1,1.0,0.5]
>>> vo.alpha = 0.3
>>> options = {'G4_WATER':vo}
>>> vis = VtkViewerColoured(materialVisOptions=options)
```

If the value in the materialVisOptions is a list or a tuple, it will be upgraded to a `VisualisationOptions` instance.

_getDefaultVis(*pv*)

pyg4ometry.visualisation.VtkViewer.**PubViewer**

class pyg4ometry.visualisation.VtkViewer.**VtkViewerColouredMaterial**(**args*, ***kwargs*)

Bases: *PubViewer*

Extension of VtkViewerColoured that uses a default material dictionary for several common materials. Material colours are in defined Colour.py for many Geant4, FLUKA and BDSIM materials.

class pyg4ometry.visualisation.VtkViewer.**MouseInteractorNamePhysicalVolume**(*renderer*, *vtkviewer*)

Bases: `vtk.vtkInteractorStyleTrackballCamera`

rightButtonPressEvent(*obj*, *event*)

pyg4ometry.visualisation.VtkViewer.**axesFromExtents**(*extent*)

pyg4ometry.visualisation.VtkViewer.**viewLogicalVolumeDifference**(*referenceLV*, *otherLV*, *otherTranslation=[0, 0, 0]*, *otherRotation=[0, 0, 0]*, *viewDifference=True*)

Parameters

- **referenceLV** (*pyg4ometry.geant4.LogicalVolume.*) – LogicalVolume instance to view viewed in red.
- **referenceLV** – LogicalVolume instance to view viewed in blue.
- **otherTranslation** (*[float, float, float]*) – Translation (in native units, mm) of otherLV w.r.t. referenceLV
- **otherRotation** (*[float, float, float]*) – Rotation (in native units, rad) of other LV w.r.t. referenceLV.

View the shapes of 2 logical volumes without their contents. The reference one will be red and the 'other' one will be blue.

The other one may optionally be translated and rotated (Tait-Bryant x,y,z) relative to the reference.

pyg4ometry.visualisation.VtkViewerNew

Module Contents

Classes

VtkViewerNew	
VtkViewerColouredNew	Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.
VtkViewerColouredMaterialNew	Extension of VtkViewerColoured that uses a default material dictionary for
MouseInteractorNamePhysicalVolume	

class pyg4ometry.visualisation.VtkViewerNew.**VtkViewerNew**

Bases: *pyg4ometry.visualisation.ViewerBase*

initVtk()

clear()

addAxes(*length=20.0*, *origin=(0, 0, 0)*)

Add x,y,z axis to the scene.

Parameters

- **length** – float - length of each axis in mm
- **origin** – (float,float,float) - (x,y,z) of origin in mm

setAxes(*iAxes*, *length*, *origin*)

addAxesWidget()

addCutter(*name*, *origin*, *normal*)

setCutter(*name*, *origin*, *normal*)

addCutterWidget()

exportCutter(*name*, *fileName*)

getCutterPolydata(*name*)

addClipper(*origin*, *normal*, *bClipperCutter=False*, *bClipperCloseCuts=True*)

setClipper(*origin*, *normal*)

addClipperWidget()

updateClipperPlaneCallback(*obj*, *event*)

_polydata2Actor(*polydata*)

buildPipelines()

buildPipelinesSeparate()

buildPipelinesAppend()

buildPipelinesTransformed()

render()

view(*interactive=True*, *resetCamera=False*)

__repr__()

addTracks(*pd*)

addScoringMesh(*gd*)

class pyg4ometry.visualisation.VtkViewerNew.**VtkViewerColouredNew**(**args*, *defaultColour=None*, *materialVisOptions=None*, ***kwargs*)

Bases: *VtkViewerNew*

Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.

Keyword Arguments

- **materialVisOptions**: {"materialName": `VisualisationOptions` or list or tuple, ... }
- **interpolation** (str): see `VtkViewer`
- **defaultColour** (str): "random" or [r,g,b]

Examples

```
>>> vMaterialMap = VtkViewerColoured(materialVisOptions={"G4_WATER":[0,0,1]})
>>> vRandom = VtkViewerColoured(defaultColour="random")
>>> vColoured = VtkViewerColoured(defaultColour=[0.1,0.1,0.1])
>>> vColourAlpha = VtkViewerColoured(defaultColour=[0.1,0.1,0.1,0.5])
```

of use visualisation options instances

```
>>> vo = pyg4ometry.visualisation.VisualisationOptions()
>>> vo.colour = [0.1,1.0,0.5]
>>> vo.alpha = 0.3
>>> options = {'G4_WATER':vo}
>>> vis = VtkViewerColoured(materialVisOptions=options)
```

If the value in the materialVisOptions is a list or a tuple, it will be upgraded to a `VisualisationOptions` instance.

_getDefaultVis(*pv*)

class pyg4ometry.visualisation.VtkViewerNew.**VtkViewerColouredMaterialNew**(**args*, ***kwargs*)

Bases: *VtkViewerColouredNew*

Extension of VtkViewerColoured that uses a default material dictionary for several common materials. Material colours are in defined Colour.py for many Geant4, FLUKA and BDSIM materials.

