PolyTex

Release 0.4.1

Bin Yang

Mar 07, 2024

CONTENTS

CHAPTER

ONE

POLYTEX: A PARAMETRIC TEXTILE GEOMETRY MODELING PACKAGE

1.1 About this project

PolyTex is an open-source toolkit for geometry modeling of woven textiles based on volumetric images. It provides functionality such as geometrical feature extraction, local variability analysis and textile geometry modeling. A meshing module was implemented to generate voxel meshes. Generation of tetrahedral conformal meshes will be implemented in future release. Local material properties are assigned to each cell in the generated mesh, such that the anisotropic and heterogeneity are reflected. This image-based model is commonly referred to as a "Digital Material Twin". The toolkit is designed to provide material scientists with accurate numerical models to predict composite behaviors while not requiring extensive experience in image processing and mesh generation. Hence, Application programming interface (API) for OpenFOAM and Abaqus is provided.

We release this toolbox as an open-source project aiming to facilitate the application of numerical simulations based on digital material twins to engineering problems. In this regard, the project is well documented (https://polytex.readthedocs.io/) and we would appreciate any contributions from the community (e.g. comments, suggestions, and corrections aimed at improving the software and documentation).

Our issue tracker is at https://github.com/binyang424/PolyTex/issues. Please report any bugs that you find or fork the repository on GitHub and create a pull request. We welcome all changes, big or small, and we will help you make the pull request if you are new to git.

> PolyTex is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version.
>
> PolyTex is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See LICENSE for more details.

1.2 Installation

To install PolyTex using PyPI, run the following command:

```
$ pip install polytex
```

To install PolyTex from the source, begin by cloning the repository using git:

```
$ git clone https://github.com/binyang424/PolyTex.git
```

If you are unfamiliar with Git, you can refer to our tutorial Git for beginners. Alternatively, you can download a specific branch of the source code from GitHub as a .zip file at https://github.com/binyang424/PolyTex.

Once the repository is cloned, navigate to the root directory of the PolyTex repository where the setup.py file is located, and execute the following command:

```
$ python setup.py install
```

To install PolyTex using the wheel file, navigate to the subdirectory ./dist/ of the downloaded PolyTex repository, and run:

```
$ pip install polytex-<version>.whl
```

1.3 Contributing to PolyTex

Thank you for considering contributing to PolyTex! This project thrives on community contributions, and we appreciate your help.

1.3.1 How Can You Contribute?

- **Reporting Bugs:** If you find a bug, please open an issue at https://github.com/binyang424/PolyTex/issues. Provide as much detail as possible, including your environment, steps to reproduce, and the expected vs. actual behavior.
- **Suggesting Enhancements:** Have an idea for a new feature or an improvement? Feel free to open an issue and discuss it with the community.
- **Code Contributions:** If you want to contribute code to PolyTex, fork the repository, create a new branch for your changes, and submit a pull request. Follow the coding standards and make sure your changes are well-tested.

1.3.2 Getting Started

1. Fork the PolyTex repository.
2. Clone your forked repository to your local machine:

   ```
   $ git clone https://github.com/your-username/PolyTex.git
   ```

3. Create a new branch for your changes:

   ```
   $ git checkout -b feature/your-feature
   ```

4. Make your changes and commit them:

   ```
   $ git add .
   $ git commit -m "Add your commit message here"
   ```

5. Push your changes to your fork:

   ```
   $ git push origin feature/your-feature
   ```

6. Open a pull request on the PolyTex repository.

1.3.3 Code Style

Follow the established code style in the project. Make sure your code is well-documented and includes tests when applicable. See style guide for docstrings used with the `numpydoc` extension for Sphinx.

1.3.4 License

By contributing to `PolyTex`, you agree that your contributions will be licensed under the project's LICENSE.

1.4 Citation

To cite `PolyTex` in publications use:

- Bin YANG, Yuwei Feng, Cédric BÉGUIN, Philippe CAUSSE and Jihui WANG. Open Source Tool for Micro-CT Aided Meso-scale Modeling and Meshing of Complex Textile Composite Structures. Submitted to *Composites Science and Technology* (2024).

