t4_geom_convert

Release 1.0.2

Davide Mancusi, François-Xavier Hugot, Martin Maurey, Jonathan

Apr 23, 2024

CONTENTS:

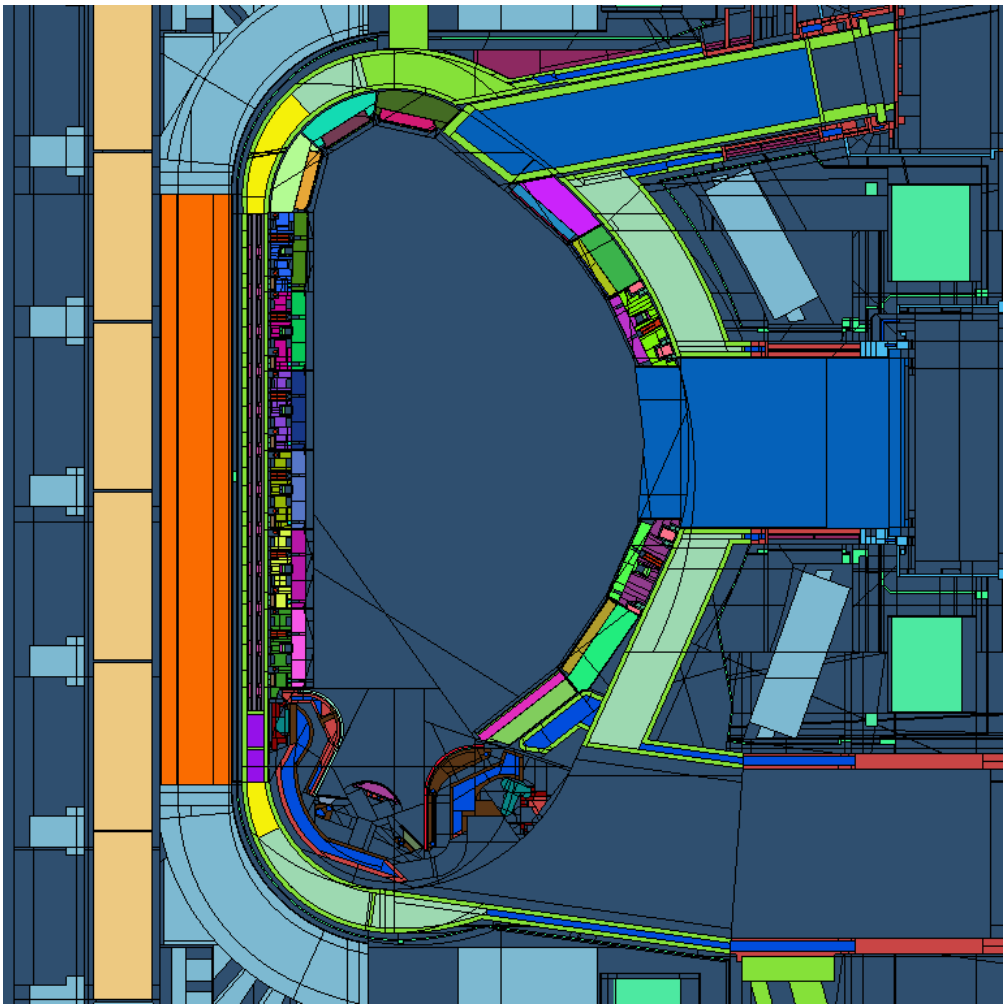


Fig. 1: A cut of the C-lite model of the ITER fusion reactor. The MCNP geometry was converted to the TRIPOLI-4 format by `t4_geom_convert`. The plot was made with T4G.

This is the documentation of `t4_geom_convert`, a tool that converts MCNP geometries into the TRIPOLI-4® format.

CHAPTER

ONE

FEATURES

Here is a list of features of the MCNP modelling engine that are at least partially supported and converted by `t4_geom_convert`:

- All surface types
- Boolean cell operators
- `LIKE n BUT` syntax
- Affine transformations on surfaces and on cells
- Boundary conditions (reflection, white surfaces)
- Isotopic compositions and cell densities
- Universes and fills, even nested, possibly with affine transformations
- Lattices

1.1 Getting started

1.1.1 Installation

`t4_geom_convert` is available on PyPI. The suggested way to install it is to use the `pip` package manager, possibly in a virtual environment so that you don't need root privileges (see the `venv` module). If `pip` and `venv` are not available on your machine, you can use your package manager to install them:

```
$ sudo apt install python3-pip python3-venv  # on Debian/Ubuntu
$ sudo yum install python3-pip python3-libs  # on CentOS 7
```

You can then create a virtual Python environment and install the latest stable version of `t4_geom_convert` there with

```
$ python3 -m venv /path/to/some/folder
$ source /path/to/some/folder/bin/activate
$ pip install -U pip setuptools
$ pip install t4_geom_convert
```

This will install the `t4_geom_convert` executable in `/path/to/some/folder/bin/t4_geom_convert`. Sourcing `/path/to/some/folder/bin/activate` will put `t4_geom_convert` in your `PATH`.

You can also install the latest development version with

```
$ pip install git+https://github.com/arekfu/t4_geom_convert.git@next
```

Dependencies

`t4_geom_convert` requires Python 3.6.

The MCNP input file is parsed by MIP. We use a slightly modified version of MIP, which is shipped along with `t4_geom_convert`. MIP depends on TatSu.

`t4_geom_convert` also depends on NumPy.

1.1.2 Usage

The basic usage is simply

```
$ t4_geom_convert <mcnp_input>
```

This will create a TRIPOLI-4 output file called `<mcnp_input>.t4` containing the converted geometry. You can also choose a different name for the output file using the `-o` option.

Use the `-h` option for a list of all available options.

Lattice conversion

`t4_geom_convert` is capable of handling the conversion of repeated structures (lattices). Hexahedral (`LAT=1`) and hexagonal (`LAT=2`) lattices are supported.

The cell declared as `LAT=1` or `LAT=2` represents the unit cell of the lattice, which is assumed to repeat in all directions up to the boundaries of the enclosing cell. Due to limitations of the TRIPOLI-4 representation of lattices, we have chosen to represent lattices using a purely surface-based approach. This means that `t4_geom_convert` will actually emit separate cell definitions for each cell of the lattice that is visible through the enclosing cell. The ranges of cell definitions to be emitted must be specified by the user via the `--lattice` command-line option. For instance, consider the following MCNP input:

```
A lattice example
1 0  -2 1 -4 3 IMP:N=1 U=2 LAT=1
10 1 -1. -10 IMP:N=1 FILL=2
1000 0 10 IMP:N=0

1 PX -1.5
2 PX 1.5
3 PY -0.5
4 PY 0.5
10 SO 4

m1 13027 1.
```

Here the unit cell is two-dimensional. The lattice fills a sphere of radius 4. Assuming the unit cell is indexed as *(0,0)*, the visible lattice cells are

- *(-1, -4)* to *(-1, 4)*
- *(0, -4)* to *(0, 4)*
- *(1, -4)* to *(1, 4)*

This can be confirmed by visual inspection of the MCNP geometry or by geometrical considerations. Once the index bounds are determined, the `--lattice` option must be specified as

```
$ t4_geom_convert --lattice 1,-1:1,-4:4 <mcnp_input>
                            ↑  ↑ ↑  ↑ ↑
              cell number ——  |  |  |  └ j-range upper bound
        i-range lower bound ——  |  |  └—— j-range lower bound
          i-range upper bound ——  |
                                  |
```

This results in the following TRIPOLI-4 geometry (X-Y cut), where a few cell indices have been annotated:

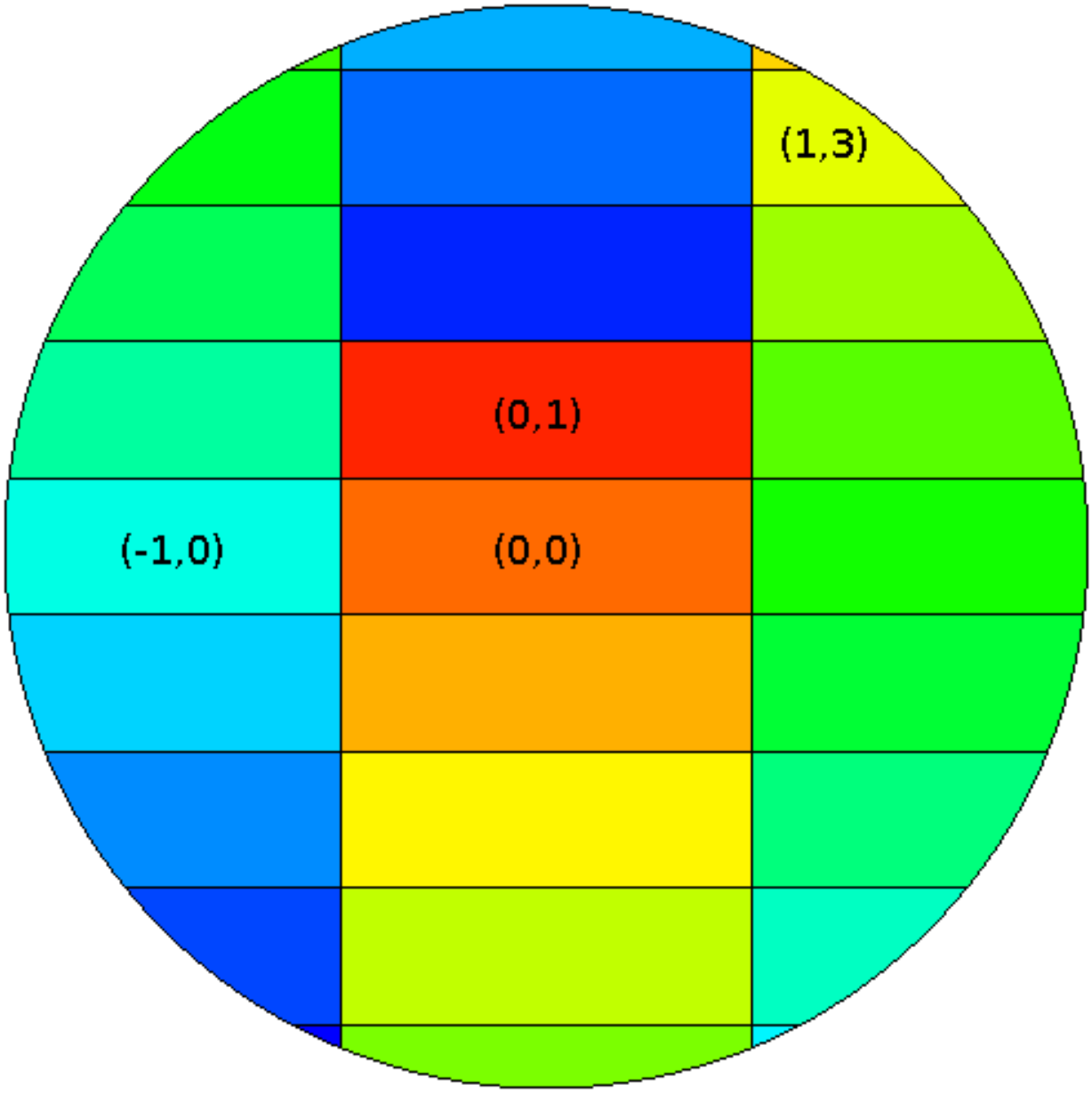


Fig. 1: Example of converted geometry with lattices.

The syntax for one-dimensional lattices is

```
--lattice <cell>,<i-from>:<i-to>
```

and for three-dimensional lattices it is

```
--lattice <cell>,<i-from>:<i-to>,<j-from>:<j-to>,<k-from>:<k-to>
```

Note that the `ijk` axes are not necessarily the same as the coordinate axes. The orientation of the lattice axes is specified by MCNP (see the *Lattice indexing* paragraph in the User's Manual) and it is determined by the order in which the surfaces of the unit cell appear specified. In our example, the first surface appearing in the definition of the unit cell is surface `2`; therefore, surface `2` separates the base cell *(0, 0)* from cell *(1, 0)*; the next surface *(``1 `)* separates the base cell from cell *(-1, 0)*; the following surfaces, `4` and `3`, separate the base cell from cells *(0, 1)* and *(0, -1)*, respectively.

For hexagonal lattices, `t4_geom_convert` follows the convention described in the MCNP manual. The lattice axes are defined by the first and the third plane appearing in the definition of the base cell. The positive direction of the third axis (if present) is defined by the normal to the seventh plane appearing in the definition.

A lattice unit cell may appear as a fill pattern in several enclosing cells. It is currently not possible to specify different fill ranges for each of them.

Fully-specified lattices

MCNP provides a syntax for the specification of lattice with heterogeneous cells. An example is

```
A lattice example
c cells
2 0  -21 u=2 imp:n=1
21 0  21 u=2 imp:n=1
3 0  -31 u=3 imp:n=1
31 0  31 u=3 imp:n=1
10 0  -2 1 3 -4  lat=1 u=20 IMP:N=1
        FILL=-1:1 -4:4
c       i=-1   i=0   i=1
        2      3     2   $ j=-4
        2      3     2   $ j=-3
        2      3     2   $ j=-2
        2      3     2   $ j=-1
        2      3     2   $ j=0
        3      3     2   $ j=1
        3      3     2   $ j=2
        3      3     2   $ j=3
        3      3     2   $ j=4
100 1 -1. -10 IMP:N=1 FILL=20
1000 0 10 IMP:N=0

1 PX -1.5
2 PX 1.5
3 PY -0.5
4 PY 0.5
10 SO 4
21 SO 0.4
31 SO 0.1

m1 13027 1.
```

The `FILL=-1:1 -4:4` option indicates the range of indices where cells will be specified. The universes filling the cells are detailed in the table below, which by convention loops over the leftmost (`i`) axis first. This syntax is supported by `t4_geom_convert` and does *not* require a `--lattice` option. One restriction applies: none of the subcells may be filled with the universe of the lattice cell itself. This syntax indicates to MCNP that the cell should be filled with the material of the lattice cell. Using `0` as a universe number for the subcells is supported and results in no subcell being generated.