class pyg4ometry.visualisation.VtkViewerNew.**MouseInteractorNamePhysicalVolume**(*renderer*, *vtkviewer*)

Bases: `vtk.vtkInteractorStyleTrackballCamera`

removeHighLight()

removeHighLightText()

rightButtonPressEvent(*obj*, *event*)

pyg4ometry.visualisation.Writer

Module Contents

Functions

writeVtkPolyDataAsSTLFile(fileName, meshes)	meshes : list of triFilters

pyg4ometry.visualisation.Writer.**writeVtkPolyDataAsSTLFile**(*fileName*, *meshes*)

meshes : list of triFilters

Package Contents

Classes

Mesh	
OverlapType	
ViewerBase	Base class for all viewers and exporters. Handles unique meshes and their instances
VisualisationOptions	
VtkViewer	
VtkViewerColoured	Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.
VtkViewerColouredMaterial	Extension of VtkViewerColoured that uses a default material dictionary for
MouseInteractorNamePhysicalVolume	
_VisOptions	Basic holder for visualisation parameters, i.e. colour and opacity.
VtkViewerNew	
VtkViewerColouredNew	Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.
VtkViewerColouredMaterialNew	Extension of VtkViewerColoured that uses a default material dictionary for
MouseInteractorNamePhysicalVolume	
RenderWriter	
VtkExporter	

Functions

_getBoundingBox(aMesh[, rotationMatrix, translation, ...])	Axes aligned bounding box. Can also provide a rotation and
_getBoundingBoxMesh(boundingBox)	
randomColour()	Return random RGB tuple
hexRGBToRGBTriplet(value)	
loadPredefined()	Construct a ColorMap initialised with default colours for various materials.
getPredefinedMaterialVisOptions()	
makeVisualisationOptionsDictFromPredefined(C	
_getPredefinedMaterialVisOptions()	
axesFromExtents(extent)	
viewLogicalVolumeDifference(referenceLV, otherLV[, ...])	**param referenceLV** LogicalVolume instance to view viewed in red.
_getPredefinedMaterialVisOptions()	
mkVtkIdList(it)	
pycsgMeshToVtkPolyData(mesh)	
vtkPolyDataToNumpy(data)	
pycsgMeshToObj(mesh, fileName)	
pyg42VtkTransformation(mtra, tra)	
vtkTransformation2PyG4(vt)	
pycsgMeshToStl(mesh, fileName)	

Attributes

_predefinedMaterialVisOptions
PubViewer
_WITH_PARAVIEW

class pyg4ometry.visualisation.**Mesh**(*solid*)

remesh()

addOverlapMesh(*mesh*)

getLocalMesh()

getBoundingBox(*rotationMatrix=None*, *translation=None*)

Axes aligned bounding box. Can also provide a rotation and a translation (applied in that order) to the vertices.

getBoundingBoxMesh()

class pyg4ometry.visualisation.**OverlapType**

protrusion = 1

overlap = 2

coplanar = 3

pyg4ometry.visualisation.**_getBoundingBox**(*aMesh*, *rotationMatrix=None*, *translation=None*, *nameForError=''*)

Axes aligned bounding box. Can also provide a rotation and a translation (applied in that order) to the vertices.

pyg4ometry.visualisation.**_getBoundingBoxMesh**(*boundingBox*)

class pyg4ometry.visualisation.**ViewerBase**

Base class for all viewers and exporters. Handles unique meshes and their instances

clear()

setSubtractDaughters(*subtractDaughters=True*)

addLogicalVolume(*lv*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *visOptions=_VisOptions(representation='wireframe')*, *depth=0*, *name=None*)

Add a logical volume to viewer (recursively)

Parameters

- **mtra** (*matrix(3,3)*) – Transformation matrix for logical volume
- **tra** (*array(3)*) – Displacement for logical volume
- **visOptions** (VisualisationOptions) – VisualisationOptions for the lv mesh

addFlukaRegions(*fluka_registry*, *max_region=1000000*, *debugIO=False*)

addMesh(*name*, *mesh*)

Add a single mesh

Parameters

- **name** (*str*) – Name of mesh (e.g logical volume name)
- **mesh** (CSG) – Mesh to be added

addInstance(*name*, *transformation*, *translation*, *instanceName=''*)

Add a new instance for mesh with name

Parameters

- **name** (*str*) – name of mesh to add instance
- **transformation** (*array(3,3)*) – Transformation matrix for instance
- **translation** (*array(3)*) – Translation for instance
- **instanceName** (*str*) – Name of the instance e.g PV

addVisOptions(*name*, *visOption*)

Add vis options to mesh with name

Parameters

- **name** (*str*) – name of mesh
- **visOptions** (VisualisationOptions) –

addPbrOptions(*name*, *pbrOption*)

Add pbr options to mesh with name

Parameters

- **name** (*str*) – name of mesh
- **pbrOptions** (*PbrOptions*) –

addMaterialVisOption(*materialName*, *visOption*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** (*str*) – material name to match
- **visOption** (*VisualisationOptions*) – instance

addMaterialPbrOption(*materialName*, *visOption*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** (*str*) – material name to match
- **visOption** (*VisualisationOptions*) – instance

setDefaultVisOptions(*visOption*)

getDefaultVisOptions()

getMaterialVisOptions(*material*)

setOverlapVisOption(*visOption*)

setCoplanarVisOption(*visOption*)

setProtusionVisOption(*visOption*)

getOverlapVisOptions(*overlaptype*)

removeInvisible()

Remove wireframe or transparent instances from self

scaleScene(*scaleFactor*)

exportGLTFScene(*gltfFileName='test.gltf'*, *singleInstance=False*)

Export entire scene as gltf file, filename extension dictates binary (glb) or readable json (gltf) singleInstance is a Boolean flag to supress all but one instance

exportGLTFAssets(*gltfFileName='test.gltf'*)

Export all the assets (meshes) without all the instances. The position of the asset is the position of the first instance

exportThreeJSScene(*fileNameBase='test'*, *lightBoxHDR='concrete_tunnel_02_4k.hdr'*)

html based on https://threejs.org/examples/#webgl_loader_gltf HRDI https://polyhaven.com/a/concrete_tunnel_02

npm install node wget https://dl.polyhaven.org/file/ph-assets/HDRIs/exr/4k/concrete_tunnel_02_4k.exr conver EXR to HDR e.g. https://convertio.co/exr-hdr/ python -m http.server 8000 open test.html

dumpMeshQuality()

__repr__()

Return repr(self).

