pytrip98 Documentation

Release 1.5.1.post0.dev

Author

Jul 04, 2017

Contents

Contents:

CHAPTER 1

Getting Started with PyTRiP

Brief overview of PyTRiP98 and how to install it.

Introduction

PyTRiP98 is a python package for working with files generated by the treatment planning systems TRiP98 and VIRTUOS/VOXELPLAN. Dicom files are also to some extent supported.

PyTRiP will simplify importing and exporting files, processing the data, and also execute TRiP98 locally or remotely. Thereby it is possible to work with TRiP98 in e.g. a Windows environment, while accessing TRiP98 on a remote server. PyTRiP enables scripting for large parameters studies of treatment plans, and also more advanced and automized manipulation than what commercial treatment planning systems might allow.

Let us for instance assume, that one wants (for whatever reason) to reduce all Hounsfield units in a CT cube with a factor of two and write the result into a new file, this can be realized with a few lines of code.

```
>>> import pytrip as pt
>>> c = pt.CtxCube()
>>> c.read("tst000001.ctx")  # read a .ctx file
```

Where the first line imports the pytrip modules, the second line initialized the CtxCube object. The new object holds (among others) the read() method, which is then being used to load the CT data. Now let's work with the CT data:

```
>>> c *= 0.5  # reduce all HUs inside c with a factor of two
```

and write it to disk.

```
>>> c.write("out0000001.ctx") # write the new file.
```

And that all.

We may want to inspect the data, what is the largest and the smalles HU value found in the cube?

```
>>> print(c.cube.min())
>>> print(c.cube.max())
```

To see all available methods and attributes, one can run the

```
>>> dir(c)
```

command, or read the detailed documentation.

Quick Installation Guide

PyTRiP is available for python 2.7, 3.2 or later, and can be installed via pip. If you intend to use **pytripgui** you need the python 2.7 version.

We recommend that you run a modern Linux distribution, like: **Ubuntu 16.04** or newer, **Debian 9 Stretch** (currently known as testing) or any updated rolling release (archLinux, openSUSE tumbleweed). In that case, be sure you have **python** and **python-pip** installed. To get them on Debian or Ubuntu, type being logged in as normal user:

```
$ sudo apt-get install python-pip
```

To automatically download and install the pytrip library, type:

```
$ sudo pip install pytrip98
```

NOTE: the package is named **pytrip98**, while the name of library is **pytrip**.

This command will automatically download and install **pytrip** for all users in your system.

For more detailed instruction, see the *Detailed Installation Guide*

To learn how to install **pytripgui** graphical user interface, proceed to following document page: https://github.com/pytrip/pytripgui

Using PyTRiP

Once installed, the package can be imported at a python command line or used in your own python program with `import pytrip as pt`. See the examples directory for both kinds of uses. Also see the *User's Guide* for more details of how to use the package.

Support

Bugs can be submitted through the issue tracker. Besides the example directory, cookbook recipes are encouraged to be posted on the wiki page

Next Steps

To start learning how to use PyTRiP, see the *PyTRiP User's Guide*.

License

PyTRiP98 is licensed under GPLv3.

CHAPTER 2

Detailed Installation Guide

Installation guide is divided in two phases: checking the prerequisites and main package installation.

Prerequisites

PyTRiP works under Linux and Mac OSX operating systems.

First we need to check if Python interpreter is installed. Try if one of following commands (printing Python version) works:

```
$ python --version
$ python3 --version
```

At the time of writing Python language interpreter has two popular versions: 2.x (Python 2) and 3.x (Python 3) families. Command `python` invokes either Python 2 or 3, while `python3` can invoke only Python 3.

pytrip supports most of the modern Python versions, mainly: 2.7, 3.2, 3.3, 3.4, 3.5 and 3.6. Check if your interpreter version is supported.

If none of `python` and `python3` commands are present, then Python interpreter has to be installed.

We suggest to use the newest version available (from 3.x family).

Python installers can be found at the python web site (http://python.org/download/).

PyTRiP aslo relies on these packages:

- NumPy – Better arrays and data processing.
- matplotlib – Needed for plotting.
- paramiko – Needed for remote execution of TRiP98 via SSH.

and if they are not installed beforehand, these will automatically be fetched by pip.

Installing using pip (all platforms)

The easiest way to install PyTRiP98 is using pip:

```
.. note::
```

Pip comes pre-installed with Python newer than 3.4 and 2.?? (for 2.x family)

Administrator installation (root access)

Administrator installation is very simple, but requires to save some files in system-wide directories (i.e. */usr*):

```
$ sudo pip install pytrip98
```

To upgrade the **pytrip** to newer version, simply type:

```
$ sudo pip install --upgrade pytrip98
```

To completely remove **pytrip** from your system, use following command:

```
$ sudo pip uninstall pytrip98
```

Now all **pytrip** commands should be installed for all users:

```
$ cubeslice --help
```

User installation (non-root access)

User installation will put the **pytrip** under hidden directory *$HOME/.local*.

To install the package, type in the terminal:

```
$ pip install pytrip98 --user
```

If *pip* command is missing on your system, replace *pip* with *pip3* in abovementioned instruction.

To upgrade the **pytrip** to newer version, simply type:

```
$ pip install --upgrade pytrip98 --user
```

To completely remove **pytrip** from your system, use following command:

```
$ pip uninstall pytrip98
```

In most of modern systems all executables found in *$HOME/.local/bin* directory can be called like normal commands (i.e. *ls*, *cd*). It means that after installation you should be able to simply type in terminal:

```
$ cubeslice --help
```

If this is not the case, please prefix the command with *$HOME/.local/bin* and call it in the following way:

```
$ $HOME/.local/bin/cubeslice --help
```

CHAPTER 3

PyTRiP User's Guide

pytrip object model, description of classes, examples

Using PyTRiP as a library

The full potential of PyTRiP is exposed when using it as a library.

Using the *dir()* and *help()* methods, you may explore what functions are available, check also the index and module tables found in this documentation.

CT and Dose data are handled by the "CtxCube" and "DosCube" classes, respectively. Structures (volume of interests) are handled by the VdxCube class. For instance, when a treatment plan was made the resulting 3D dose distribution (and referred to as a "DosCube").

```
>>> import pytrip as pt
>>> dc = pt.DosCube()
>>> dc.read("foobar.dos")
```

You can display the entire doscube by simply printing its string (str or repr) value:

```
>>> dc
....
```

We recommend you to take a look at the *Examples* and browse the modindex page.

Converters

A few converters based on PyTRiP are supplied as well. These converters are:

trip2dicom.py converts a Voxelplan formatted file to a Dicom file.

dicom2trip.py converts a Dicom file to a Voxelplan formatted file.

cubeslice.py Generates .png files for each slice found in the given cube.

gd2dat.py Converts a GD formatted plot into a stripped ASCII-file

gd2agr.py Converts a GD formatted plot into a a xmgrace formatted plot.

rst2sobp.py Converts a raster scan file to a file which can be read by FLUKA or SHIELD-HIT12A.

CHAPTER 4

Examples

Code snippets demonstrating PyTRiP capabilities.

Example 00 - Cube arithmetic

This example demonstrates simple arithmetic on dose- and LET-cubes. Two dose cubes from two fields are summed to generate a new total dose cube.

The two LET-cubes from the two fields are combined to calculate the total dose-averaged LET in the resulting treatment plan. All data are saved to disk.

```
"""
Simple example of how to do arithmetic on Cube objects in PyTRiP.
"""
import pytrip as pt

# sum two dose cubes, write result:
print("Two half boxes: out.dos")
d1 = pt.DosCube()
d2 = pt.DosCube()
d1.read("box052000.dos")
d2.read("box053000.dos")
d = (d1 + d2)
d.write("out.dos")

# print minium and maximum value found in cubes
print(d1.cube.min(), d1.cube.max())
print(d2.cube.min(), d2.cube.max())

# calculate new dose average LET cube
l1 = pt.LETCube()
l2 = pt.LETCube()
l1.read("box052000.dosemlet.dos")
l2.read("box053000.dosemlet.dos")
```

```

l = ((d1 * l1) + (d2 * l2)) / (d1 + d2)
l.write("out.dosemlet.dos")
```

Example 01 - Handling structures

This example shows how one can select a region inside a CTX data cube using a VDX file, and perform some manipulation of it.

```
"""
This example shows how to use contours to select volume of interests inside a CTX cube. The VOI is then manipulated.
"""
import pytrip as pt

# first define some paths and other important parameters
ctx_path = "/home/bassler/Projects/CTdata/TST000/TST000000.ctx"
vdx_path = "/home/bassler/Projects/CTdata/TST000/TST000000.vdx"
my_target_voi = "GTV"

# load CT cube
my_ctx = pt.CtxCube()
my_ctx.read(ctx_path)

# load VOIs
my_vdx = pt.VdxCube(my_ctx)  # my_vdx is the object which will hold all volumes of interest and the meta information
my_vdx.read(vdx_path)  # load the .vdx file
print(my_vdx.get_voi_names())  # show us all VOIs found in the .vdx file

# Select the requested VOI from the VdxCube object
target_voi = my_vdx.get_voi_by_name(my_target_voi)

# get_voi_cube() returns a DosCube() object, where all voxels inside the VOI holds the value 1000, and 0 elsewhere.
voi_cube = target_voi.get_voi_cube()

# Based on the retrieved DosCube() we calculate a three dimensional mask object,
# which assigns True to all voxels inside the Voi, and False elsewhere.
mask = (voi_cube.cube == 1000)

# "The mask object and the CTX cube have same dimensions (they are infact inherited from the same top level class).
# Therefore we can apply the mask cube to the ctx cube and work with the values.
# For instance we can set all HUs to zero within the Voi:
my_ctx.cube[mask] = 0
# or add 100 to all HUs of the voxels inside the mask:
# my_ctx.cube[mask] += 100

# save masked CT to the file in current directory
masked_ctx = "masked.ctx"
my_ctx.write(masked_ctx)
```

Working with dose cubes is fully analogous to the CTX cubes.