Here is how the example above is converted and rendered in TRIPOLI-4:

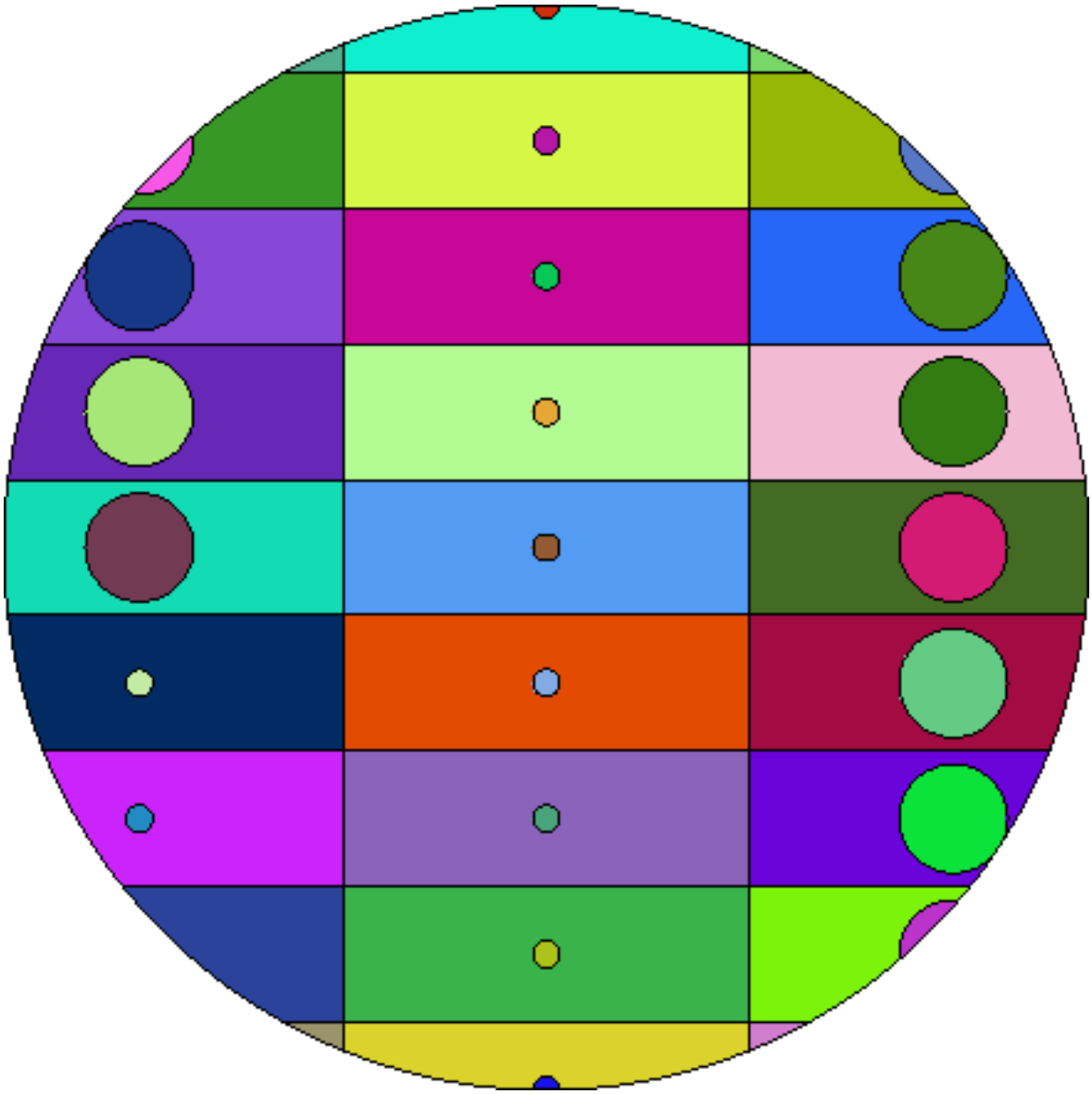


Fig. 2: Example of converted geometry with fully-specified lattice.

1.1.3 Current limitations

Here is a list of some things that `t4_geom_convert` cannot currently do, but may be able to do in the future (in roughly decreasing order of likelihood):

- Import the title of the MCNP input file (tracked in issue #5)
- Handle affine transformations with `m=-1` (the last parameter of the affine transformation) (tracked in issue #12)
- Optimize fills with negative universes (do not intersect with the enclosing cell) (tracked in issue #13)
- Warn about isotopes that are missing from the TRIPOLI-4 dictionary (currently you need to edit the converted file by hand and remove the occurrences of the missing isotopes)
- Convert cell temperatures
- Import comments describing the MCNP cells/surfaces (tracked in issue #9)
- Provide a way to specify lattice fill ranges per enclosing cell(s) (this needs to be specified in such a way that it works with nested lattices, too)
- Deduplicate repeated cell definitions (this is a bit harder than deduplicating surfaces)
- Produce a TRIPOLI-4 connectivity map for as many cells as possible (mostly lattices)
- Recognize and automatically suppress empty cells (they may be generated by lattice development or fill development)
  - Use a linear programming solver for cells bounded by planes?
  - Use a SAT solver in the general case?
- Use `PARALLELEPIPED` and `RESC` to represent rotated lattices in the T4 output. This will likely be *much* faster than representing all the cells individually.
- Convert (some) MCNP source definitions
- Convert (some) MCNP tally definitions

Your help is welcome! Feel free to open an issue if you would like to implement a new feature or contribute to the project in any way.

The full changelog is *here*.

1.1.4 Testing

The correctness of `t4_geom_convert` is tested using a specific test oracle, which is included in the source repository.

1.1.5 Reporting bugs

Please report any bug/feature request on the GitHub issues page.

1.1.6 Licence and acknowledgments

The development of `t4_geom_convert` was partially financed by the EUROfusion consortium. `t4_geom_convert` is released under the terms of the GNU General Public Licence, version 3.

1.2 MCNP/TRIPOLI-4 comparison oracle

There are a few tools to test the equivalence of an MCNP geometry and a TRIPOLI-4 geometry.

The main tool is called `oracle` and performs the equivalence tests. It writes a test report to standard output and a list of points that failed the test to an output file. The output file can subsequently be loaded in TRIPOLI-4's visualization tool T4G to get a graphical view of the problematic regions. `oracle` is mainly used as a verification tool for `t4_geom_convert`, but it can be used in other contexts, too.

When two geometries are found not to be equivalent, sometimes it is difficult to understand why. A helper tool called `explainT4` can be used to automate part of the task of figuring out what went wrong.

1.2.1 Principle

Two geometries are considered to be *equivalent* if points with the same coordinates are associated to the same material in both geometries, with the possible exception of points within some user-defined tolerance from a geometrical boundary (surface). Note that this definition does not involve cells (volumes), but only materials.

The main idea of the `oracle` tool is to use a PTRAC file produced by MCNP as a collection of results of queries to the MCNP geometry; the PTRAC file contains information about which material was seen at different locations. The advantage of using a PTRAC file is that you do not need to have access to the MCNP source files in order to query the MCNP geometry.

While a similar strategy could in principle be used to query the TRIPOLI-4 geometry, too, for the moment we have decided to use the TRIPOLI-4 geometry API directly. It is possible however to envisage that the `oracle` tool may be decoupled from TRIPOLI-4 and generalized to other geometry engines and transport codes in the future.

How do we know if the material that we find in the MCNP geometry is the same as the one that we find in TRIPOLI-4? By default, the `oracle` only checks the material names. The TRIPOLI-4 materials are expected to follow the same naming convention as those produced by `t4_geom_convert`. Specifically, if the MCNP material `m5` appears in a cell with a density of `-2.32`, the corresponding TRIPOLI-4 material should be called `m5_-2.32`. *There is a flag* (`-g`) that changes this behavior.

1.2.2 Requirements

You will need the following software to compile the comparison:

- CMake >=3.2
- a recent version of TRIPOLI-4 (sources included) and its prerequisites

You will also need an MCNP executable to run meaningful tests.

1.2.3 Compilation

All the tools are written in C++.

First, you need to compile TRIPOLI-4 using the CMake build:

```
$ mkdir /path/to/build-t4 /path/to/install-t4
$ cd /path/to/build-t4
$ cmake -DCMAKE_INSTALL_PREFIX=/path/to/install-t4 /path/to/tripoli4-sources
$ make && make install
```

Once TRIPOLI-4 has been installed, you can compile this package:

```
$ mkdir /path/to/build-oracle
$ cd /path/to/build-oracle
$ cmake -DT4_DIR=/path/to/install-t4/share/cmake /path/to/t4_geom_convert/Oracle
$ make
```

If all went well, you should find an `oracle` and an `explainT4` executable in your build directory.

1.2.4 Usage

In order to compare a TRIPOLI-4 and an MCNP geometry, you will need:

- the TRIPOLI-4 input file containing the description of the geometry (say `geometry.t4`);
- the MCNP input file (say `geometry.mcnp`)

Preparing the PTRAC file

The first thing to do is to modify the MCNP input file to generate a suitable PTRAC file.

1. Comment out the source cards (`SDEF` and friends). Add a new source that covers the portion of the geometry that you would like to test (possibly all of it). The type and energy of the source particles do not matter: the following cards, for instance, produce 14-MeV neutrons in the box `-10 < x < 1700`, `-575 < y < 575`, `-1460 < z < 1810`:

   ```
   sdef  pos=0 0 0  x=d1  y=d2  z=d3  erg=14
   si1   -10   1700
   sp1   0     1
   si2   -575  575
   sp2   0     1
   si3   -1460 1810
   sp3   0     1
   ```

   To speed things up, you may want to kill particles right below the source energy:

   ```
   cut:n  j  13.9999
   ```

   The PTRAC card should look something like this:

   ```
   ptrac  file=bin event=src max=-1000000
   ```

   Both binary (`file=bin`) and ASCII (`file=asc`) PTRAC file formats are supported by the `oracle`, but binary files are recommended (they do not drop any precision on the point coordinates, reducing the number of false

positives). The number of events to be written (`max`) can be adjusted; just make sure that you adjust the number of source particles accordingly (`nps` card).

2. Run MCNP on the modified input file.

Running the `oracle`

Let us assume the name of the PTRAC file is `geometry.ptrac`. Run the `oracle` as follows:

```
$ /path/to/oracle geometry.t4 geometry.mcnp geometry.ptrac
```

This will run the equivalence tests on the points in the PTRAC file. For each point:

- If the same material was found in the TRIPOLI-4 and MCNP geometry, the point is counted as `SUCCESSFUL`.
- If materials are different but the point was within some tolerance distance from one of the volume boundaries, the point is counted as `IGNORED`.
- If the materials are different but the point was **not** within the specified tolerance, the point is counted as `FAILED`.
- If the point did not fall inside the TRIPOLI-4 geometry, it is counted as `OUTSIDE`.

At the end of the run, you will get a report that looks like this:

```
---------------------------
Reporting on MCNP/T4 geometry comparison
---------------------------
Number of SAMPLED points : 10000
Number of SUCCESSFUL     : 9978 -> 99.78%
Number of FAILED         : 12 -> 0.12%
Number of IGNORED        : 4 -> 0.04%
Number of OUTSIDE        : 6 -> 0.06%
Number of COVERED volumes: 270
Number of INPUT   volumes: 84699
Average distance to surface for FAILED points: 6.41229e-6
Maximum distance to surface for FAILED points: 1.44246e-5
Elapsed time: 1.30655s
Time per point: 0.000130642s
```

Additional statistics are produced for the number of distinct TRIPOLI-4 volumes that were actually seen by the test, the number of *total* TRIPOLI-4 volumes in the input file (including `FICTIVE` volumes, though), the average and maximum distance from a volume boundary for failed points and the elapsed time.

The `oracle` will also produce three output files, called `geometry.failedpoints.dat`, `geometry.failedpoints.general` and `geometry.points`, which can be used to view the location of the points that failed the equivalence test in T4G.

1.2.5 Useful command-line options

Both the `oracle` and `explainT4` executables support the `-h` option for help:

```
$ /path/to/oracle -h
 *** MCNP / Tripoli-4 geometry comparison ***

oracle

  Compare MCNP and T4 geometries check that they are weakly equivalent.
  A point is assumed to match by checking the name of the composition at
  that point in each geometry.

USAGE
        oracle [options] jdd.t4 jdd.inp ptrac

INPUT FILES
        jdd.t4 .........................A TRIPOLI-4 input file converted from MCNP INP␣
↪file.
        jdd.inp ........................The MCNP INP file that was used for the␣
↪conversion.
        ptrac ..........................The MCNP PTRAC file corresponding to the INP␣
↪file.

OPTIONS
        -V, --verbose ..................Increase output verbosity.
        -h, --help .....................Displays this help message.
        -n, --npts .....................Maximum number of tested points.
        -d, --delta ....................Distance to the nearest surface below which a␣
↪failed test is ignored.
        -g, --guess-material-assocs ....guess the materials correspondence based on the␣
↪first few points
        --binary,---ascii ..............Specify the format of the MCNP PTRAC file
```

Useful options for the `oracle` executable include:

- `-V`: increase the verbosity.
- `-n NPOINTS`: limits the test run to `NPOINTS` points. There is no limit by default.
- `-d DELTA`: specifies the geometrical tolerance for ignoring mismatched materials near surfaces. Points that are within a distance `DELTA` from a volume boundary will be ignored for the purpose of counting the number of failed points. Default: `DELTA=1e-7`.
- `--binary` (default), `--ascii`: these options specify the format of the PTRAC file.
- `-g`: lets `oracle` guess the mapping between MCNP and TRIPOLI-4 materials, instead of assuming `t4_geom_convert`'s naming convention. The correspondence will be deduced on the fly: every time a new material is seen on the MCNP side, it is assumed that the material seen on the TRIPOLI-4 side is the corresponding one. Subsequent occurrences of the same MCNP materials will be checked against the TRIPOLI-4 material seen on the first point.