pyg4ometry.visualisation.**_predefinedMaterialVisOptions**

pyg4ometry.visualisation.**randomColour**()

Return random RGB tuple

pyg4ometry.visualisation.**hexRGBToRGBTriplet**(*value*)

class pyg4ometry.visualisation.**VisualisationOptions**(*representation='surface'*, *colour=[0.5, 0.5, 0.5]*, *alpha=0.5*, *visible=True*, *lineWidth=1*, *randomColour=False*, *depth=0*)

Basic holder for visualisation parameters, i.e. colour and opacity. Construct an instance then modify members.

__repr__()

Return repr(self).

getColour()

Return the colour and generate a random colour if flagged.

_generateRandomColour()

pyg4ometry.visualisation.**loadPredefined**()

Construct a ColorMap initialised with default colours for various materials.

Lods from package resource files.

pyg4ometry.visualisation.**getPredefinedMaterialVisOptions**()

pyg4ometry.visualisation.**makeVisualisationOptionsDictFromPredefined**(*ColourMap*)

pyg4ometry.visualisation.**_getPredefinedMaterialVisOptions**()

class pyg4ometry.visualisation.**VtkViewer**(*size=(1024, 1024)*, *interpolation='none'*, ***kwargs*)

Visualiser.

Parameters

- **size** – (int,int) - (nPixelsHorizontal, nPixelsVeritcal), default (1024,1024)
- **interpolation** – (str) - one of “none”, “flat”, “gouraud”, “phong”

Examples

```
>>> v = VtkViewer()
>>> v.addLogicalVolume(someLV)
>>> v.view()
```

addAxes(*length=20.0*, *origin=(0, 0, 0)*)

Add x,y,z axis to the scene.

Parameters

- **length** – float - length of each axis in mm
- **origin** – (float,float,float) - (x,y,z) of origin in mm

addAxesWidget()

setOpacity(*v*, *iActor=-1*)

setWireframe(*iActor=-1*)

setSurface(*iActor=-1*)

setOpacityOverlap(*v*, *iActor=-1*)

setWireframeOverlap(*iActor=-1*)

setSurfaceOverlap(*iActor=-1*)

setRandomColours(*seed=0*)

setCutterOrigin(*dimension*, *origin*)

Parameters

- **dimension** – str - 'x', 'y', or 'z'
- **origin** – list([x,y,z])

setCutterNormal(*dimension*, *normal*)

Parameters

- **dimension** – str - 'x', 'y', or 'z'
- **normal** – list([x,y,z]) - should be unit vector

addMaterialVisOption(*materialName*, *visOptionInstance*)

Append a visualisation option instance to the dictionary of materials.

Parameters

- **materialName** – str - material name to match
- **visOptionInstance** – *VisualisationOptions* instance

setMaterialVisOptions(*materialDict*)

Replace the (by default None) dictionary for materials to colours :param materialDict: {"materialName": VisualisationOptions}

See also *VisualisationOptions*.

setCameraFocusPosition(*focalPoint=[0, 0, 0]*, *position=[100, 100, 100]*)

start()

exportOBJScene(*fileName='scene'*)

exportVRMLScene(*fileName='scene'*)

exportGLTFScene(*fileName='scene.gltf'*)

exportScreenShot(*fileName='screenshot.png'*, *rgba=True*)

Write the render window view to an image file.

Image types supported are: BMP, JPEG, PNM, PNG, PostScript, TIFF. The default parameters are used for all writers, change as needed.

Parameters

- **fileName** – The file name, if no extension then PNG is assumed.
- **renWin** – The render window.
- **rgba** – Used to set the buffer type.