Example 02 - TRiP execution

In this example, we demonstrate how to actually perform a treatment plan using TRiP98. Most of the lines concern with the setup of TRiP.

```
"""
This example demonstrates how to load a CT cube in Voxelplan format, and the associated contours.
Then a plan is prepared and optimized using TRiP98.
"""
import os
import logging

import pytrip as pt
import pytrip.tripexecuter as pte

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)  # give some output on what is going on.

# Fist we specify the directory where all our files are:
wdir = "/home/bassler/test"

# In TRiP, the patient "TST000" would typically carry the filename "TST000000"
patient_name = "TST000000"

# so we can construc the paths to the CTX and VDX files like this:
ctx_path = os.path.join(wdir, patient_name + ".ctx")
vdx_path = os.path.join(wdir, patient_name + ".vdx")

# Next we load the CT cube:
c = pt.CtxCube()
c.read(ctx_path)

# And load the contours
v = pt.VdxCube(c)
v.read(vdx_path)

# we may print all contours found in the Vdx file, if we want to
print(v.get_voi_names())

# Ok, we have the Contours and the CT cube ready. Next we must prepare a plan.
# We may choose any basename for the patient. All output files will be named using
# this basename.
plan = pte.Plan(basename=patient_name)

# We need to specify where the kernel files can be found. The location may depend on the ion we
# wnat to treat with. This example is for carbon ions:
plan.ddd_dir = "/home/bassler/TRiP98/base/DATA/DDD/12C/RF3MM/*"
plan.spc_dir = "/home/bassler/TRiP98/base/DATA/SPC/12C/RF3MM/*"
plan.sis_path = "/home/bassler/TRiP98/base/DATA/SIS/12C.sis"
plan.hlut_path = "/home/bassler/TRiP98/base/DATA/HLUT/19990218.hlut"
plan.dedx_path = "/home/bassler/TRiP98/base/DATA/DEDX/20040607.dedx"
plan.working_dir = "/home/bassler/test/"  # working dir must exist.

# Set the plan target to the voi called "CTV"
plan.voi_target = v.get_voi_by_name('CTV')

```

```
# some optional parameters (if not set, they will all be zero by default)
plan.rifi = 3.0  # 3 mm ripple filter. (This is only for documentaiton, it will not affect the dose optimization.)
plan.bolus = 0.0  # No bolus is applied here. Set this to some value, if you are to optimize very shallow tumours.
plan.offh2o = 1.873  # Some offset mimicing the monitoring ionization chambers and exit window of the beam nozzle.

# Next we need to specify at least one field, and add that field to the plan.
field = pte.Field()
field.basename = patient_name  # This name will be used for output filenames, if any field specific output is saved.
field.gantry = 10.0  # degrees
field.couch = 90.0  # degrees
field.fwhm = 4.0  # spot size in [mm]
field.projectile = 'C'

print(field)  # We can print all parameters of this field, for checking.
plan.fields.append(field)  # attach field to plan. You may attach multiple fields.

# Next, set the flags for what output should be generated, when the plan has completed.
plan.want_phys_dose = True  # We want a physical dose cube, "TST000000.dos"
plan.want_bio_dose = False  # No biological cube (Dose * RBE)
plan.want_dlet = True  # We want to have the dose-averaged LET cube
plan.want_rst = False  # Print the raster scan files (.rst) for all fields.

# print(plan)  # this will print all plan parameters

te = pte.Execute(c, v)  # get the executer object, based on the given Ctx and Vdx cube.

# in the case that TRiP98 is not installed locally, you may have to enable remote execution:
# te.remote = True
# te.servername = "titan.phys.au.dk"
# te.username = "bassler"
# te.password = "xxxxxxxx"  # you can set a password, but this is strongly discouraged. Better to exchange SSH keys!
# te.remote_base_dir = "/home/bassler/test"
# depending on the remote .bashrc_profile setup, it may be needed to specify the full path
# for the remote TRiP installation. On some systems the $PATH is set, so this line can be omitted,
# or shortened to just "TRiP98"
# te.trip_bin_path = "/opt/aptg/TRiP98/bin/TRiP98"

te.execute(plan)  # this will run TRiP
# te.execute(plan, False)  # set to False, if TRiP98 should not be executed. Good for testing.

# requested results can be found in
# plan.dosecubes[]
# and
# plan.letcubes[]
# and they are also saved to working_dir
```

CHAPTER 5

Credits

Development

- Niels Bassler - Stockholm University, Sweden
- Leszek Grzanka - IFJ-PAN, Poland <leszek.grzanka@gmail.com>
- Jakob Toftegaard - Aarhus University Hospital, Denmark

Contributors

None yet. Why not be the first?

How to cite PyTRiP98

If you use PyTRiP for your research, please cite:

CHAPTER 6

pytrip package

TODO: documentation here.

class pytrip.**CtxCube**(*cube=None*)
Bases: *pytrip.cube.Cube*

Class for handling CT-data. In TRiP98 these are stored in VOXELPLAN format with the .ctx suffix. This class can also handle Dicom files.

create_dicom()
Creates a Dicom object from self.

This function can be used to convert a TRiP98 CTX file to Dicom format.

Returns A Dicom object.

data_file_extension = '.ctx'

header_file_extension = '.hed'

read_dicom(*dcm*)
Imports CT-images from Dicom object.

Parameters **dcm** (*Dicom*) – a Dicom object

write(*path*)
Write CT-data to disk, in TRiP98/Voxelplan format.

This method will build and write both the .hed and .ctx file.

Parameters **path** (*str*) – path to header file, data file or basename (without extension)

write_dicom(*directory*)
Write CT-data to disk, in Dicom format.

Parameters **directory** (*str*) – directory to write to. If directory does not exist, it will be created.

class pytrip.**VdxCube**(*cube=None*)
VdxCube is the master class for dealing with Volume of Interests (VOIs). A VdxCube contains one or more VOIs which are structures which represent some organ (lung, eye ...) or target (GTV, PTV...) The Voi object

contains Slice objects which corresponds to the CT slices, and the slice objects contains contour objects. Each contour object are a set of points which delimit a closed region. One single slice object can contain multiple contours.

VdxCube —> Voi[] —> Slice[] —> Contour[] —> Point[]

Note, since TRiP98 only supports one contour per slice for each voi. PyTRiP supports functions for connecting multiple contours to a single entity using infinte thin connects.

VdxCube can import both dicom data and TRiP data, and export in the those formats.

We strongly recommend to load a CT and/or a DOS cube first, see example below:

```
>>> c = CtxCube()
>>> c.read("TST000000")
>>> v = VdxCube(c)
>>> v.read("TST000000.vdx")
```

add_voi(*voi*)
Appends a new voi to this class.

Parameters **voi** (Voi) – the voi to be appened to this class.

concat_contour()
Loop through all available VOIs and check whether any have mutiple contours in a slice. If so, merge them to a single contour.

This is needed since TRiP98 cannot handle multiple contours in the same slice.

create_dicom()
Generates and returns Dicom RTSTRUCT object, which holds all VOIs.

Returns a Dicom RTSTRUCT object holding any VOIs.

get_voi_by_name(*name*)
Returns a Voi object by its name.

Parameters **name** (*str*) – Name of voi to be returned.

Returns the Voi which has exactly this name, else raise an Error.

get_voi_names()

Returns a list of available voi names.

import_vdx(*path*)
Reads a structure file in Voxelplan format. This method is identical to self.read() and self.read_vdx()

Parameters **path** (*str*) – Full path including file extension.

number_of_vois()

Returns the number of VOIs in this object.

read(*path*)
Reads a structure file in Voxelplan format. This method is identical to self.import_vdx() and self.read_vdx()

Parameters **path** (*str*) – Full path including file extension.

read_dicom(*data*, *structure_ids=None*)
Reads structures from a Dicom RTSS Object.

Parameters

- **data** (*Dicom*) – A Dicom RTSS object.

- **structure_ids** – (TODO: undocumented)

read_vdx(*path*)
Reads a structure file in Voxelplan format.

Parameters **path** (*str*) – Full path including file extension.

write(*path*)
Writes all VOIs in voxelplan format, while ensuring no slice holds more than one contour. Identical to write_trip().

Parameters **path** (*str*) – Full path, including file extension (.vdx).

write_dicom(*directory*)
Generates a Dicom RTSTRUCT object from self, and writes it to disk.

Parameters **directory** (*str*) – Diretory where the rtss.dcm file will be saved.

write_to_voxel(*path*)
Writes all VOIs in voxelplan format.

Parameters **path** (*str*) – Full path, including file extension (.vdx).

write_trip(*path*)
Writes all VOIs in voxelplan format, while ensuring no slice holds more than one contour. Identical to write().