1.2.6 Known bugs and limitations

The oracle needs to do some rudimentary parsing of the MCNP input file. The parser is not very robust and may choke on unusual spacing, line continuations, etc.

1.3 Package documentation

1.3.1 Subpackages

t4_geom_convert.IntegrationTests package

Submodules

t4_geom_convert.IntegrationTests.test_mcnp_conversion module

Integration tests for MCNP conversion.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**do_conversion**(*mcnp_i*, *output_dir*, *conv_opts*)

Perform the actual conversion from MCNP to T4.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**do_test_oracle**(*mcnp_i*, *tmp_path*, *mcnp*, *oracle*, *oracle_zero_tolerance*)

Actually perform a conversion test, followed by an oracle test.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**get_options**(*mcnp_path*, *n_lines=50*)

Detect custom CLI options in the first few lines of an MCNP file.

;param mcnp_path: the path to the MCNP file :type mcnp_path: py.path.LocalPath :returns: the parsed options, as a list.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**test_convert**(*mcnp_i*, *tmp_path*)

Test conversion for all data files in the `data` subfolder.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**test_density_zeros**(*datadir*, *tmp_path*)

Test that the the conversion of `density_zeros.imcnp` produces only one material.

This input file contains two material cards, but the materials appear multiple times, with densities that differ only in the number of trailing zeros.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**test_extra_mcnp**(*extra_input*, *tmp_path*, *mcnp*, *oracle*, *oracle_zero_tolerance*)

Test conversion + oracle for any MCNP input files provided via the –extra-mcnp-inputs CLI option.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**test_oracle**(*mcnp_i*, *tmp_path*, *mcnp*, *oracle*, *oracle_zero_tolerance*)

Run the conversion and check the oracle for all data files in the `data` subfolder.

t4_geom_convert.IntegrationTests.test_mcnp_conversion.**test_parse_outside_points**(*datadir*)

Test that *parse_outside_points()* correctly returns the number of points outside the geometry.

Module contents

t4_geom_convert.Kernel package

Subpackages

t4_geom_convert.Kernel.BoundaryCondition package

Submodules

t4_geom_convert.Kernel.BoundaryCondition.CBoundCond module

class t4_geom_convert.Kernel.BoundaryCondition.CBoundCond.**CBoundCond**(*p_typeOfBound*)

Bases: object

classdocs

t4_geom_convert.Kernel.BoundaryCondition.CConversionBoundaryCondition module

class t4_geom_convert.Kernel.BoundaryCondition.CConversionBoundaryCondition.**CConversionBoundaryConditio**

Bases: object

conversionBoundCond()

recuperateBoundaryCondition()

Method constructing a dictionary with the id of the material as a key and the instance of CBoundCondT4 as a value

Module contents

t4_geom_convert.Kernel.Composition package

Submodules

t4_geom_convert.Kernel.Composition.Abundances module

The module containing the *Abundances* class.

class t4_geom_convert.Kernel.Composition.Abundances.**Abundances**(*isotopes*, *atom_fracs*)

Bases: object

A class that represents a set of isotopes, along with their abundances.

t4_geom_convert.Kernel.Composition.CCompositionMCNP module

class t4_geom_convert.Kernel.Composition.CCompositionMCNP.**CCompositionMCNP**(*l_materialCompositionParameters*)

Bases: object

Class which permit to access precisely to the information of the part material of the block DATA

t4_geom_convert.Kernel.Composition.CCompositionT4 module

class t4_geom_convert.Kernel.Composition.CCompositionT4.**CCompositionT4**(*p_typeDensity*, *p_material*, *p_valueOfDensity*, *l_listMaterialComposition*, *nb_atom*)

Bases: object

Class which permits to objectify each element of the T4 file.

t4_geom_convert.Kernel.Composition.CDictCompositionMCNP module

class t4_geom_convert.Kernel.Composition.CDictCompositionMCNP.**CDictCompositionMCNP**(*mcnpParser*)

Bases: MutableMapping

Class inheriting of abstract class MutableMapping and listing material composition from MCNP.

t4_geom_convert.Kernel.Composition.CompositionConversionMCNPToT4 module

t4_geom_convert.Kernel.Composition.CompositionConversionMCNPToT4.**compositionConversionMCNPToT4**(*mcnp_pars*

Function that transforms the dictionary of the composition from MCNP into a dictionary of the composition for T4.

t4_geom_convert.Kernel.Composition.CompositionConversionMCNPToT4.**str_fabs**(*number_str*)

Remove the leading minus sign (if any) from a string representing a number. It behaves in a very similar way to *str(fabs(float(number_str)))*, except that it preserves the representation of *number_str* (precision, scientific notation, etc.). The input parameter must represent a valid float.

Parameters
number_str (*str*) – a number, as a string

Returns
the absolute value of the number represented by *number_str*

Examples:

```
>>> str_fabs('-1.5')
'1.5'
>>> str_fabs('-1e-30')
'1e-30'
>>> str_fabs('-1.0e24')
'1.0e24'
>>> str_fabs('56.5')
'56.5'
```

(continues on next page)

(continued from previous page)

```
>>> str_fabs('   -44  ')  # spaces are tolerated
'44'
```

t4_geom_convert.Kernel.Composition.ConstructCompositionT4 module

t4_geom_convert.Kernel.Composition.ConstructCompositionT4.**constructCompositionT4**(*mcnp_parser*, *dic_cell_mcnp*)

Function changing the tuple from compositionConversionMCNPToT4 in instance of the VolumeT4 class.

t4_geom_convert.Kernel.Composition.ConstructCompositionT4.**extract_isotopes_fractions**(*isotopes*)

Extract the list of isotopes and the respective fractions from the MCNP parsed composition.

t4_geom_convert.Kernel.Composition.ConstructCompositionT4.**rescale_fractions**(*fractions*, *concentration*)

Rescale the given atomic fractions so that the total concentration equals the given value.

Parameters

- **fractions** (*list((str, str))*) – list of `(isotope, fraction)` pairs, as strings
- **concentration** (*float*) – the total concentration

Returns

a list of `(isotope, concentration)` pairs, as strings

Return type

list((str, str))

t4_geom_convert.Kernel.Composition.ConvertIsotope module

t4_geom_convert.Kernel.Composition.ConvertIsotope.**convert_isotope**(*isotope_id*)

Function that takes a string and returns a tuple of enum and string.

t4_geom_convert.Kernel.Composition.EIsotopeAtomicNumberMCNP module

class t4_geom_convert.Kernel.Composition.EIsotopeAtomicNumberMCNP.**EIsotopeAtomicNumber**(*value*)

Bases: Enum

An enumeration.

1 = 1

10 = 10

100 = 100

101 = 101

102 = 102

103 = 103

104 = 104

105 = 105

106 = 106

107 = 107

108 = 108

109 = 109

11 = 11

110 = 110

111 = 111

112 = 112

113 = 113

114 = 114

115 = 115

116 = 116

117 = 117

118 = 118

12 = 12

13 = 13

14 = 14

15 = 15

16 = 16

17 = 17

18 = 18

19 = 19

2 = 2

20 = 20

21 = 21

22 = 22

23 = 23

24 = 24

25 = 25

26 = 26

```
27 = 27

28 = 28

29 = 29

3 = 3

30 = 30

31 = 31

32 = 32

33 = 33

34 = 34

35 = 35

36 = 36

37 = 37

38 = 38

39 = 39

4 = 4

40 = 40

41 = 41

42 = 42

43 = 43

44 = 44

45 = 45

46 = 46

47 = 47

48 = 48

49 = 49

5 = 5

50 = 50

51 = 51

52 = 52

53 = 53

54 = 54
```

55 = 55

56 = 56

57 = 57

58 = 58

59 = 59

6 = 6

60 = 60

61 = 61

62 = 62

63 = 63

64 = 64

65 = 65

66 = 66

67 = 67

68 = 68

69 = 69

7 = 7

70 = 70

71 = 71

72 = 72

73 = 73

74 = 74

75 = 75

76 = 76

77 = 77

78 = 78

79 = 79

8 = 8

80 = 80

81 = 81

82 = 82

83 = 83

84 = 84

85 = 85

86 = 86

87 = 87

88 = 88

89 = 89

9 = 9

90 = 90

91 = 91

92 = 92

93 = 93

94 = 94

95 = 95

96 = 96

97 = 97

98 = 98

99 = 99

t4_geom_convert.Kernel.Composition.EIsotopeNameElementT4 module

class t4_geom_convert.Kernel.Composition.EIsotopeNameElementT4.**EIsotopeNameElement**(*value*)

Bases: Enum

An enumeration.

AC = 89

AG = 47

AL = 13

AM = 95

AR = 18

AS = 33

AT = 85

AU = 79

```
B = 5

BA = 56

BE = 4

BH = 107

BI = 83

BK = 97

BR = 35

C = 6

CA = 20

CD = 48

CE = 58

CF = 98

CL = 17

CM = 96

CN = 112

CO = 27

CR = 24

CS = 55

CU = 29

DB = 105

DS = 110

DY = 66

ER = 68

ES = 99

EU = 63

F = 9

FE = 26

FL = 114

FM = 100

FR = 87

GA = 31
```

```
GD = 64
GE = 32
H = 1
HE = 2
HF = 72
HG = 80
HO = 67
HS = 108
I = 53
IN = 49
IR = 77
K = 19
KR = 36
LA = 57
LI = 3
LR = 103
LU = 71
LV = 116
MC = 115
MD = 101
MG = 12
MN = 25
MO = 42
MT = 109
N = 7
NA = 11
NB = 41
ND = 60
NE = 10
NH = 113
NI = 28
```

```
NO = 102

NP = 93

O = 8

OG = 118

OS = 76

P = 15

PA = 91

PB = 82

PD = 46

PM = 61

PO = 84

PR = 59

PT = 78

PU = 94

RA = 88

RB = 37

RE = 75

RF = 104

RG = 111

RH = 45

RN = 86

RU = 44

S = 16

SB = 51

SC = 21

SE = 34

SG = 106

SI = 14

SM = 62

SN = 50

SR = 38
```

```
TA = 73

TB = 65

TC = 43

TE = 52

TH = 90

TI = 22

TL = 81

TM = 69

TS = 117

U = 92

V = 23

W = 74

XE = 54

Y = 39

YB = 70

ZN = 30

ZR = 40
```

Module contents

t4_geom_convert.Kernel.Configuration package

Module contents

t4_geom_convert.Kernel.Exceptions package

Module contents

t4_geom_convert.Kernel.FileHandlers package

Subpackages

t4_geom_convert.Kernel.FileHandlers.Parser package

Submodules

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPCell module

exception t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPCell.**MissingLatticeOptError**

Bases: Exception

An exception class to raise when the `--lattice` option is missing.

class t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPCell.**ParseMCNPCell**(*mcnp_parser*, *cell_cache_path*, *lattice_params*)

Bases: object

Class that parses the CELLS block.

LIKE_RE = re.compile('like\\s+(\\d+)\\s+but')

static apply_but(*parsed_cell*, *but_options*)

Extend the list of cell options with the BUT options.

parse()

:brief method which permit to recover the information of each line of the block CELLS :return: dictionary which contains the ID of the cells as a key and as a value, a object from the *CellMCNP* class.

parse_all_cells()

Actually parse the cells.

parse_fill_kw(*elt*, *kw_list*)

Parse the arguments of the FILL and *FILL keywords.

parse_importance_cards()

Parse any importance cards and return the maximum importance value for each cell.

Returns
the maximum importances.

Return type
list(float)

parse_keywords(*kw_list*)

Parse the list of keywords following the cell definition.

static parse_lat_kw(*kw_list*)

Parse the argument of the LAT keyword.

static parse_material(*material*)

Parse the material/density pair.

parse_one_cell(*parsed_cells*, *rank*, *lat_opt*, *parsed_cell*)

Handle the `LIKE n BUT` syntax, delegate the real parsing to *parse_one_cell_worker()*.

parse_one_cell_worker(*rank*, *lat_opt*, *parsed_cell*)

Parse one cell, return new *CellMCNP* object.

parse_trcl_kw(*elt*, *kw_list*)

Parse the arguments of the TRCL and *TRCL keywords.

static to_fillid(*kws*, *lat_opt*)

Convert the values of the fill-related keywords into a single *fillid* specification.

exception t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPCell.**ParseMCNPCellError**

Bases: Exception

An exception class for errors in MCNP cell parsing.

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPComposition module

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPComposition.**parseMCNPComposition**(*mcnpParser*)

:brief method which permit to recover the information of each line of the block SURFACE :return: dictionary which contains the ID of the materials as a key and as a value, a object from the class CCompositionMCNP

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface module

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface.**normalize_surface**(*typ*, *params*)

Put the surface parametrization in a canonical form. For instance, planes defined by three points are transformed into the equivalent (A,B,C,D) representation.

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface.**parseMCNPSurface**(*mcnp_parser*)

Function that recovers the information of each line of the block SURFACE.

Returns

dictionary with keys given by the ID of the surfaces, as a MIP Surface, and value given by lists of *(:class:* `~.SurfaceMCNP`, int)` pairs. The integer represents the side of the subsurface.

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface.**to_surface_mcnp**(*key*, *bound_cond*, *transform_id*, *enum_surface*, *params*, *transform_parsed*)

Convert the parsed surface into a *SurfaceMCNP*.

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface.**to_surfaces_macro**(*key*, *bound_cond*, *transform_id*, *enum_surface*, *params*, *transform_parsed*)

Convert the parsed macro body into a collection of *SurfaceMCNP*.

t4_geom_convert.Kernel.FileHandlers.Parser.ParseMCNPSurface.**to_surfaces_mcnp**(*key*, *parsed_surface*, *transform_parsed*)

Convert the parsed surface into a collection of *SurfaceMCNP*.

This function returns a list of `(int, SurfaceMCNP)` pairs. The integers represent the side of the surface that should be considered as positive (± 1).

Module contents

t4_geom_convert.Kernel.FileHandlers.Writer package

Submodules

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4BoundCond module

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4BoundCond.**writeT4BoundCond**(*dic_surf_mcnp*, *ofile*)

Method writing GeomComp to the T4 input file.