Returns

addLogicalVolume(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *recursive=True*, *addWorld=True*)

addLogicalVolumeBounding(*logical*)

addSolid(*solid*, *rotation=[0, 0, 0]*, *position=[0, 0, 0]*, *representation='surface'*, *colour=[0.5, 0.5, 0.5]*, *opacity=0.2*)

Add a solid to the view with an optional rotation and translation.

Parameters

- **solid** (*any solid in pyg4ometry.geant4.solid*) – solid to add to the view
- **rotation** (*list(float, float, float) - 3 values*) – list of TB rotation angles in radians
- **position** (*list(float, float, float) - 3 values*) – translation in global from from centre in mm
- **representation** (*str*) – the way to visualise it, e.g. 'surface' or 'wireframe'
- **colour** (*list(float, float, float) - 3 values ranging from 0 - 1*) – normalised rgb colour to use for the solid mesh
- **opacity** (*float, from 0 to 1*) – the opacity of the solid if surface style

addBooleanSolidRecursive(*solid*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*, *first=True*)

Parameters

solid (pyg4ometry.geant4.solid.SolidBase) – pyg4ometry.geant4.solid instance.

Other parameters are for internal recursion and don't need to be provided.

Render only a Boolean solid. If one of the constituent solids is also a Boolean, visualise those too. The resultant Boolean is shown in solid form and each constituent in a wireframe. In the case of a null mesh, only the constituents can be shown.

addMeshSimple(*csgMesh*, *visOptions=_VisOptions()*, *clip=False*, *name='mesh'*)

addLogicalVolumeRecursive(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*)

addMesh(*pv_name*, *solid_name*, *mesh*, *mtra*, *tra*, *localmeshes*, *filters*, *mappers*, *mapperMap*, *actors*, *actorMap*, *visOptions=None*, *overlap=False*, *cutters=True*, *clippers=False*)

view(*interactive=True*, *resetCamera=True*)

_getCutterData(*axis='x'*, *scaling=1.0*)

exportCutterSection(*filename*, *normal='x'*, *scaling=1.0*)

Export the section lines in plane perpendicular to normal. Exported as json text.

Parameters

- **filename** – (str) - name of file to export to
- **normal** – (str) - one of "x", "y" or "z"
- **scaling** – (float) - multiplier for all cutter line coordinates on export

Examples

```
>>> v.exportCutterSection("xz-section.dat", normal="y", scaling=1000)
```

viewSection(*dir='x'*)

addCutterPlane(*position*, *normal*, *colour=None*)

Add a cutting plane at position=[x,y,z] with normal [nx,ny,nz].

Parameters

- **position** – [float, float, float] - (x,y,z) position in scene (mm)
- **normal** – [float, float, float] - (nx,ny,z) normal unit vector
- **colour** – None or [float, float, float] - [r,g,b] in range [0:1]

Cutters are stored in self.usercutters.

addActor(*actor*)

getOverlapVisOptions(*overlaptype*)

getMaterialVisOptions(*pv*)

_getDefaultVis(*pv*)

printViewParameters()

class pyg4ometry.visualisation.**VtkViewerColoured**(**args*, *defaultColour=None*, *materialVisOptions=None*, ***kwargs*)

Bases: *VtkViewer*

Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.

Keyword Arguments

- **materialVisOptions**: {"materialName": *VisualisationOptions* or list or tuple, ... }
- **interpolation** (str): see *VtkViewer*
- **defaultColour** (str): "random" or [r,g,b]

Examples

```
>>> vMaterialMap = VtkViewerColoured(materialVisOptions={"G4_WATER":[0,0,1]})
>>> vRandom = VtkViewerColoured(defaultColour="random")
>>> vColoured = VtkViewerColoured(defaultColour=[0.1,0.1,0.1])
>>> vColourAlpha = VtkViewerColoured(defaultColour=[0.1,0.1,0.1,0.5])
```

of use visualisation options instances

```
>>> vo = pyg4ometry.visualisation.VisualisationOptions()
>>> vo.colour = [0.1,1.0,0.5]
>>> vo.alpha = 0.3
>>> options = {'G4_WATER':vo}
>>> vis = VtkViewerColoured(materialVisOptions=options)
```

If the value in the materialVisOptions is a list or a tuple, it will be upgraded to a *VisualisationOptions* instance.

_getDefaultVis(*pv*)

pyg4ometry.visualisation.**PubViewer**

class pyg4ometry.visualisation.**VtkViewerColouredMaterial**(**args*, ***kwargs*)

Bases: *PubViewer*

Extension of VtkViewerColoured that uses a default material dictionary for several common materials. Material colours are in defined Colour.py for many Geant4, FLUKA and BDSIM materials.

class pyg4ometry.visualisation.**MouseInteractorNamePhysicalVolume**(*renderer*, *vtkviewer*)

Bases: vtk.vtkInteractorStyleTrackballCamera

rightButtonPressEvent(*obj*, *event*)

pyg4ometry.visualisation.**axesFromExtents**(*extent*)

pyg4ometry.visualisation.**viewLogicalVolumeDifference**(*referenceLV*, *otherLV*, *otherTranslation=[0, 0, 0]*, *otherRotation=[0, 0, 0]*, *viewDifference=True*)

Parameters

- **referenceLV** (*pyg4ometry.geant4.LogicalVolume.*) – LogicalVolume instance to view viewed in red.