Parameters **path** (*str*) – Full path, including file extension (.vdx).

class pytrip.**Voi**(*name*, *cube=None*)
This is a class for handling volume of interests (VOIs). This class can e.g. be found inside the VdxCube object. VOIs may for instance be organs (lung, eye...) or targets (PTV, GTV...), or any other volume of interest.

add_slice(*slice*)
Add another slice to this VOI, and update self.slice_z table.

Parameters **slice** (Slice) – the Slice object to be appended.

calculate_bad_angles(*voi*)
(Not implemented.)

calculate_center()
Calculates the center of gravity for the VOI.

Returns A numpy array[x,y,z] with positions in [mm]

concat_contour()
Concat all contours in all slices found in this VOI.

concat_to_3d_polygon()
Concats all contours into a single contour, and writes all data points to sefl.polygon3d.

coronal = 1

create_copy(*margin=0*)
Returns an independent copy of the Voi object

Parameters **margin** – (unused)

Returns a deep copy of the Voi object

create_dicom_contour_data(*i*)
Based on self.slices, DICOM contours are generated for the DICOM ROI.

Returns Dicom ROI_CONTOURS

create_dicom_label()
Based on self.name and self.type, a Dicom ROI_LABEL is generated.

Returns a Dicom ROI_LABEL

create_dicom_structure_roi()
Based on self.name, an empty Dicom ROI is generated.

Returns a Dicom ROI.

create_point_tree()
Concats all contours. Writes a list of points into self.points describing this VOI.

define_colors()
Creates a list of default colours [R,G,B] in self.colours.

get_2d_projection_on_basis(*basis*, *offset=None*)
(TODO: Documentation)

get_2d_slice(*plane*, *depth*)
Gets a 2d Slice object from the contour in either sagittal or coronal plane. Contours will be concated.

Parameters

- **plane** (*int*) – either self.sagittal or self.coronal
- **depth** (*float*) – position of plane

Returns a Slice object.

get_3d_polygon()
Returns a list of points rendering a 3D polygon of this VOI, which is stored in sefl.polygon3d. If this attibute does not exist, create it.

get_color(*i=None*)

Parameters **i** (*int*) – selects a colour, default if None.

Returns a [R,G,B] list.

get_min_max()
Set self.temp_min and self.temp_max if they dont exist.

Returns minimum and maximum x y coordinates in Voi.

get_name()

Returns The name of this VOI.

get_roi_type_name(*type_id*)

Returns The type name of the ROI.

get_roi_type_number(*type_name*)

Returns 1 if GTV or CTV, 10 for EXTERNAL, else 0.

get_row_intersections(*pos*)
(TODO: Documentation needed)

get_slice_at_pos(*z*)
Finds and returns a slice object found at position z [mm] (float).

Parameters **z** (*float*) – slice position in absolute coordiantes (i.e. including any offsets)

Returns VOI slice at position z, z position may be approxiamte

get_voi_cube()
This method returns a DosCube object with value 1000 in each voxel within the Voi and zeros elsewhere. It can be used as a mask, for selecting certain voxels. The function may take some time to execute the first invocation, but is faster for subsequent calls, due to caching.

Returns a DosCube object which holds the value 1000 in those voxels which are inside the Voi.

number_of_slices()

Returns number of slices covered by this VOI.

read_dicom(*info*, *data*)
Reads a single ROI (= VOI) from a Dicom data set.

Parameters

- **info** – (not used)
- **data** (*Dicom*) – Dicom ROI object which contains the contours.

read_vdx(*content*, *i*)
Reads a single VOI from Voxelplan .vdx data from 'content'. Format 2.0 :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

read_vdx_old(*content*, *i*)
Reads a single VOI from Voxelplan .vdx data from 'content', assuming a legacy .vdx format. VDX format 1.2. :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

sagital = 2

sagittal = 2

set_color(*color*)

Parameters **[3*int]** – set a color [R,G,B].

to_voxel_string()
Creates the Voxelplan formatted text, which can be written into a .vdx file (format 2.0).

Returns a str holding the all lines needed for a Voxelplan formatted file.

class pytrip.**DosCube**(*cube=None*)
Bases: *pytrip.cube.Cube*

Class for handling Dose data. In TRiP98 these are stored in VOXELPLAN format with the .dos suffix. This class can also handle Dicom files.

calculate_dvh(*voi*)
Calculate DHV for given VOI. Dose is given in relative units (target dose = 1.0). In case VOI lies outside the cube, then None is returned.

Parameters **voi** – VOI for which DHV should be calculated

Returns (dvh, min_dose, max_dose, mean, area) tuple. dvh - 2D array holding DHV histogram,

min_dose and max_dose, mean_dose - obvious, mean_volume - effective volume dose.

create_dicom()
Creates a Dicom RT-Dose object from self.

This function can be used to convert a TRiP98 Dose file to Dicom format.

Returns a Dicom RT-Dose object.

create_dicom_plan()
Create a dummy Dicom RT-plan object.

The only data which is forwarded to this object, is self.patient_name. :returns: a Dicom RT-plan object.

data_file_extension = '.dos'

header_file_extension = '.hed'

read_dicom(*dcm*)
Imports the dose distribution from Dicom object.

Parameters **dcm** (*Dicom*) – a Dicom object

write(*path*)
Write Dose data to disk, in TRiP98/Voxelplan format.

This method will build and write both the .hed and .dos file.

Parameters **path** (*str*) – Path, any file extentions will be ignored.

write_dicom(*directory*)
Write Dose-data to disk, in Dicom format.

This file will save the dose cube and a plan associated with that dose. Function call create_dicom() and create_dicom_plan() and then save these.

Parameters **directory** (*str*) – Directory where 'rtdose.dcm' and 'trplan.dcm' will be stored.

write_dvh(*voi*, *filename*)
Save DHV for given VOI to the file.

class pytrip.**DensityCube**(*ctxcube*, *hlut_path='/data/hlut_den.dat'*)
Bases: *pytrip.cube.Cube*

Class for working with denisity cubes [g/cm3]

calculate_cube()
Calculate the density values from HU table and interpolating the loaded hlut_data.

import_hlut()
Imports the Hounsfield lookup table and stores it into self.hlut_data object self.hlut_data is trained linear interpolator, it can be later called to get interpolated values

class pytrip.**LETCube**(*cube=None*)
Bases: *pytrip.cube.Cube*

This class handles LETCubes.

It is similar to DosCubes and CtxCubes, but is intended to hold LET data. The object has build-in methods to read and write the LET data cubes, calculate the LET-volume historgrams, and write these to disk. It is inherited from Cube, which contains many additional methods and attributes.

calculate_lvh(*voi*)
Calculate a LET-volume historgram.

Parameters **voi** (Voi) – The volume of interest, in the form of a Voi object.

Returns A tuple containing - lvh: the LET-volume histogram - min_lvh: array of LET values below 2% - max_lvh: array of LET values above 98% - area: TODO - what is this?

data_file_extension = '.dosemlet.dos'

get_max()
Returns the largest value in the LETCube.

Returns the largets value found in the in the LETCube.

header_file_extension = '.dosemlet.hed'

write(*path*)
Write the LETCube and its header to a file with the filename 'path'.

Parameters **path** (*str*) – path to header file, data file or basename (without extension)

write_lvh_to_file(*voi*, *path*)
Write the LET-volume historgram to a file.

Parameters

- **voi** (Voi) – The volume of interest, n the form of a Voi object.
- **path** (*str*) – Full path of file to be written.

class pytrip.**Rst**
This class handles raster scan data, which are accelerator control files in GSI format. Raster scan data are stored in .rst file, and describe the amount of particles going into each spot in each energy layer. Each energy layer is called a 'submachine'.

gaussian_blur(*sigma*)
For a loaded .rst file, apply a normal distributed setup error to each energy layer of sigma magnitude. :params float sigma" 1-sigma error to be applied to all positions.

get_min_max()
Retrieve the largest and smallest x,y position found in all energy layers.

Returns A list of four values in [x_min,x_max,y_min,y_max] in [mm].

get_stepsize()
Returns the distance between each spot in the first energy plane.

Most likely the distance will be the same in all planes.

Returns Distancce between spots in [mm]. If no submachines are found, None is returned.

get_submachines()

Returns A list of submachines.

read(*path*)
Load and parse a raster scan (.rst) file.

Parameters **path** (*str*) – Full path to the file to be loaded, including file extension.

read_from_dicom(*path*)
Load a Dicom file from 'path'

Currently, this function merely stores the dicom data into self.data. No interpretation is done.

Parameters **path** (*str*) – Full path to Dicom file.

save_random_error_rst(*path*, *sigma*)

Subpackages

pytrip.res package

Submodules

pytrip.res.interpolate module

This package provides an easy way to perform interpolation of 1-D and 2-D datasets.

1-D dataset is represented by two columns of numbers: X and Y, of the same length. X column should contain increasing sequence of numbers. It is not required that X data should form regular (evenly spaced) grid. X and Y columns of length 1 also form 1-D data (single data point).

2-D dataset is represented by two columns of numbers: X and Y. and 2-dimensional matrix Z. Number of columns in Z matrix should be equal to length of X array, while number of rows to length of Y array. Lengths of X and Y data may differ. One or both of X and Y columns may contain single element - this is also valid 2-D dataset. X and Y column should contain increasing sequence of numbers. It is not required that X and Y data should form regular (evenly spaced) grid.