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4Composition module

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4Composition.**writeT4Composition**(*mcnp_parser*, *mcnp_new_dict*, *ofile*)

Function writing composition of the T4 input file.

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4GeomComp module

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4GeomComp.**writeT4GeomComp**(*dic_vol*, *mcnp_new_dict*, *ofile*)

Function writing GeomComp of the T4 input file.

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4Geometry module

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4Geometry.**convertMCNPGeometry**(*mcnp_parser*, *lattice_params*, *args*)

Convert an MCNP geometry to T4.

t4_geom_convert.Kernel.FileHandlers.Writer.WriteT4Geometry.**writeT4Geometry**(*dic_surface_t4*, *dic_volume*, *skipped_cells*, *ofile*)

Write out a T4 geometry to the given file.

Module contents

Module contents

t4_geom_convert.Kernel.GeomComp package

Submodules

t4_geom_convert.Kernel.GeomComp.CDictGeomCompT4 module

class t4_geom_convert.Kernel.GeomComp.CDictGeomCompT4.**CDictGeomCompT4**(*d_geomCompT4*)

Bases: MutableMapping

Class inheriting of abstract class MutableMapping and listing geomcomp for T4.

t4_geom_convert.Kernel.GeomComp.CGeomCompT4 module

class t4_geom_convert.Kernel.GeomComp.CGeomCompT4.**CGeomCompT4**(*p_volumeNumberMaterial*, *l_listVolumeId*)

Bases: object

Class of the object permitting to obtain information of the GeomComp of the T4 file.

t4_geom_convert.Kernel.GeomComp.ConstructGeomCompT4 module

t4_geom_convert.Kernel.GeomComp.ConstructGeomCompT4.**constructGeomCompT4**(*dicVol*, *dic_cellMCNP*)

Method constructing a dictionary with the id of the material as a key and the instance of CGeomCompT4 as a value.

Module contents

t4_geom_convert.Kernel.Surface package

Submodules

t4_geom_convert.Kernel.Surface.CTransformationMCNP module

class t4_geom_convert.Kernel.Surface.CTransformationMCNP.**CTransformationMCNP**(*l_originTransformation*, *l_rotationTransformation*)

Bases: object

Class which permit to access precisely to the information of the transformation part of the block DATA.

t4_geom_convert.Kernel.Surface.CollectionDict module

Module containing the definition of the *CollectionDict* class.

class t4_geom_convert.Kernel.Surface.CollectionDict.**CollectionDict**

Bases: MutableMapping

This class represents a dictionary mapping MCNP surfaces to lists of objects.

For example, here we fill a dictionary with a couple of surfaces:

```
>>> from MIP.geom.semantics import Surface
>>> from t4_geom_convert.Kernel.Surface.SurfaceMCNP import SurfaceMCNP
>>> import t4_geom_convert.Kernel.Surface.ESurfaceTypeMCNP as EMS
>>> MS = EMS.ESurfaceTypeMCNP
>>> planex = SurfaceMCNP('', MS.P, [1.0, 0.0, 0.0, 0.0], [])
>>> planey = SurfaceMCNP('', MS.P, [0.0, 1.0, 0.0, 0.0], [])
>>> planez = SurfaceMCNP('', MS.P, [0.0, 0.0, 1.0, 0.0], [])
>>> dic = CollectionDict()
>>> dic[1] = [(planex, 1)]
>>> dic[Surface(2)] = [(planey, -1), (planez, 1)]
```

In this example, surface 1 divides space in two regions (*x>0* and *x<0*), with the convention that *x>0* lies on the positive side of the surface.

Surface 2 consists of two sub-surfaces (the *y=0* plane and the *z=0* plane). This again divides space in two regions: the first one, which by convention is the outer side of surface 2, lies on the negative side of *planey* and on the positive side of *planez* (i.e. it is the *y<0*, *z>0* quadrant); the other region is the complement of this quadrant, that is the union of *y>0* and *z<0*.

You can query objects from the dictionary as usual:

```
>>> dic[Surface(2)]
[(SurfaceMCNP('', <ESurfaceTypeMCNP.P: 4>, (0.0, 1.0, 0.0, 0.0), (), ()), -1),␣
↪(SurfaceMCNP('', <ESurfaceTypeMCNP.P: 4>, (0.0, 0.0, 1.0, 0.0), (), ()), 1)]
```

As you probably noticed above, you can also use integers in your queries:

```
>>> dic[2] == dic[Surface(2)]
True
```

Finally, you can also query subsurfaces. Note that the numbering of the subsurfaces starts at 1 (MCNP convention):

```
>>> dic[Surface(2, sub=1)]
[(SurfaceMCNP('', <ESurfaceTypeMCNP.P: 4>, (0.0, 1.0, 0.0, 0.0), (), ()), -1)]
>>> dic[Surface(2, sub=2)]
[(SurfaceMCNP('', <ESurfaceTypeMCNP.P: 4>, (0.0, 0.0, 1.0, 0.0), (), ()), 1)]
```

Out-of-range subsurface indices are not allowed:

```
>>> dic[Surface(2, sub=-1)]
Traceback (most recent call last):
    ...
IndexError: out of range subsurface (allowed range: [1, 2])
```

You can also use two integers to query subsurfaces:

```
>>> dic[2, 2] == dic[Surface(2, sub=2)]
True
```

You can modify subsurfaces, but you must always provide a list as a value: >>> dic[2, 2] = [(planex, -1)] >>> dic[2] [(SurfaceMCNP(‘’, <ESurfaceTypeMCNP.P: 4>, (0.0, 1.0, 0.0, 0.0), (), ()), -1), (SurfaceMCNP(‘’, <ESurfaceTypeMCNP.P: 4>, (1.0, 0.0, 0.0, 0.0), (), ()), -1)]

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

number_items()

Return a numbering of the items in the collection, as well as a matching between the keys in this dictionary and the keys in the numbering.

values() → an object providing a view on D's values

t4_geom_convert.Kernel.Surface.ConstructSurfaceT4 module

t4_geom_convert.Kernel.Surface.ConstructSurfaceT4.**construct_surface_t4**(*mcnp_parser*)
Method constructing a dictionary with the id of the surface as a key and the instance of SurfaceT4 as a value.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4 module

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**conversion_surface_params**(*key*, *val*)
Convert the MCNP surface described by *val* into a TRIPOLI-4 surface.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_cone**(*key*, *val*)
Convert the parameters for cones.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_cylinder**(*val*)
Convert the parameters for cylinders.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_mcnp_surface**(*key*, *val*)
Perform the actual conversion of an MCNP surface.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_mcnp_surfaces**(*dic_surface_mcnp*)
Method which convert MCNP surface and constructing the dictionary of Surface T4.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_plane**(*val*)
Convert the parameters for planes.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_quadric**(*val*)
Convert the parameters for a quadric.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_special_quadric**(*val*)
Convert the parameters for a quadric in SQ form.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_sphere**(*val*)
Convert the parameters for spheres.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**convert_torus**(*val*)
Convert the parameters for tori.

t4_geom_convert.Kernel.Surface.ConversionSurfaceMCNPToT4.**eval_quadric**(*params*, *point*)
Evaluate the quadric represented by *params* at *point*.

```
>>> quad = [1.0, 1.0, 1.0,  # this is a sphere of radius 1.0
...         0.0, 0.0, 0.0,
...         0.0, 0.0, 0.0, -1.0]
>>> eval_quadric(quad, (0.0, 0.0, 0.0)) < 0.0
True
>>> eval_quadric(quad, (0.999, 0.0, 0.0)) < 0.0
True
>>> eval_quadric(quad, (1.001, 0.0, 0.0)) > 0.0
True
>>> eval_quadric(quad, (2.000, 0.0, 0.0)) > 0.0
True
```

t4_geom_convert.Kernel.Surface.DTypeConversion module

Module specifying each conversion of surface type.

t4_geom_convert.Kernel.Surface.Duplicates module

This module contains utilities to simplify surface dictionaries.

t4_geom_convert.Kernel.Surface.Duplicates.**remove_duplicate_surfaces**(*surfs*)

This function that detects duplicate surfaces from a surface dictionary, removes them and provides a dictionary where the IDs of the deleted surfaces are associated with the ID of the surface that replaced them.

t4_geom_convert.Kernel.Surface.Duplicates.**renumber_surfaces**(*volus*, *renumbering*)

Apply the given surface renumbering to the volume definitions.

t4_geom_convert.Kernel.Surface.ESurfaceTypeMCNP module

class t4_geom_convert.Kernel.Surface.ESurfaceTypeMCNP.**ESurfaceTypeMCNP**(*value*)

Bases: Enum

An enumeration.

ARB = 43

BOX = 33

C = 16

CX = 13

CY = 14

CZ = 15

C_X = 10

C_Y = 11

C_Z = 12

ELL = 41

GQ = 25

HEX = 37

K = 23

KX = 20

KY = 21

KZ = 22

K_X = 17

K_Y = 18

K_Z = 19

P = 4

PX = 1

PY = 2

PZ = 3

RCC = 36

REC = 39

RHP = 38

RPP = 34

S = 6

SO = 5

SPH = 35

SQ = 24

SX = 7

SY = 8

SZ = 9

T = 26

TRC = 40

TX = 27

TY = 28

TZ = 29

WED = 42

X = 30

Y = 31

Z = 32

t4_geom_convert.Kernel.Surface.ESurfaceTypeMCNP.**mcnp_to_mip**(*en*)

t4_geom_convert.Kernel.Surface.ESurfaceTypeMCNP.**string_to_enum**(*type_surface*)

t4_geom_convert.Kernel.Surface.ESurfaceTypeT4 module

class t4_geom_convert.Kernel.Surface.ESurfaceTypeT4.**ESurfaceTypeT4**(*value*)

Bases: Enum

An enumeration.

CONE = 13

CONEX = 10

CONEY = 11

CONEZ = 12

CYL = 9

CYLX = 6

CYLY = 7

CYLZ = 8

PLANE = 4

PLANEX = 1

PLANEY = 2

PLANEZ = 3

QUAD = 14

SPHERE = 5

TORUSX = 15

TORUSY = 16

TORUSZ = 17

t4_geom_convert.Kernel.Surface.MacroBodies module

exception t4_geom_convert.Kernel.Surface.MacroBodies.**MacroBodyError**(*type_*, *expected*, *params*)

Bases: Exception

Error that is raised if the number of macrobody parameters differs from the expected number.

t4_geom_convert.Kernel.Surface.MacroBodies.**arb**(*params*)

Transform the parameters of a ARB macrobody.

Parameters
params (*list*) – the thirty (!) MCNP parameters for ARB

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**box**(*params*)

Transform the parameters of a BOX macrobody.

This function is actually capable of handling generic parallelepipeds (MCNP mandates that the parallelepiped must be right).

Parameters
params (*list*) – the twelve MCNP parameters for BOX

Returns
a list of triples representing the surfaces that collectively describe the macrobody. The first element of each triple is the type of the surface; the second element is a list of coefficients; the third element is an integer (=±1) indicating on which side of the plane the macrobody lies; if the integer is +1, the **outside** of the macrobody lies in the "positive" direction for the surface.

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**check_params_length**(*type_*, *expected*, *params*)

Throw a *MacroBodyError* if the number of parameters present does not match the expected number.

t4_geom_convert.Kernel.Surface.MacroBodies.**ell**(*params*)

Transform the parameters of an ELL macrobody.

An ELL is not a generic ellipsoid, but a spheroid. It can be oblate or prolate depending on the parameters.

MCNP provides two possible ways to enter the parameters, depending on the sign of the last one. The documentation in the MCNP manual is terse and does not correspond to what the code does. For instance, the surfaces

ELL 0 0 -2 0 0 2 6 ELL 0 0 0 0 0 0 3 -2

are claimed to be equivalent, but they are not. I am not sure if the bug is in the code, the documentation or both. Anyway, the converter treats ELL like MCNP does.

Parameters
params (*list*) – the seven MCNP parameters for ELL

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**parse_facet**(*facet_int*)

Convert an integer representing a facet into a tuple of its elements.

Polyhedron facets in the MCNP input are represented as integers. Each digit represents the index of a vertex. For instance, the integer 1256 represents the facet bounded by vertices 1, 2, 5 and 6.

This function converts the integer representation into a tuple of indices, which is more convenient to work with. For example:

```
>>> parse_facet(1256)
(0, 1, 4, 5)
```

Note that the resulting indices are zero-based, which is more suitable for Python.

Zeros are ignored:

```
>>> parse_facet(5230)
(4, 1, 2)
>>> parse_facet(7024)
(6, 1, 3)
```

Parameters
facet_int (*int*) – the integer representing the facets

Returns
a tuple of indices

Return type
tuple(int)

t4_geom_convert.Kernel.Surface.MacroBodies.**rcc**(*params*)
Transform the parameters of an RCC macrobody.

Parameters
params (*list*) – the seven MCNP parameters for RCC

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**rec**(*params*)
Transform the parameters of an REC macrobody.

Parameters
params (*list*) – the ten/twelve MCNP parameters for REC

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**rhp**(*params*)
Transform the parameters of an RHP/HEX macrobody.

Parameters
params (*list*) – the fifteen MCNP parameters for RHP

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**rpp**(*params*)
Transform the parameters of an RPP macrobody.

Parameters
params (*list*) – the six MCNP parameters for RPP

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**sph**(*params*)

Transform the parameters of an SPH macrobody.

Parameters
params (*list*) – the four MCNP parameters for SPH

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**trc**(*params*)

Transform the parameters of a TRC macrobody.

Parameters
params (*list*) – the fifteen MCNP parameters for TRC

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.MacroBodies.**wed**(*params*)

Transform the parameters of a WED macrobody.

Parameters
params (*list*) – the twelve MCNP parameters for WED

Returns
see *box()*

Return type
list((*ESurfaceTypeMCNP*, list(float), int))

t4_geom_convert.Kernel.Surface.SurfaceCollection module

Module containing the *SurfaceCollection* class.

class t4_geom_convert.Kernel.Surface.SurfaceCollection.**SurfaceCollection**(*surfs*)

Bases: Sequence

A class that represents a single surface as a collection of surfaces.