CHAPTER

TWO

DOCUMENTATION AND USAGE

2.1 Classes

2.1.1 Textile objects

class polytex.textile.**Textile**(*name='textile'*)

Bases: object

Initialize the textile object.

Attributes

bounds
Bounding box of the textile.

items
Return tow names as a list.

n_tows
Number of tows in the textile.

Methods

add_group([name, tow])	Groups of the tows in the textile.
add_tow(tow[, group])	Add a tow to the textile.
case_prepare([path])	Prepare a case for OpenFOAM simulation.
cell_labeling([surface_mesh, intersection, ...])	Label the cells of the background mesh with tow id.
decimate()	Decimate the mesh.
export_as_inp([fp, scale, orientation])	Export the textile mesh as inp file for Abaqus simulation.
export_as_openfoam(fp[, scale, ...])	Export the textile mesh as polyMesh folder for OpenFOAM simulation.
export_as_vtu(fp[, binary])	Export the textile mesh as a vtu file.
from_file(path)	Load a textile object from a file.
meshing(bbox[, voxel_size, show, labeling, ...])	Generate a mesh for the textile.
reconstruct()	Reconstruct the textile object from the saved file.
remove(tow)	Remove a tow from the textile.
save([path, filename, data_size])	Save the textile object to a file.
triangulate()	Hexahedral mesh to tetrahedral mesh.

`add_group`(*name='group1'*, *tow=None*)

Groups of the tows in the textile. Each is a group of Tow objects. if tow is None, then the group is empty.

Parameters

name
[str] Name of the group.

tow
[Tow object] Tow to be added to the group. If tow is None, then the group is empty. If tow is not None, then tow is added to the group. Besides, if tow is not in the self.__tows__ yet, it will be added to self.__tows__.

Returns

None.

`add_tow`(*tow*, *group=None*)

Add a tow to the textile. If tow is already in the textile, then raise a ValueError.

Parameters

tow
[Tow object] Tow to be added to the textile. Stored in self.__tows__ as a dictionary. Tow.name is the key, and tow is the value.

group
[str] Group name of the tow. If group is None, then the tow is not added to any group. Stored in self.groups.

Returns

None.

`case_prepare`(*path=None*)

Prepare a case for OpenFOAM simulation.

Parameters

path
[str] Path of the case to be prepared. The default is None. If *path* is None, it is set to the root directory of the OpenFOAM mesh generated by *Textile.export_as_openfoam()*.

Returns

None.

noindex

`cell_labeling`(*surface_mesh=None*, *intersection=False*, *check_surface=False*, *yarn_permeability='Gebart'*, *threshold=1*, *verbose=False*)

Label the cells of the background mesh with tow id.

Parameters

surface_mesh
[pyvista mesh object (PolyData)] Surface mesh of fiber tows. If *None*, then the surface meshes is selected user interactively. Otherwise, the surface mesh is loaded from the path specified by *surface_mesh*.

intersection
[bool, optional] Whether to detect the intersection of the tows. The default is False.

check_surface
[bool, optional] Whether to check if the surface mesh is watertight. The default is False.

yarn_permeability
[str, optional] The permeability model used to calculate the permeability tensor of the fiber tow. The default is "Gebart". The available permeability models are "Gebart", "CaiBerdichevsky", and "DrummondTahir".

threshold
[float, optional] The tolerance for the fiber tow section detection. The default is 1. A wavy fiber tow may have several intersections with the plane of a cross-section. The threshold is used to determine if the cells is correctly labelled.

verbose
[bool, optional] Whether to print the information. The default is False.

Returns

None.

Notes

Please make sure that the name of the tow (*tow.name*) is in the format of *towType_towNumber*.

`decimate()`

Decimate the mesh.

`export_as_inp`(*fp='./mesh-C3D8R.inp'*, *scale=1*, *orientation=True*)

Export the textile mesh as inp file for Abaqus simulation.