- **referenceLV** – LogicalVolume instance to view viewed in blue.
- **otherTranslation** (*[float, float, float]*) – Translation (in native units, mm) of otherLV w.r.t. referenceLV
- **otherRotation** (*[float, float, float]*) – Rotation (in native units, rad) of other LV w.r.t. referenceLV.

View the shapes of 2 logical volumes without their contents. The reference one will be red and the 'other' one will be blue.

The other one may optionally be translated and rotated (Tait-Bryant x,y,z) relative to the reference.

class pyg4ometry.visualisation.**_VisOptions**(*representation='surface'*, *colour=[0.5, 0.5, 0.5]*, *alpha=0.5*, *visible=True*, *lineWidth=1*, *randomColour=False*, *depth=0*)

Basic holder for visualisation parameters, i.e. colour and opacity. Construct an instance then modify members.

__repr__()
Return repr(self).

getColour()
Return the colour and generate a random colour if flagged.

_generateRandomColour()

pyg4ometry.visualisation._getPredefinedMaterialVisOptions()

class pyg4ometry.visualisation.VtkViewerNew
Bases: *pyg4ometry.visualisation.ViewerBase*

initVtk()

clear()

addAxes(*length=20.0*, *origin=(0, 0, 0)*)
Add x,y,z axis to the scene.

Parameters

- **length** – float - length of each axis in mm
- **origin** – (float,float,float) - (x,y,z) of origin in mm

setAxes(*iAxes*, *length*, *origin*)

addAxesWidget()

addCutter(*name*, *origin*, *normal*)

setCutter(*name*, *origin*, *normal*)

addCutterWidget()

exportCutter(*name*, *fileName*)

getCutterPolydata(*name*)

addClipper(*origin*, *normal*, *bClipperCutter=False*, *bClipperCloseCuts=True*)

setClipper(*origin*, *normal*)

addClipperWidget()

updateClipperPlaneCallback(*obj*, *event*)

_polydata2Actor(*polydata*)

buildPipelines()

buildPipelinesSeparate()

buildPipelinesAppend()

buildPipelinesTransformed()

render()

view(*interactive=True*, *resetCamera=False*)

__repr__()

addTracks(*pd*)

addScoringMesh(*gd*)

class pyg4ometry.visualisation.**VtkViewerColouredNew**(**args*, *defaultColour=None*, *materialVisOptions=None*, ***kwargs*)

Bases: *VtkViewerNew*

Visualiser that extends VtkViewer. Uses "flat" interpolation and introduces control over colours.

Keyword Arguments

- **materialVisOptions**: {"materialName": *VisualisationOptions* or list or tuple, ... }
- **interpolation** (str): see *VtkViewer*
- **defaultColour** (str): "random" or [r,g,b]

Examples

```
>>> vMaterialMap = VtkViewerColoured(materialVisOptions={"G4_WATER":[0,0,1]})
>>> vRandom = VtkViewerColoured(defaultColour="random")
>>> vColoured = VtkViewerColoured(defaultColour=[0.1,0.1,0.1])
>>> vColourAlpha = VtkViewerColoured(defaultColour=[0.1,0.1,0.1,0.5])
```

of use visualisation options instances

```
>>> vo = pyg4ometry.visualisation.VisualisationOptions()
>>> vo.colour = [0.1,1.0,0.5]
>>> vo.alpha = 0.3
>>> options = {'G4_WATER':vo}
>>> vis = VtkViewerColoured(materialVisOptions=options)
```

If the value in the materialVisOptions is a list or a tuple, it will be upgraded to a *VisualisationOptions* instance.

_getDefaultVis(*pv*)

class pyg4ometry.visualisation.**VtkViewerColouredMaterialNew**(**args*, ***kwargs*)

Bases: *VtkViewerColouredNew*

Extension of VtkViewerColoured that uses a default material dictionary for several common materials. Material colours are in defined Colour.py for many Geant4, FLUKA and BDSIM materials.

class pyg4ometry.visualisation.**MouseInteractorNamePhysicalVolume**(*renderer*, *vtkviewer*)

Bases: vtk.vtkInteractorStyleTrackballCamera

removeHighLight()

removeHighLightText()

rightButtonPressEvent(*obj*, *event*)

class pyg4ometry.visualisation.**RenderWriter**

addLogicalVolumeRecursive(*logical*, *mtra=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *tra=_np.array([0, 0, 0])*)

addMesh(*logical*)

addInstance(*logical*, *transformation*, *translation*)

write(*outputDirectory*)

pyg4ometry.visualisation.**mkVtkIdList**(*it*)

pyg4ometry.visualisation.**pycsgMeshToVtkPolyData**(*mesh*)

pyg4ometry.visualisation.**vtkPolyDataToNumpy**(*data*)

pyg4ometry.visualisation.**pycsgMeshToObj**(*mesh*, *fileName*)

pyg4ometry.visualisation.**pyg42VtkTransformation**(*mtra*, *tra*)

pyg4ometry.visualisation.**vtkTransformation2PyG4**(*vt*)

pyg4ometry.visualisation.**pycsgMeshToStl**(*mesh*, *fileName*)

pyg4ometry.visualisation.**_WITH_PARAVIEW = False**

class pyg4ometry.visualisation.**VtkExporter**(*path='.'*)

export_to_Paraview(*reg*, *fileName='Paraview_model.pvsm'*, *model=True*, *df_model=None*, *df_color=None*)

Method that exports the visible logical volumes of the registry reg into Paraview VTK files (.vtm) and creates a Paraview State file (.pvsm) for the Paraview Model.