This class is necessary to represent, for instance, MCNP's macrobodies or one-nappe cones.

classmethod join(*surf_colls*)

Create a *SurfaceCollection* from a list of pairs of *SurfaceCollection* objects and integers.

t4_geom_convert.Kernel.Surface.SurfaceConversionError module

Module containing the *SurfaceConversionError* class.

exception t4_geom_convert.Kernel.Surface.SurfaceConversionError.**SurfaceConversionError**

Bases: Exception

An error in surface conversion.

t4_geom_convert.Kernel.Surface.SurfaceMCNP module

class t4_geom_convert.Kernel.Surface.SurfaceMCNP.**SurfaceMCNP**(*boundary_cond*, *type_surface*, *param_surface*, *compl_param*, *idorigin=None*)

Bases: object

Class that contains the information of the MCNP surface cards.

t4_geom_convert.Kernel.Surface.SurfaceT4 module

class t4_geom_convert.Kernel.Surface.SurfaceT4.**SurfaceT4**(*type_surface*, *param_surface*, *idorigin=None*, *transform=None*)

Bases: object

Class that contains the information of the TRIPOLI-4 SURF keyword.

comment()

Return the comment string for this surface, if any.

transform_block()

Return a string representing a coordinate transformation to be applied to the T4 surface, or *None* if no transformation is required.

Module contents

t4_geom_convert.Kernel.Transformation package

Submodules

t4_geom_convert.Kernel.Transformation.Transformation module

t4_geom_convert.Kernel.Transformation.Transformation.**adjust_matrix**(*matrix*)

Perform a certain number of magical tweaks to the matrix that make sure that it is orthogonal.

MCNP tolerates some skewness but internally adjusts the matrix. We need to do the same thing, or we will not generate the same geometry.

```
>>> adjust_matrix([1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0])
[1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
```

```
>>> from math import cos, sin, isclose
>>> cos_a, sin_a = cos(1.23), sin(1.23)
>>> rot = [1.0, 0.0, 0.0, 0.0, cos_a, sin_a, 0.0, -sin_a, cos_a]
>>> rot2 = adjust_matrix(rot)
>>> all(isclose(x, y) for x, y in zip(rot, rot2))
True
```

The following matrix has too few significant digits, so *adjust_matrix()* raises a warning:

```
>>> import pytest
>>> mat = [ 0.8021,  0.1056, -0.5878,
...        -0.1305,  0.9914,  0.0000,
...         0.5828,  0.0767,  0.8090]
>>> with pytest.warns(UserWarning, match='is not orthogonal'):
...     mat2 = adjust_matrix(mat)
```

The new matrix is close to the original one, with a tolerance of 1e-4:

```
>>> all(isclose(x, y, abs_tol=1e-4) for x, y in zip(mat, mat2))
True
```

Based on rational geometrical considerations careful reading of the MCNP source code desperate debugging with gdb, the adjusted matrix generated by MCNP is:

```
>>> mat2_expected = [
...     0.80207826071476684, 0.10559100917526101, -0.58781034566441959,
...     -0.13050431946672519, 0.99144774047316964, 2.260914180407525e-05,
...     0.58278562635784137, 0.07669365483530112, 0.80899876206252763
...     ]
```

The matrix generated by `adjust_mat()` is close to the expected value:

```
>>> all(isclose(x, y) for x, y in zip(mat2_expected, mat2))
True
```

The entries are really close to the expected values!

```
>>> for x, y in zip(mat2_expected, mat2):
...     ratio = (x-y)/x
...     print(f'{x: 20.18e} {y: 20.18e} {ratio: 7.5e}')
 8.02078260714766842e-01  8.02078260714766842e-01  0.00000e+00
 1.05591009175261014e-01  1.05591009175261014e-01  0.00000e+00
-5.87810345664419587e-01 -5.87810345664419678e-01 -1.88874e-16
-1.30504319466725194e-01 -1.30504319466725194e-01 -0.00000e+00
 9.91447740473169640e-01  9.91447740473169640e-01  0.00000e+00
 2.26091418040752501e-05  2.26091418040752535e-05 -1.49857e-16
 5.82785626357841370e-01  5.82785626357841370e-01  0.00000e+00
 7.66936548353011199e-02  7.66936548353011199e-02  0.00000e+00
 8.08998762062527632e-01  8.08998762062527854e-01 -2.74468e-16
```

Parameters

matrix (*list*(*float*)) – a list of 9 floats, representing the entries of a rotation matrix.

t4_geom_convert.Kernel.Transformation.Transformation.**compose_transform**(*trans1*, *trans2*)

Compose *trans1* and *trans2*.

Returns *trans2 o trans1* (i.e. *trans1* is applied first).

t4_geom_convert.Kernel.Transformation.Transformation.**get_mcnp_transforms**(*parser*)

Return the dictionary of parsed MCNP transformation, in a canonical, 12-parameter form.

Parameters
parser – the MCNP parser

Returns
a dictionary associating each transformation number to a list of 12 transformation parameters.

t4_geom_convert.Kernel.Transformation.Transformation.**is_matrix_rowwise**(*matrix*)

Returns *True* if the matrix has at least one fully defined row.

```
>>> is_matrix_rowwise([1, 2, 3, 4, 5, 6, 7, 8, 9])
True
>>> is_matrix_rowwise([None, None, None, 4, 5, 6, 7, 8, 9])
True
>>> is_matrix_rowwise([1, 2, 3, None, None, None, 7, 8, 9])
True
>>> is_matrix_rowwise([1, 2, 3, 4, 5, 6, None, None, None])
True
>>> is_matrix_rowwise([None, 2, 3, None, 5, 6, None, 8, 9])
False
>>> is_matrix_rowwise([1, None, 3, 4, None, 6, 7, None, 9])
False
>>> is_matrix_rowwise([1, 2, None, 4, 5, None, 7, 8, None])
False
>>> is_matrix_rowwise([None, None, None, None, None, None, 7, 8, 9])
True
>>> is_matrix_rowwise([None, None, None, 4, 5, 6, None, None, None])
True
>>> is_matrix_rowwise([1, 2, 3, None, None, None, None, None, None])
True
>>> is_matrix_rowwise([None, None, 3, None, None, 6, None, None, 9])
False
>>> is_matrix_rowwise([1, None, None, 4, None, None, 7, None, None])
False
>>> is_matrix_rowwise([None, 2, None, None, 5, None, None, 8, None])
False
```

t4_geom_convert.Kernel.Transformation.Transformation.**normalize_matrix**(*matrix*)

Return a normalized version of the given 3x3 rotation matrix.

The input matrix may be missing some elements (3, 5, 6 and 9 elements are possible) or some of the elements may be *None*. The rules for normalizing a 3x3 rotation matrix are given in the MCNP6 manual, section 3.3.1.3 (TR card, Surface coordinate transformation).

t4_geom_convert.Kernel.Transformation.Transformation.**normalize_matrix3**(*matrix*)

Return the canonical form of a 3x3 matrix where exactly one row has been defined.

```
>>> from ..VectUtils import scal, rotate
>>> mat = normalize_matrix3([1.0, 0.0, 0.0,
```

(continues on next page)

(continued from previous page)

```
...                          None, None, None,
...                          None, None, None])
>>> rows = matrix_rows(mat)
>>> rows[0] == [1, 0, 0]
True
>>> scal(rows[0], rows[1])
0.0
>>> scal(rows[0], rows[2])
0.0
>>> scal(rows[1], rows[2])
0.0
>>> scal(rows[0], rows[0])
1.0
>>> scal(rows[1], rows[1])
1.0
>>> scal(rows[2], rows[2])
1.0
```

We can also check that it works with a somewhat generically aligned vector:

```
>>> vec = rotate((1.0, 0.0, 0.0), (0.0, 1.0, 0.0), 37.5)
>>> vec = rotate(vec, (0.0, 0.0, 1.0), 194.2)
>>> vec = rotate(vec, (1.0, 0.0, 0.0), -446.3)
>>> mat = normalize_matrix3([None]*3 + list(vec) + [None]*3)
>>> rows = matrix_rows(mat)
>>> rows[1] == list(vec)
True
>>> abs(scal(rows[0], rows[1])) < 1e-10
True
>>> abs(scal(rows[0], rows[2])) < 1e-10
True
>>> abs(scal(rows[1], rows[2])) < 1e-10
True
>>> abs(scal(rows[0], rows[0]) - 1.0) < 1e-10
True
>>> abs(scal(rows[1], rows[1]) - 1.0) < 1e-10
True
>>> abs(scal(rows[2], rows[2]) - 1.0) < 1e-10
True
```

t4_geom_convert.Kernel.Transformation.Transformation.**normalize_matrix5**(*matrix*)

Return the canonical form of a 3x3 matrix where exactly one row and one column have been defined.

```
>>> from math import sqrt, cos, sin
>>> def make_euler(alpha, beta, gamma):
...     s1, c1 = sin(alpha), cos(alpha)
...     s2, c2 = sin(beta), cos(beta)
...     s3, c3 = sin(gamma), cos(gamma)
...     mat = [c2, -c3*s2, s2*s3,
...            c1*s2, c1*c2*c3 - s1*s3, - c3*s1 - c1*c2*s3,
...            s1*s2, c1*s3 + c2*c3*s1, c1*c3 - c2*s1*s3]
...     return mat
```

(continues on next page)

(continued from previous page)

```
>>> full_mat = make_euler(100.0, 200.0, 300.0)
```

```
>>> some_mat = full_mat[0:4] + [None]*2 + [full_mat[6]] + [None]*2
>>> norm_mat = normalize_matrix5(some_mat)
>>> np.allclose(norm_mat, full_mat)
True
```

```
>>> some_mat = [full_mat[0]] + [None]*2 + full_mat[3:7] + [None]*2
>>> norm_mat = normalize_matrix5(some_mat)
>>> np.allclose(norm_mat, full_mat)
True
```

t4_geom_convert.Kernel.Transformation.Transformation.**normalize_matrix6**(*matrix*)

Return the canonical form of a 3x3 matrix where exactly two rows have been defined.

t4_geom_convert.Kernel.Transformation.Transformation.**normalize_transform**(*transf*)

Return a normalized, 12-param version of the affine transformation.

t4_geom_convert.Kernel.Transformation.Transformation.**to_numpy**(*trans*)

Split a transformation into a *NumPy* 3x3 matrix and a vector.

```
>>> to_numpy([1., 2., 3.,
...           1., 0., 0.,
...           0., 1., 0.,
...           0., 0., 1.])
(array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]]), array([1., 2., 3.]))
```

t4_geom_convert.Kernel.Transformation.Transformation.**transform_vector**(*trans*, *vec*)

Apply the (normalised) transformation *trans* to the vector *vec*.

If *trans* is `(b, A)`, then this function returns

v' = A*v+b

t4_geom_convert.Kernel.Transformation.Transformation.**transformation**(*trpl*, *surface*)

Apply a transformation to the given surface parameters.

Parameters

- **trpl** – the transformation
- **surface** (SurfaceMCNP) – an MCNP surface

t4_geom_convert.Kernel.Transformation.TransformationError module

Module that defines an error type for affine transformations.

exception t4_geom_convert.Kernel.Transformation.TransformationError.**TransformationError**

Bases: Exception

A class to represent errors in the application of affine transformations.

t4_geom_convert.Kernel.Transformation.TransformationQuad module

t4_geom_convert.Kernel.Transformation.TransformationQuad.**transformation_quad**(*params*, *trans*)

Apply the *trans* affine transformation to the quadric described by the *params* parameters.

Parameters

- **params** (*list(float)*) – the list of 10 quadric parameters
- **trans** (*list(float)*) – the affine transformation (12 elements)

Returns
the parameters of the transformed quadric

Return type
list(float)

Module contents

t4_geom_convert.Kernel.Volume package

Submodules

t4_geom_convert.Kernel.Volume.ByUniverse module

t4_geom_convert.Kernel.Volume.ByUniverse.**by_universe**(*mcnp_cell_dict*)

Classify MCNP cells by the universe which they belong to. Return the classification as a dictionary associating the universe number to the list of cell IDs.

t4_geom_convert.Kernel.Volume.CellConversion module

class t4_geom_convert.Kernel.Volume.CellConversion.**CellConversion**(*int_cell*, *int_surf*, *d_dictClassT4*, *d_dictSurfaceT4*, *d_dicSurfaceMCNP*, *d_dicCellMCNP*)

Bases: object

Class which contains methods to convert the Cell of MCNP in T4 Volume

apply_trcl(*trcls*, *geometry*)

Apply the given coordinate transformation to the given cell AST (*geometry*).