Two types of interpolation are supported: linear and spline (3rd degree polynomials).

Linear interpolation on 1-D dataset and on reduced 2-D datasets (where X or Y is single element column) is performed using *interp* method from *numpy* package.

Spline interpolation on all datasets is performed using *InterpolatedUnivariateSpline* (for 1-D data) and *RectBivariateSpline* (for 2-D data) from *scipy* package.

Cubic spline interpolation requires at least 4 data points. In case number of data points is lower, interpolator implemented in this package will fall back to quadratic spline (for 3 data points) or to linear interpolation (2 or less data points). This is different behaviour from what is implemented in *scipy* package where exception is thrown is such situation.

Note: scipy package is not imported her as default as its installation is problematic for some group of users (i.e. Windows users working without Anaconda python distribution). Most of the functionality of this package (i.e. 1-D and some cases of 2-D linear interpolation) will work without scipy package installed. If user without scipy package calls cubic spline interpolation method then an exception will be thrown and an error message printed, requesting user to install scipy.

class pytrip.res.interpolate.**RegularInterpolator**(*x*, *y*, *z=None*, *kind='spline'*)

Bases: object

RegularInterpolator is a helper class to easy interpolation of single- and double-variable function. To use it usually two steps are needed:

1. construction of the object, providing training data (data points) and interpolation type (linear or spline). During this step coefficient of interpolating function will be calculated and stored in the object. RegularInterpolator objects are so-called callables and can be called in same way as plain functions on interpolated values. Example:

```
>>> interp_func_1d = RegularInterpolator(x=exp_data_x, y=exp_data_y, kind=
↪'linear')
>>> interp_func_2d = RegularInterpolator(x=exp_data_x, y=exp_data_y,␣
↪z=exp_data_z, kind='spline')
```

2.Calling interpolation function to get intermediate values (single number or array) on interpolated ones.

```
>>> interpolated_y = interp_func_1d(x=intermediate_x)
>>> interpolated_z = interp_func_2d(x=intermediate_x, y=intermediate_y)
```

Interpolation in single step is also possible (although less efficient than two-step method):

```
>>> interpolated_y = RegularInterpolator.eval(x=intermediate_x, xp=exp_data_x,␣
↪yp=exp_data_y, kind='linear')
>>> interpolated_z = RegularInterpolator(x=intermediate_x, y=intermediate_y,␣
↪   xp=exp_data_x, yp=exp_data_y, zp=exp_data_z, kind='spline')
```

classmethod **eval** (*x*, *y=None*, *xp=None*, *yp=None*, *zp=None*, *kind='linear'*)

Perform interpolation in a single step. Find interpolation function, based on data points (xp, yp and zp) and then execute it on interpolated values (x and y).

Parameters **x** – array_like or single value

Input x-coordinate(s).

Parameters **y** – array_like or single value

Input y-coordinate(s) (in case of 2-D dataset).

Parameters **xp** – array_like

The 1-d array of data-points x-coordinates, must be in strictly ascending order.

Parameters **yp** – array_like

The 1-d array of data-points y-coordinates. If y is from 1-D data points (z is None) then it should be of the same length (shape) as x. Otherwise (2-D data points, z is not None) then it must be in strictly ascending order.

Parameters **zp** – array_like

None in case of 1-D dataset. Otherwise the 2-d array of data-points z-coordinates, of shape (x.size, y.size).

Parameters **kind** – interpolation algorithm: 'linear' or 'spline' (default).

Returns array_like or single value

Interpolated value

pytrip.res.point module

TODO: documentation here.

pytrip.res.point.**angles_from_trip** (*gantry*, *couch*)

pytrip.res.point.**angles_to_trip** (*gantry*, *couch*)

pytrip.res.point.**array_to_point_array** (*points*, *offset*)

pytrip.res.point.**get_area_contour** (*polygon*)

pytrip.res.point.**get_basis_from_angles** (*gantry*, *couch*)

pytrip.res.point.**get_nearest_point** (*point*, *contour*)

pytrip.res.point.**get_x_intersection** (*y*, *polygon*)

pytrip.res.point.**max_list** (*a*, *b*)

pytrip.res.point.**min_list** (*a*, *b*)

pytrip.res.point.**point_in_polygon** (*x*, *y*, *polygon*)

pytrip.res.point.**short_distance_polygon_idx** (*poly1*, *poly2*)

pytrip.res.point.**vector_to_angles** (*vec*)

pytrip.res.utils module

TODO: documentation here.

pytrip.res.utils.**get_max**(*a*, *b*)

pytrip.res.utils.**get_min**(*a*, *b*)

pytrip.tripexecuter package

The tripexecuter module provides functions for executing TRiP98 locally or remotely.

class pytrip.tripexecuter.**Field**(*basename=''*)
Bases: object

One or more Field() object, which then can be added to a Plan() object. :params str basename: basename of field without file extension (input or output will be suffixed with proper file extension)

set_isocenter_from_string(*isocenter_str*)
Override the automatically determined isocenter from TRiP98. :params str isocenter_str: x,y,z coordinates of the isocenter/target in [mm] in a comma delimted string. such as "123.33,158.4,143.5". If and empty string is provided, then the isocenter is unset, and TRiP98 will calculate it itself.

Following the target() option in the TRiP98 field command, one can specify the isocenter of the target. If not used, TRiP98 will determine the isocenter from the target Voi provided.

class pytrip.tripexecuter.**Execute**(*ctx*, *vdx*, *ctx_path=''*, *vdx_path=''*)
Bases: object

Execute class for running trip using attached Ctx, Vdx and Plan objects.

add_log_listener(*listener*)
A listener is something which has a .write(txt) method.

execute(*plan*, *run=True*, *_callback=None*)
Executes the Plan() object using TRiP98.

log(*txt*)
Writes txt to all listeners, stripping any newlines.

test_local_trip()
Test if TRiP98 can be reached locally. :returns tripname, tripver: Name of TRiP98 installation and its version. :returns: None, None if not installed

test_remote_trip()
Test if TRiP98 can be reached remotely. :returns tripname, tripver: Name of TRiP98 installation and its version. :returns: None, None if not installed

class pytrip.tripexecuter.**Plan**(*basename=''*, *comment=''*)
Bases: object

Class of handling plans.

clean_voi_caches()
TODO: document me

destroy()
Destructor for Vois and Fields in this class.

load_dose(*path*, *_type*, *target_dose=0.0*)
Load and append a new DOS cube from path to self.doscubes.

load_let(*path*)
Load and append a new LET cube from path to self.letcubes.

make_exec(*no_output=False*)
Generates the .exec script from the plan and stores it into self._trip_exec.

make_sis(*projectile*, *focus=()*, *intensity=()*, *position=()*, *path=''*, *write=False*)
Creates a SIS table based on path, focus, intensity and position. example: plan.make_sis("1H", "4", "1E6,1E7,1E8", (2, 20, 0.1)) results in: makesis 1H / focus(4) intensity(1E6,1E7,1E8) position(2 TO 20 BY 0.1)

:params str projectile" such as "12C" or "1H" :params str focus: comma-delimited list of FWHM focus positions in [mm], such as "2,4,6,8" :params str intensities: tuple/list of intensities, such as "1E6,1E7,1E8" :params float position: tuple describing min/max/step range, i.e. (5,40,0.1) for 5 to 40 cm in 1 mm steps. :params path: write path :params write: Set to True, if the generated sis file should be written to path when executed.

save_data(*images*, *path*)
Save this plan, including associated data. TODO: may have to be implemented in a better way.

save_exec(*exec_path=None*)
Generates (overwriting) self._trip_exec, and saves it to exec_path.

save_plan(*images*, *path*)
Saves the complete plan to path. 1) *.exec 2) *.dos

Submodules

pytrip.tripexecuter.execute module

This file contains the Execute() class which can execute a Plan() object locally or on a remote server with a complete TRiP98 installations.

class pytrip.tripexecuter.execute.**Execute**(*ctx*, *vdx*, *ctx_path=''*, *vdx_path=''*)
Bases: object

Execute class for running trip using attached Ctx, Vdx and Plan objects.

add_log_listener(*listener*)
A listener is something which has a .write(txt) method.

execute(*plan*, *run=True*, *_callback=None*)
Executes the Plan() object using TRiP98.

log(*txt*)
Writes txt to all listeners, stripping any newlines.

test_local_trip()
Test if TRiP98 can be reached locally. :returns tripname, tripver: Name of TRiP98 installation and its version. :returns: None, None if not installed

test_remote_trip()
Test if TRiP98 can be reached remotely. :returns tripname, tripver: Name of TRiP98 installation and its version. :returns: None, None if not installed

pytrip.tripexecuter.field module

Field() objects here, which can be passed to a Plan() object.

class pytrip.tripexecuter.field.**Field**(*basename=''*)
Bases: object

One or more Field() object, which then can be added to a Plan() object. :params str basename: basename of field without file extension (input or output will be suffixed with proper file extension)

set_isocenter_from_string(*isocenter_str*)
Override the automatically determined isocenter from TRiP98. :params str isocenter_str: x,y,z coordinates of the isocenter/target in [mm] in a comma delimted string. such as "123.33,158.4,143.5". If and empty string is provided, then the isocenter is unset, and TRiP98 will calculate it itself.