df_model and df_color are optional: if they are not given, the Paraview model will take the materials colors

Parameters

- **reg** – Registry
- **fileName** – Name of the Paraview State file (.pvsm)
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param df_model: (optional) pandas.DataFrame linking the NAME of the element and their TYPE :param df_color: (optional) pandas.DataFrame linking the TYPE with its specific R G B color

export_to_VTK(*reg*, *model=True*, *df_model=None*, *df_color=None*)

Method that exports the visible logical volumes of the registry reg into Paraview VTK files (.vtm).

Parameters

- **reg** – Registry
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param df_model: (optional) pandas.DataFrame linking the NAME of the element and their TYPE :param df_color: (optional) pandas.DataFrame linking the TYPE with its specific R G B color

fill_color_dico(*df_gdml*, *df_model*, *df_color*)

Method that fills a dictionary linking the logical volumes names and their respective color based on the pandas.DataFrame df_color linking the TYPE of the elements and their respective RGB color values.

Parameters

- **df_gdml** – pandas.DataFrame that represents the GDML Structure
- **df_model** – pandas.DataFrame linking the NAME of the element and their TYPE
- **df_color** – pandas.DataFrame linking the TYPE with its specific R G B color

Returns: A dictionary with the keys R, G and B whose item is a dictionary with as keys the logical volumes names and as item the respective RGB value

add_logical_world_volume(*lv*, *model*, *color_dico={'R': {}, 'G': {}, 'B': {}}*, *rotation=_np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])*, *translation=_np.array([0, 0, 0])*)

Method that receives the logical world volume and calls the necessary methods to write the .vtm file(s).

Parameters

- **lv** – Logical world volume
- **model** – Boolean informing whether we work with a "model" GDML or a "simple" GDML

True: it will consider that each daughter volume of the GDML world volume will need a .vtm file False: it will create one .vtm file for the whole GDML :param color_dico: A dictionary with the keys R, G and B whose item is a dictionary with as keys the logical volumes names and as item the respective RGB value :param rotation: (optional) numpy.matrix :param translation: (optional) numpy.array

add_logical_volume_recursive(*lv*, *rotation*, *translation*, *color_dico*, *first_level=True*)

Method that receives a logical volume and add calls addMesh() on its mesh. The method is recursive and will be called on all the daughter logical volumes of the logical volume.

Parameters

- **lv** – Logical volume
- **rotation** – numpy.matrix
- **ranslation** – numpy.array
- **color_dico** – A dictionary with the keys R, G and B whose item is a dictionary

with as keys the logical volumes names and as item the respective RGB value :param first_level: Boolean indicating if we are at the first level of recursivity

addMesh(*solid_name*, *mesh*, *rotation*, *translation*, *visOptions=None*)

Method that converts the mesh into VtkPolyData (.vtp file) and gives it the correct rotation, translation and color.

Parameters

- **solid_name** – Name of the logical volume solid
- **mesh** – Mesh of the logical volume
- **rotation** – numpy.matrix
- **translation** – numpy.array
- **visOptions** – VisualisationOptions instance

getMaterialVisOptions(*name*)

Method that "cleans" the logical volume material string.

Parameters
name (*str*) – raw name of the logical volume material

Returns: clean name of the logical volume material

getElementName(*logicalVolumeName*)

Method that "cleans" the logical volume name string.

Parameters
logicalVolumeName (*str*) – raw name of the logical volume

Returns: clean name of the logical volume

countVisibleDaughters(*lv*, *element_name*, *n=0*)

Method that counts the number of "visible" daughter logical volumes of the mother logical volume lv.

Parameters

- **lv** (pyg4ometry.geant4.LogicalVolume) – logical volume
- **element_name** (*str*) – name of the element
- **n** (*int*) – number of "visible" daughter volumes

Returns: n

5.1.2 Submodules

pyg4ometry.cli

Module Contents

Functions

_loadFile(fileName)	
_writeFile(fileName, reg)	
_parseStrMultipletAsFloat(strTriplet)	
_parseStrPythonAsSolid(reg, strPython)	
_parseStrPythonAsDict(strPython)	
_printCitation()	
cli([inputFileName, view, bounding, checkOverlaps, ...])	