Returns
the transformed AST

cell_transform(*cell_key*, *transform*, *cache=True*)

Apply *transform* to *cell*, update dictionaries and return the ID of the new cell.

static conv_equa(*list_surface*)

Method converting a list of if of surface and return a tuple with the informations of the volume EQUA T4

static conv_intersection(**ids*)

Convert a T4 INTE and return a tuple with the information of the T4 VOLUME

static conv_union(**ids*)

Convert a T4 UNION and return a tuple with the information of the T4 VOLUME

static conv_union_helpers(**ids*, *union_ids*)

Convert a T4 UNION and return a tuple with the information of the T4 VOLUME

convert_cellref(*cell*, *matching*, *union_ids*)

convert_surface(*surf*, *idorigin*)

develop_lattice(*key*)

extract_surfaces(*cell*)

pot_complement(*tree*)

pot_convert(*cell*, *matching*, *union_ids*)

pot_expand_surfs(*p_tree*, *matching*)

Replace collections of surfaces with ASTs representing the intersection/union of the collection (necessary for one-nappe cones and macrobodies). Also replace MCNP surface IDs with T4 surface IDs.

pot_fill(*key*, *dict_universe*, *inline_filled=False*, *inline_filling=False*)

pot_flag(*p_tree*)

Method that takes a tree and return a tuple of tuple with flag to decorate each tree in the tree.

pot_optimise(*p_tree*)

Method that optimizes the MCNP cells.

pot_to_t4_cell(*p_tree*, *idorigin*, *matching*, *union_ids*)

Take the tree create by *pot_flag()* and fill a dictionary (of VolumeT4 instance).

pot_transform(*p_tree*, *p_transf*)

t4_geom_convert.Kernel.Volume.CellConversionError module

Module containing the definition of the *CellConversionError* class.

exception t4_geom_convert.Kernel.Volume.CellConversionError.**CellConversionError**

Bases: Exception

A class that models errors in the cell conversion process.

t4_geom_convert.Kernel.Volume.CellInlining module

Utilities for cell definition inlining.

t4_geom_convert.Kernel.Volume.CellInlining.**compute_inlining_scores**(*dic*, *occurrences*)

Associate scores for inlining to each of the cells in *occurrences*. The lower the score, the more likely the cell is to be inlined.

t4_geom_convert.Kernel.Volume.CellInlining.**extract_subcells**(*geometry*)

Extract any subcells from the given geometry tree and return them as a list.

```
>>> from .CellMCNP import CellRef
>>> extract_subcells(['*', 1, 2])
[]
>>> extract_subcells(['*', 1, CellRef(100), 2])
[100]
>>> extract_subcells(['*', 1, CellRef(100), [':', CellRef(5), CellRef(6)]])
[100, 5, 6]
```

t4_geom_convert.Kernel.Volume.CellInlining.**find_occurrences**(*dic*)

Find occurrences of inlined cells. Returns a dictionary associating a cell id *key* to a list of cell ids where *key* is inlined.

t4_geom_convert.Kernel.Volume.CellInlining.**geometry_size**(*geometry*)

Count the number of nodes in the given geometry.

```
>>> from .CellMCNP import CellRef
>>> geometry_size(['*', 1, 2])
2
>>> geometry_size(['*', 1, CellRef(100), 2])
3
>>> geometry_size(['*', 1, CellRef(100), [':', CellRef(5), CellRef(6)]])
4
```

t4_geom_convert.Kernel.Volume.CellInlining.**inline_cells**(*dic*, *max_inline_score*)

Perform inlining on the cells in *dic*.

t4_geom_convert.Kernel.Volume.CellInlining.**inline_cells_worker**(*geometry*, *dic*, *to_inline*)

Inline the cells in the *to_inline* set in *geometry*. Returns a new geometry.

t4_geom_convert.Kernel.Volume.CellMCNP module

class t4_geom_convert.Kernel.Volume.CellMCNP.**CellMCNP**(*p_materialID*, *p_density*, *syntaxTreeMCNP*, *p_importance*, *p_universe*, *fillid*, *filltr*, *lattice*, *trcl*, *idorigin=None*)

Bases: object

Class which permit to access precisely to the information of the block CELLS.

copy()

evaluateASTMCNP()

Method evaluating the syntax tree of the geometry of a cell of MCNP.

inverseASTMCNP()

Method applying the De Morgan law on a syntax tree.

class t4_geom_convert.Kernel.Volume.CellMCNP.**CellRef**(*cell*)

Bases: object

t4_geom_convert.Kernel.Volume.ConstructVolumeT4 module

t4_geom_convert.Kernel.Volume.ConstructVolumeT4.**construct_volume_t4**(*mcnp_parser*, *lattice_params*, *cell_cache_path*, *dic_surface_t4*, *dic_surface_mcnp*, *inline_filled*, *inline_filling*, *max_inline_score*)

A function that orchestrates the conversion steps for TRIPOLI-4 volumes.

t4_geom_convert.Kernel.Volume.ConstructVolumeT4.**extract_tr_surf_ids**(*mcnp_dict*)

Return the list of MCNP surface IDs above 1000.

Surface IDs above 1000 are interpreted by MCNP as (surf_id + 1000*cell_id), where cell_id indicates a cell with a trcl card. The surface is modified by applying the trcl card of the cell.

t4_geom_convert.Kernel.Volume.ConstructVolumeT4.**extract_used_surfaces**(*volumes*)

Return the IDs of the surfaces used in the given volumes, as a set.

t4_geom_convert.Kernel.Volume.ConstructVolumeT4.**remove_empty_volumes**(*dic_volume*)

Remove cells that are patently empty.

t4_geom_convert.Kernel.Volume.ConstructVolumeT4.**remove_unused_volumes**(*dic*)

Remove unused virtual (FICTIVE) volumes from the given dictionary. This function modifies the given dictionary in place.

Parameters

dic (DictVolumeT4) – a dictionary of *VolumeT4* objects.

```
>>> from .VolumeT4 import VolumeT4
>>> dic = DictVolumeT4()
>>> dic[1] = VolumeT4([], [], ops=['UNION', (2, 3)], fictive=False)
>>> dic[2] = VolumeT4([], [], ops=None, fictive=True)
>>> dic[3] = VolumeT4([], [], ops=None, fictive=True)
>>> dic[4] = VolumeT4([], [], ops=None, fictive=True)
>>> dic[5] = VolumeT4([], [], ops=None, fictive=False)
>>> remove_unused_volumes(dic)
>>> sorted(list(dic.keys()))
[1, 2, 3, 5]
```

t4_geom_convert.Kernel.Volume.DictVolumeT4 module

class t4_geom_convert.Kernel.Volume.DictVolumeT4.**DictVolumeT4**

Bases: MutableMapping

A simple wrapper around an collections.OrderedDict for storing *VolumeT4* objects.

copy()

Return a copy of *self*.

t4_geom_convert.Kernel.Volume.Lattice module

Utilities for handling lattices.

class t4_geom_convert.Kernel.Volume.Lattice.**LatticeBounds**(*bounds*)

Bases: object

A simple class to hold a list of range bounds. It provides some useful services such as the *size()* method.

copy()

Return a copy of *self*.

dims()

Return the number of non-trivial dimensions.

indices()

Yield all the valid indices in the bounds, in canonical order (loop over the leftmost index first).

```
>>> bounds = LatticeBounds([(-1, 1), (-2, 2)])
>>> list(bounds.indices())
[(-1, -2), (0, -2), (1, -2), (-1, -1), (0, -1), (1, -1), (-1, 0), (0, 0), (1,␣
↪0), (-1, 1), (0, 1), (1, 1), (-1, 2), (0, 2), (1, 2)]
>>> list(LatticeBounds([(0, 2)]).indices())
[(0,), (1,), (2,)]
```

size()

Return the total size of the range bounds, i.e. the product of the range lengths.

```
>>> LatticeBounds([(0, 4)]).size()
5
>>> LatticeBounds([(-1, 1), (-3, 3)]).size()
21
>>> LatticeBounds([]).size()
1
```

exception t4_geom_convert.Kernel.Volume.Lattice.**LatticeError**

Bases: Exception

An exception class for errors generated during lattice processing.

class t4_geom_convert.Kernel.Volume.Lattice.**LatticeSpec**(*bounds*, *spec*)

Bases: object

A simple class that holds a list of *n*m*l* integers and provides n-dimensional indexing into the list.

items()

Iterate over the lattice indices and the lattice specification, as *(indices, spec)* pairs.

```
>>> bounds = LatticeBounds([(1, 2), (0, 1)])
>>> spec = LatticeSpec(bounds, ['a', 'b', 'c', 'd'])
>>> list(spec.items())
[((1, 0), 'a'), ((2, 0), 'b'), ((1, 1), 'c'), ((2, 1), 'd')]
```

t4_geom_convert.Kernel.Volume.Lattice.**areHexSidesAdjacent**(*plane1*, *plane2*, *other_surf1*, *other_surf2*)

If *plane1* and *plane2* are adjacent in the hexagonal prism closed off by *other_plane1* and *other_plane2*, this

function returns the line given by the intersection of the two planes, in the form of a *(point, vector)* pair; otherwise, it returns *None*.

t4_geom_convert.Kernel.Volume.Lattice.**hexLatticeBaseVectors**(*surfaces*)

Compute the base vectors for a hexagonal lattice.

t4_geom_convert.Kernel.Volume.Lattice.**hexSortSides**(*surfs*)

Return an adjacency dictionary for the sides of the hexagon. The dictionary keys are pairs of indices of sides of the hexagon; the possible values are *None* if the given sides are not adjacent, or a straight line (in *(point, direction)* form) representing the intersection between the two planes if they are adjacent. The keys are always sorted in such a way that the smallest index comes first: so, for instance, *(0, 1)* is a possible key, but *(1, 0)* is not.

The *surfs* argument is the list of surfaces representing the hexagon. Each surface must be a *(plane, side)* pair, where *plane* is a plane (represented as a *(point, normal)* pair) and *side* indicates on which side of the plane the hexagonal cell lies (± 1). The surfaces in *surfs* must appear in the canonical order: first, the surface that separates the base cell of the hexagonal lattice from the *(1, 0, 0)* cell; then the opposite plane; then the surface that separates the base cell from the *(0, 1, 0)* cell; then the opposite plane; and, finally, the two remaining planes, in no particular order.

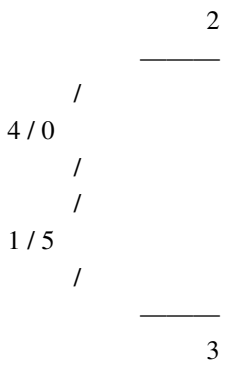
As an example, consider the regular hexagon:

```
>>> from math import cos, sin, pi, isclose, fabs
>>> vertices = [(cos(i*pi/3.), sin(i*pi/3.), 0.) for i in range(6)]
>>> sides = [vdiff(v2, v1)
...          for v1, v2 in zip(vertices, vertices[1:] + [vertices[0]])]
```

We compute the normals to the planes:

```
>>> from t4_geom_convert.Kernel.VectUtils import renorm, vect
>>> normals = [renorm(vect(side, (0, 0, 1))) for side in sides]
... # these are outgoing normals
```

The hexagon looks like this:



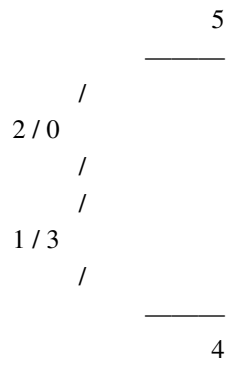
The numbers indicate the way we have chosen to order the planes. We construct the list of surfaces to respect this constraint:

```
>>> planes = [((vertices[i], normals[i]), -1)  # -1 is the side
...           for i in (0, 3, 1, 4, 2, 5)]
```

Here is the adjacency dictionary:

```
>>> adj = hexSortSides(planes)
>>> adj[(0, 2)] is not None
True
>>> adj[(0, 5)] is not None
True
>>> adj[(0, 1)] is None
True
```

We can also modify the plane numbering. For instance:




```
>>> planes = [((vertices[i], normals[i]), -1)  # -1 is the side
...           for i in (0, 3, 2, 5, 4, 1)]
>>> adj = hexSortSides(planes)
>>> adj[(0, 2)] is None
True
>>> adj[(0, 3)] is not None
True
>>> point, direction = adj[(0, 3)]
>>> (isclose(point[0], 1)
...  and isclose(point[1], 0, abs_tol=1e-10)
...  and isclose(point[2], 0, abs_tol=1e-10))
True
>>> (isclose(direction[0], 0, abs_tol=1e-10)
...  and isclose(direction[1], 0, abs_tol=1e-10)
...  and isclose(fabs(direction[2]), 1))
True
```

We can also test a weird hexagon:

```
>>> vertices = [(0, 1, 0), (3, 1, 0), (5, 2, 0),
...             (5, 3, 0), (2, 3, 0), (0, 2, 0)]
>>> sides = [vdiff(v2, v1)
...          for v1, v2 in zip(vertices, vertices[1:] + [vertices[0]])]
>>> normals = [renorm(vect(side, (0, 0, 1))) for side in sides]
```

It looks approximately like this:

```
          /———+
    3 / |
          / | 4
    | /
5 | / 2
    x———/
          1
```

The *x* represents the first vertex and the sides unfold counterclockwise. We impose the numbering given in the figure:

```
>>> planes = [((vertices[i], normals[i]), -1)  # -1 is the side
...           for i in (3, 0, 1, 4, 2, 5)]
>>> adj = hexSortSides(planes)
>>> adj[(0, 1)] is None
True
>>> adj[(0, 2)] is None
True
>>> adj[(0, 3)] is not None
True
>>> point, _ = adj[(0, 3)]
>>> point
(2.0, 3.0, 0.0)
>>> point, _ = adj[(1, 2)]
>>> point
(3.0, 1.0, 0.0)
```

t4_geom_convert.Kernel.Volume.Lattice.**hexVertices**(*surfs*, *first_side*)

Return the vertices of a base of the hexagonal prism described by the given surfaces and the direction of the prism axis.

The vertices are returned as a list of points. This function guarantees that the returned vertices are consecutive (i.e. `v[i]` and `v[i+i]` share a side, and so do `v[-1]` and `v[0]`).

The *first_side* argument is an integer from 0 to 5 that specifies which side should be shared by `v[0]` and `v[-1]`.

The prism axis is returned as a vector.

Example:

```
>>> vertices = [(0, 1, 0), (3, 1, 0), (5, 2, 0),
...             (5, 3, 0), (2, 3, 0), (0, 2, 0)]
>>> sides = [vdiff(v2, v1)
...          for v1, v2 in zip(vertices, vertices[1:] + [vertices[0]])]
>>> from t4_geom_convert.Kernel.VectUtils import renorm, vect
>>> normals = [renorm(vect(side, (0, 0, 1))) for side in sides]
>>> planes = [((vertices[i], normals[i]), -1)  # -1 is the side
...           for i in (3, 0, 1, 4, 2, 5)]
```

This is the same weird hexagonal prism that is used in the docstring for *hexSortSides()*.