Following the target() option in the TRiP98 field command, one can specify the isocenter of the target. If not used, TRiP98 will determine the isocenter from the target Voi provided.

pytrip.tripexecuter.plan module

Structure: A Plan() may hold one or multiple Field() objects. When a Plan() is proper setup, it can be executed with methods in Execute().

class pytrip.tripexecuter.plan.**Plan**(*basename=''*, *comment=''*)
Bases: object

Class of handling plans.

clean_voi_caches()
TODO: document me

destroy()
Destructor for Vois and Fields in this class.

load_dose(*path*, *_type*, *target_dose=0.0*)
Load and append a new DOS cube from path to self.doscubes.

load_let(*path*)
Load and append a new LET cube from path to self.letcubes.

make_exec(*no_output=False*)
Generates the .exec script from the plan and stores it into self._trip_exec.

make_sis(*projectile*, *focus=()*, *intensity=()*, *position=()*, *path=''*, *write=False*)
Creates a SIS table based on path, focus, intensity and position. example: plan.make_sis("1H", "4", "1E6,1E7,1E8", (2, 20, 0.1)) results in: makesis 1H / focus(4) intensity(1E6,1E7,1E8) position(2 TO 20 BY 0.1)

:params str projectile" such as "12C" or "1H" :params str focus: comma-delimited list of FWHM focus positions in [mm], such as "2,4,6,8" :params str intensities: tuple/list of intensities, such as "1E6,1E7,1E8" :params float position: tuple describing min/max/step range, i.e. (5,40,0.1) for 5 to 40 cm in 1 mm steps. :params path: write path :params write: Set to True, if the generated sis file should be written to path when executed.

save_data(*images*, *path*)
Save this plan, including associated data. TODO: may have to be implemented in a better way.

save_exec(*exec_path=None*)
Generates (overwriting) self._trip_exec, and saves it to exec_path.

save_plan(*images*, *path*)
Saves the complete plan to path. 1) *.exec 2) *.dos

pytrip.utils package

Submodules

pytrip.utils.bevlet2oer module

Convert .bevlet (Beams Eye View LET) to OER (Oxygen Enhancement Ratio) values.

class pytrip.utils.bevlet2oer.**ReadGd**(*gd_filename*, *_dataset=0*, *dat_filename=None*)
Bases: object

Reads a bevlet formatted file. TODO: must be renamed

pytrip.utils.bevlet2oer.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

pytrip.utils.cubeslice module

Generates .png files for each slice found in the given cube.

pytrip.utils.cubeslice.**load_ct_cube**(*filename*)
loads the CT cube

Params str filename path to filename which must be loaded

Returns a CtxCube object and a str containing the path to the basename.

pytrip.utils.cubeslice.**load_data_cube**(*filename*)
Loads either a Dos or LET-cube.

Params str filname path to Dos or LET-cube.

Returns a DosCube or LETCube object and a str containing the path to the basename.

pytrip.utils.cubeslice.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
The main function for cubeslice.py

pytrip.utils.dicom2trip module

Script for converting a DICOM file to TRiP98 / Voxelplan style files.

pytrip.utils.dicom2trip.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Main function for dicom2trip.py

pytrip.utils.dvhplot module

Script for generating DVH or other volume histograms.

pytrip.utils.dvhplot.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
Main function for dvhplot.py

pytrip.utils.gd2agr module

Converts GD files to xmgrace files in .agr format.

pytrip.utils.gd2agr.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Main function for gd2agr.

pytrip.utils.gd2dat module

Reads .gd files and can convert them into (xmgrace readable) ascii data.

Can be used in the command line

Mandatory arguments: Name Type Default Description filename STRING "" File name of .gd file, should be first.

Optional arguments: Name Type Default Description exp STRING "exp" if "exp": Export data of .gd into a .dat file agr STRING "" if "agr": Export data of .gd into a .agr file LET STRING "LET" if "LET": Export also data entitled "DoseLET"

Example: 1) reading the file foo.gd and exporting it as ascii file foo.dat python gd2dat.py foo.gd

2) reading the file foo.gd and exporting it as xmgrace file foo.agr python gd2dat.py foo.gd agr

class pytrip.utils.gd2dat.**ReadGd**(*gd_filename*, *exp=False*, *agr=False*, *let=False*)

Bases: object

This class reads .gd files.

export(*out_file_name=None*)

Export data to out_file_name :params str out_file_name: full path to output file including file extension

read()

Reads the filename set when constructing the ReadGd class

pytrip.utils.gd2dat.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

pytrip.utils.rst2sobp module

This script converts a raster scan file in GSI format to a sobp.dat file which can be used by FLUKA or SHIELD-HIT12A to simulate the beam using Monte Carlo methods.

pytrip.utils.rst2sobp.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Main function of the rst2sobp script.

pytrip.utils.rst_plot module

This script plots the raster scan file for verication of the spot delivery of the ion accelerator.

pytrip.utils.rst_plot.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

pytrip.utils.trip2dicom module

Script for converting Voxelplan / TRiP98 Ctx and Vdx data to a DICOM file.

pytrip.utils.trip2dicom.**main**(*args=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)

Main function for trip2dicom.py

Submodules

pytrip.ctx module

The CTX module contains the CtxCube class which is inherited from the Cube class. It is used for handling CT-data, both Voxelplan and Dicom.

class pytrip.ctx.**CtxCube**(*cube=None*)

Bases: *pytrip.cube.Cube*

Class for handling CT-data. In TRiP98 these are stored in VOXELPLAN format with the .ctx suffix. This class can also handle Dicom files.

create_dicom()

Creates a Dicom object from self.

This function can be used to convert a TRiP98 CTX file to Dicom format.

Returns A Dicom object.

data_file_extension = '.ctx'

header_file_extension = '.hed'

read_dicom(*dcm*)

Imports CT-images from Dicom object.

Parameters **dcm** (*Dicom*) – a Dicom object

write(*path*)

Write CT-data to disk, in TRiP98/Voxelplan format.

This method will build and write both the .hed and .ctx file.

Parameters **path** (*str*) – path to header file, data file or basename (without extension)

write_dicom(*directory*)

Write CT-data to disk, in Dicom format.

Parameters **directory** (*str*) – directory to write to. If directory does not exist, it will be created.

pytrip.cube module

This module provides the Cube class, which is used by the CTX, DOS, LET and VDX modules. A cube is a 3D object holding data, such as CT Hounsfield units, Dose- or LET values.

class pytrip.cube.**Cube**(*cube=None*)

Bases: object

Top level class for 3-dimensional data cubes used by e.g. DosCube, CtxCube and LETCube. Otherwise, this cube class may be used for storing different kinds of data, such as number of cells, oxygenation level, surviving fraction, etc.

static check_compatibility(*a*, *b*)
Simple comparison of cubes. if X,Y,Z dims are the same, and voxel sizes as well, then they are compatible. (Duck typed)

See also the function is_compatible().

Params Cube a the first cube to be compared with the second (b).

Params Cube b the second cube to be compared with the first (a).

create_cube_from_equation(*equation*, *center*, *limits*, *radial=True*)
Create Cube from a given equation.

This function is currently out of order.

create_dicom_base()

create_empty_cube(*value*, *dimx*, *dimy*, *dimz*, *pixel_size*, *slice_distance*, *slice_offset=0.0*)
Creates an empty Cube object.

Values are stored as 2-byte integers.

Parameters

- **value** (*int16*) – integer value which will be assigned to all voxels.
- **dimx** (*int*) – number of voxels along x
- **dimy** (*int*) – number of voxels along y
- **dimz** (*int*) – number of voxels along z
- **pixel_size** (*float*) – size of each pixel (x == y) in [mm]
- **slice_distance** (*float*) – the distance between two slices (z) in [mm]
- **slice_offset** (*float*) – start position of the first slice in [mm] (default 0.0 mm)

static discover_file(*file_name*)
Checks if file_name exists in the filesystem. If yes, gives back its path. If not, checks if gzipped file with same name exists. If gzipped file exists, gives back it path, otherwise returns None.

Parameters **file_name** – File name or path

Returns file_name, file_name + ".gz" or None

indices_to_pos(*indices*)
Translate index number of a voxel to real position in [mm], including any offsets.

The z position is always following the slice positions.

Params [int] indices tuple or list of integer indices (i,j,k) or [i,j,k]

Returns list of positions, including offsets, as a list of floats [x,y,z]

is_compatible(*other*)
Check if this Cube object is compatible in size and dimensions with 'other' cube.

A cube object can be a CtxCube, DosCube, LETCube or similar object. Unlike check_compatibility(), this function compares itself to the other cube.

Parameters **other** (Cube) – The other Cube object which will be checked compability with.

Returns True if compatibe.

mask_by_voi (*voi*, *value*)
Overwrites the Cube voxels within the given Voi with 'value'.

Voxels within the structure are filled it with 'value'. Voxels outside the contour are not touched.

Parameters

- **voi** (Voi) – the volume of interest
- **value=0** – value to be assigned to the voxels within the contour.

mask_by_voi_add (*voi*, *value=0*)
Add 'value' to all voxels within the given Voi

'value' is added to each voxel value within the given volume of interest. Voxels outside the volume of interest are not touched.