main()	

pyg4ometry.cli.**_loadFile**(*fileName*)

pyg4ometry.cli.**_writeFile**(*fileName*, *reg*)

pyg4ometry.cli.**_parseStrMultipletAsFloat**(*strTriplet*)

pyg4ometry.cli.**_parseStrPythonAsSolid**(*reg*, *strPython*)

pyg4ometry.cli._parseStrPythonAsDict(*strPython*)

pyg4ometry.cli._printCitation()

pyg4ometry.cli.cli(*inputFileName=None*, *view=False*, *bounding=False*, *checkOverlaps=False*, *analysis=False*, *nullMeshException=False*, *compareFileName=None*, *appendFileName=None*, *lvName=None*, *info=None*, *exchangeLvName=None*, *clip=None*, *solid=None*, *translation=None*, *rotation=None*, *materials=None*, *outputFileName=None*, *planeCutterData=None*, *planeCutterOutputFileName=None*, *featureData=None*, *featureDataOutputFileName=None*, *gltfScale=None*, *verbose=None*)

pyg4ometry.cli.main()

pyg4ometry.config

We check that sweep angles aren't greater than 2 pi. This is the tolerance for rounding errors. The default is around float precision.

Module Contents

Classes

meshingType
SolidDefaults

Functions

backendName()
setGlobalMeshSliceAndStack(value)

Attributes

twoPiComparisonTolerance
meshing
meshingNullException
doMeshing

pyg4ometry.config.**twoPiComparisonTolerance = 1e-07**

class pyg4ometry.config.**meshingType**

pycsg = 1

cgal_sm = 2

cgal_np = 3

pyg4ometry.config.**meshing**

pyg4ometry.config.**meshingNullException = True**

pyg4ometry.config.**backendName**()

pyg4ometry.config.**doMeshing = True**

pyg4ometry.config.**setGlobalMeshSliceAndStack**(*value*)

class pyg4ometry.config.**SolidDefaults**

class Tubs

nslice = 16

class Cons

nslice = 16

class CutTubs

nslice = 16

class Ellipsoid

nslice = 8

nstack = 8

class EllipticalCone

nslice = 16

nstack = 16

class EllipticalTube

nslice = 20

nstack = 10

class GenericPolycone

nslice = 16

class Hype

nslice = 16

nstack = 16

```
class Orb
    nslice = 16
    nstack = 16
class Paraboloid
    nslice = 16
    nstack = 8
class Polycone
    nslice = 16
class Sphere
    nslice = 10
    nstack = 10
class Torus
    nslice = 50
    nstack = 10
class TwistedBox
    nstack = 20
class TwistedTrap
    nstack = 3
class TwistedTrd
    nstack = 20
class TwistedTubs
    nslice = 20
    nstack = 20
class Wedge
    nslice = 16
```

pyg4ometry.exceptions

Module Contents

exception pyg4ometry.exceptions.**NullMeshError**(*arg*)

Bases: Exception

Null Mesh exception. Should be raised when the output of mesh.toPolygons evaluates to the empty list. arg can be a user-provided message (string), or a SolidBase-derived instance.

exception pyg4ometry.exceptions.**IdenticalNameError**(*name*, *nametype=None*)

Bases: Exception

Exception for trying to add the same name to the geant4 registry.

Parameters

- **name** (*str*) – the name that has been duplicated
- **nametype** (*str*) – optional extra information regarding the nature of thecduplicate ("material", "solid", etc.)

exception pyg4ometry.exceptions.**FLUKAError**

Bases: Exception

Common base class for all non-exit exceptions.

exception pyg4ometry.exceptions.**FLUKAInputError**(*message*)

Bases: Exception

Common base class for all non-exit exceptions.

pyg4ometry.meshutils

Module Contents

Functions

MeshShrink(m[, shrinkFactor])	

pyg4ometry.meshutils.**MeshShrink**(*m*, *shrinkFactor=1e-05*)

pyg4ometry.transformation

Module Contents

Functions

rad2deg(rad)	Convert rad in radians into degrees
deg2rad(deg)	Convert deg in degrees into radians
grad2rad(gradians)	Convert rad in gradians into radians
tbxyz2axisangle(rv)	Tait-Bryan x-y-z rotation to axis-angle representation
matrix2axisangle(matrix)	Convert 3x3 transformation matrix to axis angle representation
axisangle2matrix(axis, angle)	Convert axis angle to transformation matrix
matrix2tbxyz(matrix)	Convert rotation matrix to Tait-Bryan angles.
axisangle2tbxyz(axis, angle)	Convert axis and angle to tait bryan angles
tbxyz2matrix(angles)	Convert tait bryan angles to a single passive rotation matrix.
tbzyx2matrix(angles)	Convert Tait-Bryan angles to a single passive rotation matrix.
matrix_from(v_from, v_to)	Returns the rotation matrix that rotates v_from to parallel to
_rodrigues_anti_parallel(v_from, v_to)	v_from = vector FROM
are_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
are_anti_parallel(vector_1, vector_2[, tolerance])	Check if vector vector_1 is parallel to vector vector_2 down to
reverse(angles)	Invert the rotation represented by these angles.
two_fold_orientation(v1, v2, e1, e2)	matrix_from will align one vector with another, but there are

pyg4ometry.transformation.**rad2deg**(*rad*)

Convert rad in radians into degrees

Parameters
rad (*float*) – Input in radians

Returns
rad in degrees

Return type
float

pyg4ometry.transformation.**deg2rad**(*deg*)

Convert deg in degrees into radians

Parameters
deg (*float*) – Input in degrees

Returns
deg in radians

Return type
float

pyg4ometry.transformation.**grad2rad**(*gradians*)

Convert rad in gradians into radians

Parameters
gradians (*float*) – Input in gradians

Returns
gradians in radians

Return type
float

pyg4ometry.transformation.**tbxyz2axisangle**(*rv*)

Tait-Bryan x-y-z rotation to axis-angle representation Algorithm from http://www.sedris.org/wg8home/Documents/WG80485.pdf

For converting rotation angles to an active axis/angle pair for use in pycsg. Order of rotation: x->y->z.