Calling *hexVertices()* with `first_side=0` yields the vertices of the hexagon sorted in such a way that the first one and the last one lie on side 0:

```
>>> verts, axis = hexVertices(planes, 0)
>>> from t4_geom_convert.Kernel.VectUtils import isPointOnPlane
>>> isPointOnPlane(verts[0], planes[0][0])
True
>>> isPointOnPlane(verts[-1], planes[0][0])
True
```

Other values of *first_side* lead to different orderings of the vertices:

```
>>> verts, axis = hexVertices(planes, 4)
>>> isPointOnPlane(verts[0], planes[4][0])
True
>>> isPointOnPlane(verts[-1], planes[4][0])
True
```

We can check that each plane contains exactly two vertices:

```
>>> [sum(1 for vert in verts if isPointOnPlane(vert, plane[0]))
...  for plane in planes]
[2, 2, 2, 2, 2, 2]
```

The *axis* vector, by construction is parallel to all the side planes:

```
>>> from t4_geom_convert.Kernel.VectUtils import isVectorParallelToPlane
>>> all(isVectorParallelToPlane(axis, plane[0]) for plane in planes)
True
```

It is also possible to pass a list of eight planes. In this case, the hexagon is guaranteed to lie on the "top" plane (i.e. *surfs[-2]*).

```
>>> planes += [(((10.0, 0.0, 0.0), (1.0, 1.0, 1.0)), -1),
...            (((-10.0, 0.0, 0.0), (1.0, 1.0, 1.0)), 1)]
>>> verts, axis = hexVertices(planes, 2)
>>> all(isPointOnPlane(vert, planes[-2][0]) for vert in verts)
True
>>> [sum(1 for vert in verts if isPointOnPlane(vert, plane[0]))
...  for plane in planes[:-2]]
[2, 2, 2, 2, 2, 2]
>>> all(isVectorParallelToPlane(axis, plane[0]) for plane in planes[:-2])
True
```

t4_geom_convert.Kernel.Volume.Lattice.**latticeReciprocal**(*base_vecs*)

Yield the unit vectors of the reciprocal lattice.

Parameters

base_vecs – a list of one, two or three vectors describing the base cell of the direct lattice. Vectors are triples of the form `(x, y, z)`.

Returns

a list of one, two or three base vectors of the reciprocal lattice.

```
>>> from math import isclose
>>> from ..VectUtils import scal
```

In the case of a one-dimensional lattice, the reciprocal vector has the same direction as the base vector, but its length is equal to the inverse of the length of the direct vector: >>> vec1 = (3, 0, 4) >>> rec1 = latticeReciprocal([vec1])[0] >>> isclose(scal(rec1, vec1), 1.0) True >>> isclose(scal(rec1, rec1) * scal(vec1, vec1), 1.0) True

A few cases with two-dimensional lattices. The square lattice is self-dual: >>> vec1 = (1, 0, 0) >>> vec2 = (0, 1, 0) >>> latticeReciprocal([vec1, vec2]) [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0)]

A skew lattice: >>> vec1 = (1, 0, 0) >>> vec2 = (1, 1, 0) >>> latticeReciprocal([vec1, vec2]) [(1.0, -1.0, 0.0), (0.0, 1.0, 0.0)]

In three dimensions, the cubic lattice is self-dual: >>> vec1 = (1, 0, 0) >>> vec2 = (0, 1, 0) >>> vec3 = (0, 0, 1) >>> latticeReciprocal([vec1, vec2, vec3]) [(1.0, 0.0, 0.0), (0.0, 1.0, 0.0), (0.0, 0.0, 1.0)]

The length of the reciprocal vectors are inversely proportional to the length of the base vectors of the direct lattice: >>> vec1 = (2, 0, 0) >>> vec2 = (0, 4, 0) >>> vec3 = (0, 0, 0.5) >>> latticeReciprocal([vec1, vec2, vec3]) [(0.5, 0.0, 0.0), (0.0, 0.25, 0.0), (0.0, 0.0, 2.0)]

t4_geom_convert.Kernel.Volume.Lattice.**latticeVector**(*base_vecs*, *index*)

Compute a lattice displacement vector from a set of basis vectors and a tuple of indices.

```
>>> base_vecs = [(1, 0, 0),
...              (0, 0, 2)]
>>> latticeVector(base_vecs, (0, 0))
(0.0, 0.0, 0.0)
>>> latticeVector(base_vecs, (3, 2))
(3.0, 0.0, 4.0)
>>> latticeVector(base_vecs, (-1, -1))
(-1.0, 0.0, -2.0)
```

t4_geom_convert.Kernel.Volume.Lattice.**parse_ranges**(*intervals*)

Parse a list of intervals (in the MCNP syntax: `start:end`) into a list of pairs of integers.

Parameters
intervals (*list(str)*) – A list of strings of the form `'i:j'`, where *i* and *j* are integers.

Returns
a list of pairs of integers representing the parsed bounds

Return type
list((int,int))

```
>>> parse_ranges(['0:4', '0:4', '0:4'])
[(0, 4), (0, 4), (0, 4)]
>>> parse_ranges(['1:2', '3:4', '5:6', '7:8'])
[(1, 2), (3, 4), (5, 6), (7, 8)]
>>> parse_ranges([])
[]
>>> parse_ranges(['0::5'])
Traceback (most recent call last):
    ...
ValueError: needs exactly 2 colon-separated range bounds in argument '0::5'
```

t4_geom_convert.Kernel.Volume.Lattice.**squareLatticeBaseVectors**(*surfaces*)

Compute the base vectors of a square lattice.

t4_geom_convert.Kernel.Volume.Lattice.**squareLatticeReciprocalVecs**(*surfaces*)

Compute the reciprocal vectors of a square lattice.

t4_geom_convert.Kernel.Volume.TreeFunctions module

t4_geom_convert.Kernel.Volume.TreeFunctions.**isCellRef**(*tree*)

Returns *True* if *tree* is a *CellRef*.

Return type
bool

t4_geom_convert.Kernel.Volume.TreeFunctions.**isIntersection**(*tree*)

Function that tells if a node is an intersection.

Return type
bool

t4_geom_convert.Kernel.Volume.TreeFunctions.**isLeaf**(*tree*)

Function which tells if a tree is an instance of a Surface or a Geometry.

Return type
bool

t4_geom_convert.Kernel.Volume.TreeFunctions.**isSurface**(*tree*)

Returns *True* if *tree* is a surface.

Return type
bool

t4_geom_convert.Kernel.Volume.TreeFunctions.**isUnion**(*tree*)

Function that tells if a node is a union.

Return type
bool

t4_geom_convert.Kernel.Volume.TreeFunctions.**largestPureIntersectionNode**(*nodes*)

Returns the index of the largest node of the *nodes* list that is an intersection of surfaces, or *None* if no such node is present.

```
>>> from .CellMCNP import CellRef
>>> largestPureIntersectionNode([[2, '*', 1, 2], [3, '*', 4, 5, 6]])
1
>>> largestPureIntersectionNode([4, 5, 6])
0
>>> largestPureIntersectionNode([[2, '*', 1, 2],
...                              [3, '*', 4, 5, 6],
...                              [4, ':', 7, 8, 9, 10]])
1
>>> largestPureIntersectionNode([[3, '*', 4, 5, 6],
...                              [2, '*', 1, 2],
...                              [4, ':', 7, 8, 9, 10]])
0
>>> largestPureIntersectionNode([[2, ':', 1, 2], [3, ':', 4, 5, 6]])
>>> largestPureIntersectionNode([[3, '*', CellRef(4), 5, 6],
...                              [2, '*', 1, 2]])
1
```

t4_geom_convert.Kernel.Volume.VolumeT4 module

class t4_geom_convert.Kernel.Volume.VolumeT4.**VolumeT4**(*pluses*, *minuses*, *ops=None*, *idorigin=None*, *fictive=True*)

Bases: object

Class which permits to access precisely of the value of a volume T4.

comment()

copy()

Return a copy of *self*.

empty()

Return *True* if the cell is patently empty, i.e. if the same surface ID appears with opposite signs.

surface_ids()

Return the surface IDs used in this volume, as a set.

Module contents

Submodules

t4_geom_convert.Kernel.Progress module

Implementation of a simple progress meter.

class t4_geom_convert.Kernel.Progress.**Progress**(*message*, *n_items*, *longest_item*)

Bases: object

A simple progress Meter.

update(*iteration*, *item*)

Update the object that we have reached *iteration* and that we are treating *item*.

The *Progress* object will print to stdout if we have gained at least one percentage point.

t4_geom_convert.Kernel.VectUtils module

t4_geom_convert.Kernel.VectUtils.**isPointOnPlane**(*point*, *plane*, *, *tol=1e-10*)

Returns *True* if the point lies on the plane within the specified tolerance.

```
>>> point = (1.0, 1.0, 1.0)
>>> plane = ((3.0, 0.0, 0.0), (1.0, 1.0, 1.0))
>>> isPointOnPlane(point, plane)
True
```

t4_geom_convert.Kernel.VectUtils.**isVectorParallelToPlane**(*vector*, *plane*, *, *tol=1e-10*)

Returns *True* if the vector is parallel to the plane within the specified tolerance.

```
>>> vector = (1.0, 1.0, -2.0)
>>> plane = ((0.0, 0.0, 0.0), (1.0, 1.0, 1.0))
>>> isVectorParallelToPlane(vector, plane)
True
>>> isVectorParallelToPlane(rescale(2.0, vector), plane)
True
>>> isVectorParallelToPlane(plane[1], plane)
False
```

t4_geom_convert.Kernel.VectUtils.**mag**(*vec*)

Return the magnitude of *vec*.

t4_geom_convert.Kernel.VectUtils.**mag2**(*vec*)

Return the square of the magnitude of *vec*.

t4_geom_convert.Kernel.VectUtils.**matrix_rows**(*matrix*)

Transform a 3x3 matrix into the list of its rows.

```
>>> mat = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> matrix_rows(mat)
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

t4_geom_convert.Kernel.VectUtils.**mixed**(*v1*, *v2*, *v3*)

Yields the mixed product of *v1*, *v2* and *v3*.

The mixed product is defined as $v1 \cdot (v2 \times v3)$ and is equal to the determinant of the matrix having the components of *v1*, *v2* and *v3* as rows.

t4_geom_convert.Kernel.VectUtils.**planeParamsFromNormalAndPoint**(*normal*, *point*)

Return the MCNP-style parameters of the plane having the given normal and passing through the given points.

t4_geom_convert.Kernel.VectUtils.**planeParamsFromPoints**(*pt1*, *pt2*, *pt3*)

Compute the parameters *(a ,b, c, d)* of the plane passing through the three given points *pt1*, *pt2*, *pt3*.

The equation of the plane is written in the MCNP form as

$$ax + by + cz - d = 0$$

Furthermore, the normal to the plane is oriented in such as way that the origin has negative sense (this is the MCNP convention).

t4_geom_convert.Kernel.VectUtils.**planeSide**(*point*, *plane*)

Returns 1 if point lies on the positive side of the plane, -1 if it lies on the negative side and 0 if it lies on the plane (no numerical tolerance).

t4_geom_convert.Kernel.VectUtils.**pointInPlaneIntersection**(*plane1*, *plane2*)

Construct a point in the intersection of two planes. The planes must be given as a *(point, normal)* pair.

```
>>> from math import isclose
>>> def is_in_plane(point, plane):
...     pt, norm = plane
...     return isclose(scal(vdiff(point, pt), norm), 0., abs_tol=1e-10)
```

```
>>> plane1 = ((0, 0, 0), (1, 0, 0))
>>> plane2 = ((0, 0, 0), (0, 1, 0))
>>> pointInPlaneIntersection(plane1, plane2)
((0.0, 0.0, 0.0), (0.0, 0.0, 1.0))
```

```
>>> plane1 = ((0, 0, 0), (1, 0, 0))
>>> plane2 = ((0, 5, 0), (0, 1, 0))
>>> int_p, line_vec = pointInPlaneIntersection(plane1, plane2)
>>> is_in_plane(int_p, plane1) and is_in_plane(int_p, plane2)
True
>>> isclose(scal(line_vec, (1, 0, 0)), 0., abs_tol=1e-10)
True
>>> isclose(scal(line_vec, (0, 1, 0)), 0., abs_tol=1e-10)
True
```

t4_geom_convert.Kernel.VectUtils.**projectPointOnPlane**(*point*, *plane*, *direction*)

Project a point on a plane along the given direction.

```
>>> from math import isclose, sqrt
>>> point = (0, 0, 3)
>>> plane = ((0, 0, 0), (0, 0, 1))
>>> direction = (1, 0, 1)
>>> proj = projectPointOnPlane(point, plane, direction)
>>> expected = (-3, 0, 0)
>>> all(isclose(p, e, abs_tol=1e-10) for p, e in zip(proj, expected))
True
```

```
>>> point = (0, 0, 3)
>>> plane = ((0, 0, 1), (0, 0, 1))
>>> direction = (2, 0, 1)
>>> proj = projectPointOnPlane(point, plane, direction)
>>> expected = (-4, 0, 1)
>>> all(isclose(p, e, abs_tol=1e-10) for p, e in zip(proj, expected))
True
```

t4_geom_convert.Kernel.VectUtils.**renorm**(*vec*, *norm=1.0*)

Return a new vector parallel to *vec* whose norm is equal to *norm*.

t4_geom_convert.Kernel.VectUtils.**rescale**(*a*, *v1*)

Return *v1* multiplied by a scalar *a*, as a new vector.

t4_geom_convert.Kernel.VectUtils.**rotate**(*vec*, *axis*, *angle*)

Rotate vector *vec* around the axis *axis* by angle *angle*, according to the right-hand rule.

This function uses Rodrigues' rotation formula. If we denote *vec* as v, the angle as and the axis as k, the formula reads:

$$v' = v\cos(\theta) + (k^v)\sin(\theta) + k(k \cdot v)(1 - \cos(\theta))$$

Examples:

```
>>> from math import pi
>>> vec = (1, 1, 0)
```

(continues on next page)

(continued from previous page)

```
>>> axis = (0, 0, 1)
>>> rot_vec = rotate(vec, axis, 0.5*pi)
>>> print('({:.5f}, {:.5f}, {:.5f})'.format(*rot_vec))
(-1.00000, 1.00000, 0.00000)
>>> rot_vec = rotate(vec, axis, 0.25*pi)
>>> print('({:.5f}, {:.5f}, {:.5f})'.format(*rot_vec))
(0.00000, 1.41421, 0.00000)
```

Parameters

- **vec** – the vector to rotate
- **axis** – the rotation axis (must be a unit vector)
- **angle** – the rotation angle, in radians

t4_geom_convert.Kernel.VectUtils.**rotation_from_vectors**(*vector_from*, *vector_to*)

Return a rotation matrix transforming *vector_from* into a vector parallel to *vector_to*.