Parameters

- **voi** (Voi) – the volume of interest
- **value=0** – value to be added to the voxel values within the contour.

mask_by_voi_all (*voi*, *preset=0*, *data_type=<type 'numpy.int16'>*)
Attaches/overwrites Cube.data based on a given Voi.

Voxels within the structure are filled it with 'preset' value. Voxels outside the contour will be filled with Zeros.

Parameters

- **voi** (Voi) – the volume of interest
- **preset** (*int*) – value to be assigned to the voxels within the contour.
- **data_type** – numpy data type, default is np.int16

merge (*cube*)

merge_zero (*cube*)

classmethod parse_path (*path_name*)
Parse path_name which can have form of: bare name (i.e. TST001), plain file (TST001.hed or TST001.ctx), gzipped file (TST001.hed.gz or TST001.ctx.gz) or some other name. Calculates plain file names of header and data file. Extension of data file is extracted from the class from which this method was called (.ctx for CtxCube, .dos for DosCube, .dosemlet.dos for LetCube). In case of non-parseable data None is returned.

```
>>> from pytrip import CtxCube
>>> CtxCube.parse_path("frodo.hed")
('frodo.hed', 'frodo.ctx')
>>> CtxCube.parse_path("baggins.ctx")
('baggins.hed', 'baggins.ctx')
>>> CtxCube.parse_path("mordor")
('mordor.hed', 'mordor.ctx')
>>> CtxCube.parse_path("legolas.hed.gz")
('legolas.hed', 'legolas.ctx')
>>> CtxCube.parse_path("gimli.son.of.gloin.ctx.gz")
('gimli.son.of.gloin.hed', 'gimli.son.of.gloin.ctx')
>>> CtxCube.parse_path("bilbo.dos")
(None, None)
>>> pt.CtxCube.parse_path("/home/pytrip/patient.ctx")
('/home/pytrip/patient.hed', '/home/pytrip/patient.ctx')
>>> from pytrip import DosCube
```

```
>>> DosCube.parse_path("bilbo.dos")
('bilbo.hed', 'bilbo.dos')
>>> DosCube.parse_path("baggins.ctx")
(None, None)
>>> from pytrip import LETCube
>>> LETCube.parse_path("aragorn.dosemlet.dos")
('aragorn.dosemlet.hed', 'aragorn.dosemlet.dos')
>>> LETCube.parse_path("aragorn.dosemlet")
(None, None)
>>> LETCube.parse_path("aragorn")
('aragorn.dosemlet.hed', 'aragorn.dosemlet.dos')
>>> LETCube.parse_path("aragorn.ctx")
(None, None)
```

Parameters **path_name** – path to header file, data file or basename path (path to file without extension).

Path can be absolute or relative. It can also lead to gzipped files. :return: pair of filenames for header and data

read(*path*)
Reads both TRiP98 data and its associated header into the Cube object.

Parameters **path** (*str*) – Path to filename to be read, file extention may be given but is not neccesary.

set_byteorder(*endian=None*)
Set/change the byte order of the data to be written to disk.

Available options are: - 'little' vms, Intel style little-endian byte order. - 'big' aix, Motorola style big-endian byte order. - if unspecified, the native system dependent endianess is used.

Parameters **endian** (*str*) – optional string containing the endianess.

set_data_type(*type*)
Sets the data type for the TRiP98 header files.

Parameters **type** (*numpy.type*) – numpy type, e.g. np.uint16

slice_to_z(*slice_number*)
Return z-position in [mm] of slice number (starting at 1).

Params int slice_number slice number, starting at 1 and no bound check done here.

Returns position of slice in [mm] including offset

pytrip.ddd module

This module provides the DDD class for handling depth-dose curve kernels for TRiP98.

class pytrip.ddd.**DDD**
Class for handling Depth-Dose Data.

get_ddd_by_energy(*energy*, *points*)
TODO: documentation

get_ddd_data(*energy*, *points*)
TODO: documentation

get_ddd_grid(*energy_list*, *n*)
TODO: documentation

get_dist(*energy*)
TODO: documentation

load_ddd(*directory*)
Loads all .ddd files found in 'directory'

Params str directory directory where the .ddd files are found.

pytrip.dicomhelper module

Auxilliary functions for handling Dicom data.

pytrip.dicomhelper.**compare_dicom_key**(*dcm*)
Specifying the sorting key for CT images.

pytrip.dicomhelper.**read_dicom_dir**(*dicom_dir*)
Reads a directory with dicom files. Identifies each dicom file with .dcm suffix and returns a dict containing a dicom object. Dicom object may be "CT", "RTSTRUCT", "RTDOSE" or "RTPLAN". Corresponding keys for lookup are "images", "rtss", "rtdose" or "rtplan" repsectively. CT objects are lists of images. They will be sorted by the position in patient given by the ImagePositionPatient[2] tag.

Returns A dict containing dicom objects and corresponding keys 'images','rtss','rtdose' or 'rtplan'.

pytrip.dos module

This module provides the DosCube class, which the user should use for handling plan-generated dose distributions.

class pytrip.dos.**DosCube**(*cube=None*)
Bases: *pytrip.cube.Cube*

Class for handling Dose data. In TRiP98 these are stored in VOXELPLAN format with the .dos suffix. This class can also handle Dicom files.

calculate_dvh(*voi*)
Calculate DHV for given VOI. Dose is given in relative units (target dose = 1.0). In case VOI lies outside the cube, then None is returned.

Parameters **voi** – VOI for which DHV should be calculated

Returns (dvh, min_dose, max_dose, mean, area) tuple. dvh - 2D array holding DHV histogram,

min_dose and max_dose, mean_dose - obvious, mean_volume - effective volume dose.

create_dicom()
Creates a Dicom RT-Dose object from self.

This function can be used to convert a TRiP98 Dose file to Dicom format.

Returns a Dicom RT-Dose object.

create_dicom_plan()
Create a dummy Dicom RT-plan object.

The only data which is forwarded to this object, is self.patient_name. :returns: a Dicom RT-plan object.

data_file_extension = '.dos'

header_file_extension = '.hed'

read_dicom(*dcm*)
Imports the dose distribution from Dicom object.

Parameters **dcm** (*Dicom*) – a Dicom object

write(*path*)
Write Dose data to disk, in TRiP98/Voxelplan format.

This method will build and write both the .hed and .dos file.

Parameters **path** (*str*) – Path, any file extentions will be ignored.

write_dicom(*directory*)
Write Dose-data to disk, in Dicom format.

This file will save the dose cube and a plan associated with that dose. Function call create_dicom() and create_dicom_plan() and then save these.

Parameters **directory** (*str*) – Directory where 'rtdose.dcm' and 'trplan.dcm' will be stored.

write_dvh(*voi*, *filename*)
Save DHV for given VOI to the file.

pytrip.error module

Internal Classes for error message handling.

exception pytrip.error.**InputError**(*msg*)
Bases: exceptions.Exception

exception pytrip.error.**ModuleNotLoadedError**(*msg*)
Bases: exceptions.Exception

pytrip.field module

TODO: documentation here.

class pytrip.field.**Field**(*ddd*)

get_cube_basis()

get_ddd_list()

get_energy_list()

get_max_dist()

get_merged_raster_points()

load_from_raster_points(*rst*)

class pytrip.field.**SubField**(*submachine*, *ddd*, *rst*, *resolution=0.5*)

get_lateral(*resolution=0.5*)

get_max_dist()

get_merge_raster_points(*size*)

get_raster_matrixs(*size*)

get_size()

get_subfield_cube(*resolution=0.5*)

pytrip.field.**compare_raster_point**(*a*, *b*)

pytrip.file_parser module

This module contains a parser method for parsing voxelplan files.

pytrip.file_parser.**parse_to_var**(*data*, *var*, *stoptag=''*)
Parses a variable 'var' from 'data'.

Returns (out,i) tuple

out - the rest of the line with 'var' and a " " removed. If nothing is followed, then True is returned. i - number of lines parsed.

pytrip.let module

This module provides the LETCube for handling LET data.

class pytrip.let.**LETCube**(*cube=None*)
Bases: *pytrip.cube.Cube*

This class handles LETCubes.

It is similar to DosCubes and CtxCubes, but is intended to hold LET data. The object has build-in methods to read and write the LET data cubes, calculate the LET-volume historgrams, and write these to disk. It is inherited from Cube, which contains many additional methods and attributes.

calculate_lvh(*voi*)
Calculate a LET-volume historgram.

Parameters **voi** (Voi) – The volume of interest, in the form of a Voi object.

Returns A tuple containing - lvh: the LET-volume histogram - min_lvh: array of LET values below 2% - max_lvh: array of LET values above 98% - area: TODO - what is this?

data_file_extension = '.dosemlet.dos'

get_max()
Returns the largest value in the LETCube.

Returns the largets value found in the in the LETCube.

header_file_extension = '.dosemlet.hed'

write(*path*)
Write the LETCube and its header to a file with the filename 'path'.

Parameters **path** (*str*) – path to header file, data file or basename (without extension)

write_lvh_to_file(*voi*, *path*)
Write the LET-volume historgram to a file.