Parameters
rv (*float(3)*) – rotation angles

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.transformation.**matrix2axisangle**(*matrix*)

Convert 3x3 transformation matrix to axis angle representation

Parameters
matrix (*array(3,3)*) – 3x3 rotation matrix array

Returns
[axis,angle]

Return type
list(list(3),float)

pyg4ometry.transformation.**axisangle2matrix**(*axis*, *angle*)

Convert axis angle to transformation matrix

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
transformation matrix

Return type
array(3,3)

pyg4ometry.transformation.**matrix2tbxyz**(*matrix*)

Convert rotation matrix to Tait-Bryan angles. Order of rotation is x -> y -> z.

Parameters
matrix – active (positive angle = anti-clockwise rotation about that axis when looking at the axis) matrix.

Returns
[x, y, z] Tait-Bryan angles in a list.

Return type
list(3)

pyg4ometry.transformation.**axisangle2tbxyz**(*axis*, *angle*)

Convert axis and angle to tait bryan angles

Parameters

- **axis** (*list/array(3)*) – axis for rotation
- **angle** (*float*) – rotation angle

Returns
tait bryan angles (x-y-z)

Return type
array(3)

pyg4ometry.transformation.**tbxyz2matrix**(*angles*)

Convert tait bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.transformation.**tbzyx2matrix**(*angles*)

Convert Tait-Bryan angles to a single passive rotation matrix. rotation order = x -> y -> z.

Parameters
angles – list of angles: x, y, z

Returns
rotation matrix

Return type
array(3,3)

pyg4ometry.transformation.**matrix_from**(*v_from*, *v_to*)

Returns the rotation matrix that rotates v_from to parallel to v_to.

Useful for ensuring a given face points in a certain direction.

v_from and v_to should be array-like three-vectors.

pyg4ometry.transformation.**_rodrigues_anti_parallel**(*v_from*, *v_to*)

v_from = vector FROM v_to = vector TO

source: http://en.citizendium.org/wiki/Rotation_matrix#Case_that_.22from.22_and_.22to.22_vectors_are_anti-parallel

pyg4ometry.transformation.**are_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are parallel

Return type
bool

pyg4ometry.transformation.**are_anti_parallel**(*vector_1*, *vector_2*, *tolerance=1e-10*)

Check if vector vector_1 is parallel to vector vector_2 down to some tolerance.

Parameters

- **vector_1** (*array*) – First input vector
- **vector_2** (*array*) – Second input vector
- **tolerance** (*float*) – Tolerance for calculation

Returns
if vectors are antiparallel

Return type
bool

pyg4ometry.transformation.**reverse**(*angles*)

Invert the rotation represented by these angles.

pyg4ometry.transformation.**two_fold_orientation**(*v1*, *v2*, *e1*, *e2*)

matrix_from will align one vector with another, but there are an infinite number of such matrices that align two vectors. This further contrains the rotation by introducing a second pair of vectors.

v1 start v v2 end v e1 start e e2 end v

pyg4ometry.utils

Module Contents

Classes

Samples	
Timer	A class for recording the CPU time at specific positions within code

Functions

_write_pickle(obj, path)	
_load_pickle(path)	

pyg4ometry.utils.**_write_pickle**(*obj*, *path*)

pyg4ometry.utils.**_load_pickle**(*path*)

class pyg4ometry.utils.**Samples**(**metadata*)

add(*name*, *time*)

n_samples_description()

sample_names(*exclude=None*)

means(*exclude=None*)

stds(*exclude=None*)

classmethod from_existing(**samples*)

Merge two or more existing samples non-destructively to create a new Sample instance.

__str__()

Return str(self).

write(*path*)

writeAppend(*path*, *verbose=False*)

__getitem__(*key*)

__setitem__(*key*, *value*)

__contains__(*key*)

class pyg4ometry.utils.**Timer**(***metadata*)

A class for recording the CPU time at specific positions within code with the use of a name. Records time stamps in a Samples instance. To record multiple samples of a given event, simply call add with the same name repeatedly.

timer = Timer()

while True:

… timer.add("step1") … timer.add("step2")

mean_step1 = timer.samples.means()["step1"] std_step1 = timer.samples.stds()["step1"]

add(*name*)

update()

Update the last_time attribute to now.

__str__()

Return str(self).

updateTotal()

5.1.3 Package Contents

pyg4ometry.**__version__ = 'unknown version'**

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z