```
>>> rotation_from_vectors((0.0, 0.0, 1.0), (0.0, 0.0, 1.0))
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> rotation_from_vectors((0.0, 0.0, 3.0), (0.0, 0.0, 5.0))
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
>>> rotation_from_vectors((0.0, 0.0, 1.0), (0.0, 1.0, 0.0))
array([[ 1.,  0.,  0.],
       [ 0.,  0.,  1.],
       [ 0., -1.,  0.]])
```

t4_geom_convert.Kernel.VectUtils.**scal**(*v1*, *v2*)

Yields the scalar product of *v1* and *v2*.

t4_geom_convert.Kernel.VectUtils.**transpose**(*matrix*)

Transpose a 3x3 matrix.

```
>>> mat = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> transpose(mat)
[1, 4, 7, 2, 5, 8, 3, 6, 9]
```

t4_geom_convert.Kernel.VectUtils.**vdiff**(*v1*, *v2*)

Return the vector difference of *v1* and *v2* (*v1*-*v2*).

t4_geom_convert.Kernel.VectUtils.**vect**(*v1*, *v2*)

Yields the vector product of *v1* and *v2*.

t4_geom_convert.Kernel.VectUtils.**vsum**(**args*)

Return the vector sum of its arguments.

Module contents

t4_geom_convert.UnitTests package

Subpackages

t4_geom_convert.UnitTests.CompositionTest package

Submodules

t4_geom_convert.UnitTests.CompositionTest.test_CompositionConversionMCNPToT4 module

Tests for the *CompositionConversionMCNPToT4* module.

t4_geom_convert.UnitTests.CompositionTest.test_CompositionConversionMCNPToT4.**formats**() → SearchStrategy

Generate a random Python format specification for floats.

Examples:

- {:8.14f}
- {:.13e}
- {:15.3g}

t4_geom_convert.UnitTests.CompositionTest.test_CompositionConversionMCNPToT4.**test_str_fabs_conversion**() → None

Test that *str_fabs()* correctly converts floats.

t4_geom_convert.UnitTests.CompositionTest.test_CompositionConversionMCNPToT4.**test_str_fabs_spaces**() → None

Test that *str_fabs()* is robust with respect to space and format specifications.

Module contents

t4_geom_convert.UnitTests.VolumeTest package

Submodules

t4_geom_convert.UnitTests.VolumeTest.test_Lattice module

Unit tests for the *Lattice* module.

t4_geom_convert.UnitTests.VolumeTest.test_Lattice.**latticeReciprocalTest**(*vecs*)

Call *latticeReciprocal()*, and additionally test that the list of vectors it returns has the same length as the argument.

t4_geom_convert.UnitTests.VolumeTest.test_Lattice.**lattices**() → SearchStrategy

Generate base vectors for a 1D, 2D or 3D square lattice. Make sure that they are not collinear.

t4_geom_convert.UnitTests.VolumeTest.test_Lattice.**test_bravais_property**() → None
Test that the reciprocal vectors of a 1D, 2D or 3D lattice satisfy the Bravais property:

$$v_i \cdot r_j = delta_{ij}, i, j = 1, 2, 3$$

t4_geom_convert.UnitTests.VolumeTest.test_Lattice.**vectors**() → SearchStrategy
Generate a random three-dimensional vector.

Module contents

Submodules

t4_geom_convert.UnitTests.test_VectUtils module

Tests for the *VectUtils* module.

t4_geom_convert.UnitTests.test_VectUtils.**is_in_plane**(*point*, *plane*)
Returns *True* if the point lies in the given plane.

t4_geom_convert.UnitTests.test_VectUtils.**is_parallel**(*vector1*, *vector2*)
Returns *True* if the vectors are parallel

t4_geom_convert.UnitTests.test_VectUtils.**test_intersection**() → None
Test that the point generated by *pointInPlaneIntersection()* always lies in both planes.

t4_geom_convert.UnitTests.test_VectUtils.**test_rotation_from_vectors**() → None
Test that *rotation_from_vectors()* actually rotates *vector1* on top of *vector2*.

t4_geom_convert.UnitTests.test_VectUtils.**test_rotation_unitary**() → None
Test that the matrix given by *rotation_from_vectors()* is unitary.

t4_geom_convert.UnitTests.test_VectUtils.**vectors**(*, *norm=None*) → SearchStrategy
Generate random vectors, possibly with a given norm.

Module contents

1.3.2 Submodules

1.3.3 t4_geom_convert.Debug module

Useful tools for debugging.

t4_geom_convert.Debug.**debug**(*wrapped*)
Prints input and output values of any function it decorates.

1.3.4 t4_geom_convert.conftest module

pytest configuration file.

class t4_geom_convert.conftest.**MCNPRunner**(*path*, *work_path*)

Bases: object

A helper class to run MCNP on a given input file.

run(*input_file*)

Run MCNP on the given input file.

Parameters
input_file (*str*) – absolute path to the input file

Returns
the paths to the generated output and PTRAC files

Return type
(pathlib.Path, pathlib.Path)

class t4_geom_convert.conftest.**OracleRunner**(*path*, *work_path*)

Bases: object

A helper class to run the test oracle.

run(*t4_o*, *mcnp_i*, *mcnp_ptrac*, *oracle_opts=None*)

Run the test oracle on the given files.

Parameters

- **t4_o** (*str*) – absolute path to the TRIPOLI-4 file to test
- **mcnp_i** (*str*) – absolute path to the MCNP input file
- **mcnp_ptrac** (*str*) – absolute path to the MCNP PTRAC file

Returns
the number of failed points in the comparison

t4_geom_convert.conftest.**check_failed_points**(*failed_path*)

Read the *failed_path* file and return the number of failed points and the maximum distance for them.

t4_geom_convert.conftest.**datadir**(*tmp_path*, *request*)

Fixture responsible for searching a folder called 'data' in the same directory as the test module and, if available, moving all contents to a temporary directory so tests can use them freely.

t4_geom_convert.conftest.**foreach_data**(**args*, ***kwargs*)

Decorator that parametrizes a test function over files in the data directory for the current tests.

Assume that the following snippet resides in `tests/submod/test_submod.py`:

```
@foreach_data('datafile')
def test_something(datafile):
    pass
```

When *pytest* imports `test_submod.py`, it will parametrize the *datafile* argument to `test_something()` over all the files present in `tests/submod/data/`.

If you wish to filter away some of the files, you can use the alternative syntax:

```
@foreach_data(datafile=lambda path: str(path).endswith('.txt'))
def test_something(datafile):
    pass
```

Here the argument to the *datafile* keyword argument is a predicate that must return *True* if *path* is to be parametrized over, and *False* otherwise. Note that the *path* argument to the lambda is a `pathlib.Path` object. In this example, *pytest* will parametrize `test_something()` only over files whose name ends in `'.txt'`.

t4_geom_convert.conftest.**mcnp**(*mcnp_path*, *tmp_path*)

Return an instance of the *MCNPRunner* class.

t4_geom_convert.conftest.**mcnp_path**(*request*)

Fixture yielding the path to the MCNP executable specified on the command line.

t4_geom_convert.conftest.**oracle**(*oracle_path*, *tmp_path*)

Return an instance of the *OracleRunner* class.

t4_geom_convert.conftest.**oracle_path**(*request*)

Fixture yielding the path to the oracle executable specified on the command line.

t4_geom_convert.conftest.**parse_outside_points**(*stdout_path*)

Parse the file at *stdout_path* and return the number of points that fall outside the geometry.

1.3.5 t4_geom_convert.main module

t4_geom_convert.main.**conversion**(*args*)

Orchestrate the conversion.

t4_geom_convert.main.**main**()

Main entry point for the CLI tool.

t4_geom_convert.main.**parse_args**(*argv*)

Parse the command-line arguments.

Returns

a namespace containing the parsed arguments.

t4_geom_convert.main.**parse_lattice**(*lattice_list*)

Check the arguments to the –lattice option and parse them into a usable form.

The arguments to –lattice (the option may be given multiple times) have the form *cell*`,`*i_min*`:`*i_max*`[,`*j_min*`:`*j_max*`[,`*k_min*`:`*k_max*`]]`

```
>>> opt_lattice = ['200,2:5,0:4', '5902,0:5,0:5,0:5', '10,-4:4']
```

This function parses the *opt_lattice* list into a dictionary associating cell numbers to a list of ranges, as tuples:

```
>>> dic = parse_lattice(opt_lattice)
>>> dic == {200: [(2, 5), (0, 4)],
...         5902: [(0, 5), (0, 5), (0, 5)],
...         10: [(-4, 4)]}
True
```

The function does some error checking, too:

```
>>> parse_lattice(['malformed'])
Traceback (most recent call last):
    ...
ValueError: no ranges specified in option 'malformed'
>>> parse_lattice(['three,-1:5'])
Traceback (most recent call last):
    ...
ValueError: cell number 'three' is not an integer in option 'three,-1:5'
>>> parse_lattice(['100,'])
Traceback (most recent call last):
    ...
ValueError: needs exactly 2 colon-separated range bounds in argument '' in option
↪'100,'
>>> parse_lattice(['100,0:4,0:4,0:4,0:4'])
Traceback (most recent call last):
    ...
ValueError: too many ranges specified in option '100,0:4,0:4,0:4,0:4'
>>> parse_lattice(['100,0:6.022e23'])
Traceback (most recent call last):
    ...
ValueError: range bound '6.022e23' is not an integer in option '100,0:6.022e23'
>>> parse_lattice(['100,-6.022e23:0'])
Traceback (most recent call last):
    ...
ValueError: range bound '-6.022e23' is not an integer in option '100,-6.022e23:0'
```

t4_geom_convert.main.**writeHeader**(*ofile*)

Write a short header for the TRIPOLI-4 output file.

1.3.6 Module contents

1.4 Changelog

1.4.1 v1.0.0

- Handle surface IDs >= 1000 as implicitly defined, if not given in the input file.

  Surface IDs >= 1000 have a special meaning in MCNP. They are interpreted as (surf_id + 1000*cell_id), where cell_id refers to a cell card and surf_id refers to another surface card. For example, surface ID 123456 refers to cell 123 and surface 456. The meaning of this syntax is that surface 123456 is obtained by applying the transformation defined in the TRCL card of cell 123 to surface 456.

- Print the real MCNP cell number in the T4 comments whenever possible.

1.4.2 v0.6.0

- Python 3.8 required.
- Implement torus rotations (fixes issue #34).
- Fix adjustment of imprecise rotation matrices.

1.4.3 v0.5.0

- Python 3.6 required.
- Add caching for cell definitions. This reduces the redundance of the generated TRIPOLI-4 output files and usually results in faster conversion and faster TRIPOLI-4 execution.
- Add CLI options to control cell inlining.
- Fix conversion of isotopes starting with a zero.
- Handle complement of lattice base cells.
- Fix the conversion of cells with TRCL and FILL.
- Move to poetry for package creation.

1.4.4 v0.4.0

- Implement conversion of `SQ` surfaces (fixes issue #11)
- Improve error reporting (works towards fixing issue #23, although I doubt this issue can ever be closed).
- Correctly support MCNP input files containing tabs.
- Support rotation matrices in short form (3 parameters, 5 parameters, 6 parameters).
- Largely reduce the memory footprint of the conversion process; this is crucial for largish (a few GB) output files.
- Add a flag (`--skip-deduplication`) to skip surface deduplication.
- Support references to macrobody facets in cell definitions.
- Fix a few bugs.

1.4.5 v0.3.0

- Implement conversion of the `LIKE n BUT` syntax (fixes issue #26)
- Implement conversion of hexagonal lattices (`LAT=2`, fixes issue #22)
- Implement conversion of macrobodies (fixes issue #25)
- Add a command-line option (-e) to specify the encoding of the input file (fixes issue #27)
- Handle the MCNP data card shortcuts (`14R`, `J`, `I`, etc., fixes issue #24)
- Support importance keywords for multiple particles
- Support “message:” MCNP cards
- Support affine transformations with 13 parameters
- Improve error message on TatSu parse errors (partially fixes issue #23)

- Add integration tests with MCNP and Oracle (fixes issue #6)
- Add documentation for the Oracle tooling
- Add a message with the list of the cells that were skipped during the conversion
- Bug fixes:
  - Fix parsing of consecutive complement operators (`#1#2`, fixes issue #33)
  - Fix handling of TRCL keyword on cells defined using the complement operator (fixes issue #32)
  - Fix several issues with line continuations (fixes issue #29)
  - Fix parsing of input files without a newline at the end (fixes issue #28)
  - Fix conversion of material cards containing keywords (ntab, ptab, etc.)
- Lots of refactoring and cleaning

1.4.6 v0.2.0

- Support conversion of planes defined by three points (fixes issue #1).
- Add support for 0 as a cone sheet specifier (fixes issue #2).
- Add support for elliptic tori in MIP (fixes issue #3).
- Handle the specification of `FILL` transformations by ID (fixes issue #4).
- Partially handle conversion of lattices in fully-specified form (fixes issue #16).
- Fix parsing of cell card options starting with `*FILL` in MIP.
- Fix detection of line continuation.
- Preserve the precision of isotope concentrations in the MCNP file.
- Make pytest runnable from the package root directory.
- Fix conversion of materials specified by density and atomic fractions (fixes issue #8).
- Fix conversion of materials specified by total atomic concentration and atomic fractions (fixes issue #14).
- Support spaces between the # operator and its arguments (fixes issue #10).
- Handle the conversion of cards with a TRCL keyword (fixes issue #21).
- Remove duplicated surfaces (fixes issue #19).
- Do not use fixed IDs for the helper surface used in UNION cells (fixes issue #15).
- Do not emit TRIPOLI-4 cells where the same surface appears twice.
- Some linting and refactoring.

1.4.7 v0.1.4

- First public release

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z