Parameters

- **voi** (Voi) – The volume of interest, n the form of a Voi object.
- **path** (*str*) – Full path of file to be written.

pytrip.paths module

This module provides the DensityCube class and some special class to find robust angles for treament based on a quality index factor defined in http://dx.doi.org/10.3109/0284186X.2015.1067720.

class pytrip.paths.**DensityCube**(*ctxcube*, *hlut_path='/data/hlut_den.dat'*)
Bases: *pytrip.cube.Cube*

Class for working with denisity cubes [g/cm3]

calculate_cube()
Calculate the density values from HU table and interpolating the loaded hlut_data.

import_hlut()
Imports the Hounsfield lookup table and stores it into self.hlut_data object self.hlut_data is trained linear interpolator, it can be later called to get interpolated values

class pytrip.paths.**DensityProjections**(*cube*)
Functions here were mosly used buy for the publication http://dx.doi.org/10.3109/0284186X.2015.1067720

calculate_angle_quality(*voi*, *gantry*, *couch*, *calculate_from=0*, *stepsize=1.0*, *avoid=[]*, *voi_cube=None*, *gradient=True*)
Calculates a quality index for a given gantry/couch combination.

calculate_angle_quality_thread(*voi*, *gantry*, *couch*, *calculate_from=0*, *stepsize=1.0*, *q=None*, *avoid=[]*, *voi_cube=None*, *gradient=True*)
TODO: Documentation

calculate_back_start_voi(*voi*, *start*, *beam_axis*)
TODO: Documentation

Params voi

Params start

Params beam axis

Returns

calculate_best_angles(*voi*, *fields*, *min_dist=20*, *gantry_limits=[-90, 90]*, *couch_limits=[0, 359]*, *avoid=[]*)
TODO: Documentation

calculate_front_start_voi(*voi*, *start*, *beam_axis*)
TODO: Documentation

Params voi

Params start

Params beam axis

Returns

calculate_projection(*voi*, *gantry*, *couch*, *calculate_from=0*, *stepsize=1.0*)
TODO: documentation

Parameters

- **voi** (*Voi*) – tumortarget, type is Voi
- **gantry** (*float*) – angle in degrees
- **couch** (*float*) – angle in degrees

- **calculate_from** (*int*) – 0 is mass center 1 is the most distant point in the tumor from the beamaxis
- **stepsize** (*float*) – relative to pixelsize, 1 is a step of 1 pixel

Returns

calculate_quality_grid(*voi*, *gantry*, *couch*, *calculate_from=0*, *stepsize=1.0*, *avoid=[]*, *gradient=True*)
TODO: Documentation

calculate_quality_list(*voi*, *gantry*, *couch*, *calculate_from=0*, *stepsize=1.0*, *avoid=[]*, *gradient=True*)
TODO: Documentation

optimize_angle(*voi*, *couch*, *gantry*, *margin*, *iteration*, *avoid=[]*)
TODO: Documentation

pytrip.paths.**cmp_sort**(*a*, *b*)
Sorting key. If gantry angle is equal, then sort by couch angle.

pytrip.raster module

This module provides the top-level Rst class which can handle accelerator control files generated by TRiP98. It also provides the SubMachine class which treats individual energy layers.

class pytrip.raster.**Rst**
This class handles raster scan data, which are accelerator control files in GSI format. Raster scan data are stored in .rst file, and describe the amount of particles going into each spot in each energy layer. Each energy layer is called a 'submachine'.

gaussian_blur(*sigma*)
For a loaded .rst file, apply a normal distributed setup error to each energy layer of sigma magnitude. :params float sigma" 1-sigma error to be applied to all positions.

get_min_max()
Retrieve the largest and smallest x,y position found in all energy layers.

Returns A list of four values in [x_min,x_max,y_min,y_max] in [mm].

get_stepsize()
Returns the distance between each spot in the first energy plane.

Most likely the distance will be the same in all planes.

Returns Distancce between spots in [mm]. If no submachines are found, None is returned.

get_submachines()

Returns A list of submachines.

read(*path*)
Load and parse a raster scan (.rst) file.

Parameters **path** (*str*) – Full path to the file to be loaded, including file extension.

read_from_dicom(*path*)
Load a Dicom file from 'path'

Currently, this function merely stores the dicom data into self.data. No interpretation is done.

Parameters **path** (*str*) – Full path to Dicom file.

save_random_error_rst(*path*, *sigma*)

class pytrip.raster.**SubMachine**

get_raster_grid()

raster_min_max()
Returns the smallest and largest x and y positions for this energy layer. :returns: a list of four elements [min_x, max_x, min_y, max_y]

save_random_error_machine(*fp*, *sigma*)
Generates and stores a single energy layer where Gaussian blur has been applied.

Parameters

- **fp** – file pointer
- **sigma** (*float*) – sigma of the Gaussian blur to be applied [mm]

pytrip.util module

Module with auxilliary functions (mostly internal use).

pytrip.util.**evaluator**(*funct*, *name='funct'*)
Wrapper for evaluating a function.

Params str funct string which will be parsed

Params str name name which will be assigned to created function.

Returns function f build from 'funct' input.

pytrip.util.**get_class_name**(*item*)

Returns name of class of 'item' object.

pytrip.util.**volume_histogram**(*cube*, *voi=None*, *bins=256*)
Generic volume histogram calculator, useful for DVH and LVH or similar.

Params cube a data cube of any shape, e.g. Dos.cube

Params voi optional voi where histogramming will happen.

Returns [x],[y] coordinates ready for plotting. Dose (or LET) along x, Normalized volume along y in %.

If VOI is not given, it will calculate the histogram for the entire dose cube.

Providing voi will slow down this function a lot, so if in a loop, it is recommended to do masking i.e. only provide Dos.cube[mask] instead.

pytrip.vdx module

This module holds all the user needs to deal with Volume of interests. It provides the top-level VdxCube class, Voi, Slice and Contour classes. The Voi class represents a volume of interest 'VOI', also called region of interest 'ROI' in Dicom lingo. Each Voi holds several Slice, which are noramlly synced with an associated CT-cube. Each Slice holds one or more Contours.

class pytrip.vdx.**Contour**(*contour*, *cube=None*)
Class for handling single Contours.

add_child(*contour*)
(TODO: Document me)

calculate_center()
Calculate the center for a single contour, and the area of a contour in 3 dimensions.

Returns Center of the contour [x,y,z] in [mm], area [mm**2] (TODO: to be confirmed)

concat()
In case of multiple contours in the same slice, this method will concat them to a single conour. This is important for TRiP98 compability, as TRiP98 cannot handle multiple contours in the same slice of of the same VOI.

contains_contour(*contour*)

Returns True if contour in argument is contained inside self.

get_min_max()

Returns The lowest x,y,z values and the highest x,y,z values found in this Contour object.

has_childs()

Returns True or False, whether this Contour object has children.

number_of_points()

Returns Number of points in this Contour object.

print_child(*level*)
Print child to stdout.

Parameters **level** (*int*) – (TODO: needs documentation)

push(*contour*)
Push a contour on the contour stack.

Parameters **contour** (Contour) – a Contour object.

read_vdx(*content*, *i*)

Reads a single Contour from Voxelplan .vdx data from 'content'. VDX format 2.0.

Note:

•in VDX files the last contour point is repeated if the contour is closed.

•If we have a point of interest, the length is 1.

•Length 2 and 3 should thus never occur in VDX files (assuming all contours are closed)

params [str] content list of lines with the .vdx content

params int i line number to the list.

returns current line number, after parsing the VOI.

---

read_vdx_old(*slice_number*, *xy_line*)
Reads a single Contour from Voxelplan .vdx data from 'content' and appends it to self.contour data VDX format 1.2.

See also notes in read_vdx(), regarding the length of a contour.

Params slice_number list of numbers (as characters) with slice number

Params xy_line list of numbers (as characters) representing X and Y coordinates of a contour

remove_inner_contours()
(TODO: needs documentation)

to_voxel_string()
Creates the Voxelplan formatted text, which can be written into a .vdx file.

Returns a str holding the contour points needed for a Voxelplan formatted file.

class pytrip.vdx.**Slice**(*cube=None*)
The Slice class is specific for structures, and should not be confused with Slices extracted from CTX or DOS objects.

add_contour(*contour*)
Adds a new 'contour' to the existing contours.

Parameters **contour** (Contour) – the contour to be added.

add_dicom_contour(*dcm*)
Adds a Dicom CONTOUR to the existing list of contours in this Slice class.

Parameters **dcm** (*Dicom*) – a Dicom CONTOUR object.

calculate_center()
Calculate the center position of all contours in this slice.

Returns a list of center positions [x,y,z] in [mm] for each contour found.

concat_contour()
Concat all contours in this Slice object to a single contour.

create_dicom_contours(*dcmcube*)
Creates and returns a list of Dicom CONTOUR objects from self.

get_intersections(*pos*)
(TODO: needs documentation)

get_min_max()
Set self.temp_min and self.temp_max if they dont exist.

Returns minimum and maximum x y coordinates in Voi.

get_position()

Returns the position of this slice in [mm] including zoffset

number_of_contours()

Returns number of contours found in this Slice object.

read_vdx(*content*, *i*)
Reads a single Slice from Voxelplan .vdx data from 'content'. VDX format 2.0. :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

read_vdx_old(*content*, *i*)
Reads a single Slice from Voxelplan .vdx data from 'content'. VDX format 1.2. :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

remove_inner_contours()
Removes any "holes" in the contours of this slice, thereby changing the topology of the contour.

to_voxel_string()
Creates the Voxelplan formatted text, which can be written into a .vdx file (format 2.0)

Returns a str holding the slice information with the countour lines for a Voxelplan formatted file.

class pytrip.vdx.**VdxCube**(*cube=None*)

VdxCube is the master class for dealing with Volume of Interests (VOIs). A VdxCube contains one or more VOIs which are structures which represent some organ (lung, eye ...) or target (GTV, PTV...) The Voi object contains Slice objects which corresponds to the CT slices, and the slice objects contains contour objects. Each contour object are a set of points which delimit a closed region. One single slice object can contain multiple contours.

VdxCube —> Voi[] —> Slice[] —> Contour[] —> Point[]

Note, since TRiP98 only supports one contour per slice for each voi. PyTRiP supports functions for connecting multiple contours to a single entity using infinte thin connects.

VdxCube can import both dicom data and TRiP data, and export in the those formats.

We strongly recommend to load a CT and/or a DOS cube first, see example below:

```
>>> c = CtxCube()
>>> c.read("TST000000")
>>> v = VdxCube(c)
>>> v.read("TST000000.vdx")
```

add_voi(*voi*)

Appends a new voi to this class.

Parameters **voi** (Voi) – the voi to be appened to this class.

concat_contour()

Loop through all available VOIs and check whether any have mutiple contours in a slice. If so, merge them to a single contour.

This is needed since TRiP98 cannot handle multiple contours in the same slice.

create_dicom()

Generates and returns Dicom RTSTRUCT object, which holds all VOIs.

Returns a Dicom RTSTRUCT object holding any VOIs.

get_voi_by_name(*name*)

Returns a Voi object by its name.

Parameters **name** (*str*) – Name of voi to be returned.

Returns the Voi which has exactly this name, else raise an Error.

get_voi_names()

Returns a list of available voi names.

import_vdx(*path*)

Reads a structure file in Voxelplan format. This method is identical to self.read() and self.read_vdx()

Parameters **path** (*str*) – Full path including file extension.

number_of_vois()

Returns the number of VOIs in this object.

read(*path*)

Reads a structure file in Voxelplan format. This method is identical to self.import_vdx() and self.read_vdx()

Parameters **path** (*str*) – Full path including file extension.

read_dicom(*data*, *structure_ids=None*)
Reads structures from a Dicom RTSS Object.

Parameters

- **data** (*Dicom*) – A Dicom RTSS object.
- **structure_ids** – (TODO: undocumented)

read_vdx(*path*)
Reads a structure file in Voxelplan format.

Parameters **path** (*str*) – Full path including file extension.

write(*path*)
Writes all VOIs in voxelplan format, while ensuring no slice holds more than one contour. Identical to write_trip().

Parameters **path** (*str*) – Full path, including file extension (.vdx).

write_dicom(*directory*)
Generates a Dicom RTSTRUCT object from self, and writes it to disk.

Parameters **directory** (*str*) – Diretory where the rtss.dcm file will be saved.

write_to_voxel(*path*)
Writes all VOIs in voxelplan format.

Parameters **path** (*str*) – Full path, including file extension (.vdx).

write_trip(*path*)
Writes all VOIs in voxelplan format, while ensuring no slice holds more than one contour. Identical to write().

Parameters **path** (*str*) – Full path, including file extension (.vdx).

class pytrip.vdx.**Voi**(*name*, *cube=None*)
This is a class for handling volume of interests (VOIs). This class can e.g. be found inside the VdxCube object. VOIs may for instance be organs (lung, eye...) or targets (PTV, GTV...), or any other volume of interest.

add_slice(*slice*)
Add another slice to this VOI, and update self.slice_z table.

Parameters **slice** (Slice) – the Slice object to be appended.

calculate_bad_angles(*voi*)
(Not implemented.)

calculate_center()
Calculates the center of gravity for the VOI.

Returns A numpy array[x,y,z] with positions in [mm]

concat_contour()
Concat all contours in all slices found in this VOI.

concat_to_3d_polygon()
Concats all contours into a single contour, and writes all data points to sefl.polygon3d.

coronal = 1
id for coronal view

create_copy(*margin=0*)
Returns an independent copy of the Voi object

Parameters **margin** – (unused)

Returns a deep copy of the Voi object

create_dicom_contour_data(*i*)
Based on self.slices, DICOM contours are generated for the DICOM ROI.

Returns Dicom ROI_CONTOURS

create_dicom_label()
Based on self.name and self.type, a Dicom ROI_LABEL is generated.

Returns a Dicom ROI_LABEL

create_dicom_structure_roi()
Based on self.name, an empty Dicom ROI is generated.

Returns a Dicom ROI.

create_point_tree()
Concats all contours. Writes a list of points into self.points describing this VOI.

define_colors()
Creates a list of default colours [R,G,B] in self.colours.

get_2d_projection_on_basis(*basis*, *offset=None*)
(TODO: Documentation)

get_2d_slice(*plane*, *depth*)
Gets a 2d Slice object from the contour in either sagittal or coronal plane. Contours will be concated.

Parameters

- **plane** (*int*) – either self.sagittal or self.coronal
- **depth** (*float*) – position of plane

Returns a Slice object.

get_3d_polygon()
Returns a list of points rendering a 3D polygon of this VOI, which is stored in sefl.polygon3d. If this attibute does not exist, create it.

get_color(*i=None*)

Parameters **i** (*int*) – selects a colour, default if None.

Returns a [R,G,B] list.

get_min_max()
Set self.temp_min and self.temp_max if they dont exist.

Returns minimum and maximum x y coordinates in Voi.

get_name()

Returns The name of this VOI.

get_roi_type_name(*type_id*)

Returns The type name of the ROI.

get_roi_type_number(*type_name*)

Returns 1 if GTV or CTV, 10 for EXTERNAL, else 0.

get_row_intersections(*pos*)
(TODO: Documentation needed)

get_slice_at_pos(*z*)
Finds and returns a slice object found at position z [mm] (float).

Parameters **z** (*float*) – slice position in absolute coordiantes (i.e. including any offsets)

Returns VOI slice at position z, z position may be approxiamte

get_voi_cube()
This method returns a DosCube object with value 1000 in each voxel within the Voi and zeros elsewhere. It can be used as a mask, for selecting certain voxels. The function may take some time to execute the first invocation, but is faster for subsequent calls, due to caching.

Returns a DosCube object which holds the value 1000 in those voxels which are inside the Voi.

number_of_slices()

Returns number of slices covered by this VOI.

read_dicom(*info*, *data*)
Reads a single ROI (= VOI) from a Dicom data set.

Parameters

- **info** – (not used)
- **data** (*Dicom*) – Dicom ROI object which contains the contours.

read_vdx(*content*, *i*)
Reads a single VOI from Voxelplan .vdx data from 'content'. Format 2.0 :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

read_vdx_old(*content*, *i*)
Reads a single VOI from Voxelplan .vdx data from 'content', assuming a legacy .vdx format. VDX format 1.2. :params [str] content: list of lines with the .vdx content :params int i: line number to the list. :returns: current line number, after parsing the VOI.

sagital = 2
deprecated, backwards compability to pytripgui, do not use.

sagittal = 2
id for sagittal view

set_color(*color*)

Parameters **[3*int]** – set a color [R,G,B].

to_voxel_string()
Creates the Voxelplan formatted text, which can be written into a .vdx file (format 2.0).

Returns a str holding the all lines needed for a Voxelplan formatted file.

pytrip.vdx.**create_cube**(*cube*, *name*, *center*, *width*, *height*, *depth*)
Creates a new VOI which holds the contours rendering a square box

Parameters

- **cube** (Cube) – A CTX or DOS cube to work on.
- **name** (*str*) – Name of the VOI
- **center** (*[float*3]*) – Center position [x,y,z] in [mm]
- **width** (*float*) – Width of box, along x in [mm]
- **height** (*float*) – Height of box, along y in [mm]

- **depth** (*float*) – Depth of box, along z in [mm]

Returns A new Voi object.

pytrip.vdx.**create_cylinder**(*cube*, *name*, *center*, *radius*, *depth*)
Creates a new VOI which holds the contours rendering a cylinder along z

Parameters

- **cube** (Cube) – A CTX or DOS cube to work on.
- **name** (*str*) – Name of the VOI
- **center** (*[float*3]*) – Center position of cylinder [x,y,z] in [mm]
- **radius** (*float*) – Radius of cylinder in [mm]
- **depth** (*float*) – Depth of cylinder, along z in [mm]

Returns A new Voi object.

pytrip.vdx.**create_sphere**(*cube*, *name*, *center*, *radius*)
Creates a new VOI which holds the contours rendering a sphere along z

Parameters

- **cube** (Cube) – A CTX or DOS cube to work on.
- **name** (*str*) – Name of the VOI
- **center** (*[float*3]*) – Center position of sphere [x,y,z] in [mm]
- **radius** (*float*) – Radius of sphere in [mm]

Returns A new Voi object.

pytrip.vdx.**create_voi_from_cube**(*cube*, *name*, *value=100*)
Creates a new VOI which holds the contours following an isodose lines.

Parameters

- **cube** (Cube) – A CTX or DOS cube to work on.
- **name** (*str*) – Name of the VOI
- **value** (*int*) – The isodose value from which the countour will be generated from.

Returns A new Voi object.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W