Parameters

fp
[str] The file path and filename of the output mesh. The default is "./mesh-C3D8R.inp".

scale
[float, optional] The scale factor of the mesh. To convert the mesh from mm to m, the scale factor should be 0.001. The default is 1.

orientation
[bool, optional] Whether to export the orientation of the yarns. The default is True.

Returns

None.

`export_as_openfoam`(*fp*, *scale=1*, *boundary_type=None*, *cell_data=['yarnIndex', 'D']*)

Export the textile mesh as polyMesh folder for OpenFOAM simulation.

The structure of the output case folder is:

```
fp/ textile.name/
├── 0
│   ├── D
│   ├── yarnIndex
│   └── ...
└── constant
    ├── polyMesh
    │   ├── boundary
```

(continues on next page)

(continued from previous page)

```
│    ├── faces
│    ├── neighbour
│    ├── owner
│    ├── points
│    ├── ...
└── ...
```

Parameters

fp
[str] The file path of the output mesh.

scale
[float, optional] The scale factor of the mesh. To convert the mesh from mm to m, the scale factor should be 0.001. The default is 1.

boundary_type
[dict, optional] The boundary type of the mesh. The default is None. If None, the boundary type will be set as "wall" for all boundaries.

cell_data
[list, optional] The cell data to be written into the mesh. The default is ["yarnIndex", "D"].

Returns

None.

`export_as_vtu`(*fp*, *binary=True*)

Export the textile mesh as a vtu file.

Parameters

fp
[str] The file path of the output mesh.

binary
[bool, optional] Whether to save the mesh in binary format. The default is True.

Returns

None.

`classmethod from_file`(*path*)

Load a textile object from a file.

Parameters

path
[str] Path of the file to be loaded.

Returns

Textile object.

`meshing`(*bbox*, *voxel_size=None*, *show=False*, *labeling=False*, *yarn_permeability='Gebart'*, *surface_mesh=None*, *verbose=False*)

Generate a mesh for the textile.

Parameters

bbox
[numpy.ndarray] bounding box of the background mesh specified through a numpy array contains the minimum and maximum coordinates of the bounding box [xmin, xmax, ymin, ymax, zmin, zmax]

voxel_size
[float, or numpy.ndarray] voxel size of the background mesh. if *None*, the voxel size is set to the 1/20 of the diagonal length of the bounding box; if *float*, the voxel size is set to the float value in x, y, z directions; if *list*, *set*, or *tuple* of size 3, the voxel size is set to the values for the x-, y- and z- directions.

show
[bool, optional] Whether to show the mesh. The default is False.

labeling
[bool, optional] Whether to label the background mesh cells with tow id. The default is True.

yarn_permeability
[str, optional] The permeability model used to calculate the permeability tensor of the fiber tow. The default is "Gebart". The available permeability models are "Gebart", "CaiBerdichevsky", and "DrummondTahir".

surface_mesh
[str, optional] Path of the surface meshes of fiber tows. The default is None. If *labeling=True*, then the surface meshes are loaded from the path specified by *surface_mesh*. The surface meshes are used to label the background mesh cells with tow id. If surface_mesh is *None*, then the surface meshes are selected by the user interactively. TODO : Use the surface mesh generated by the tow object.

Returns

None.

`reconstruct()`

Reconstruct the textile object from the saved file. The saved file must be in the format of ".tex" and loaded by the *pk_load* function.

Parameters

None.

Returns

None.

noindex

`remove(`*tow*`)`

Remove a tow from the textile.

Parameters

tow
[str or Tow object] The tow to be removed.

Returns

None.

`save`(*path=None*, *filename=None*, *data_size='minimal'*)

Save the textile object to a file.

Parameters

path
[str, optional] Path of the file to be saved. The default is None. If *path* is None, then the user is asked to select the directory to save the file.

filename
[str, optional] Filename of the file to be saved. The default is None. If *filename* is None, then the filename is set to the textile name with extension ".tex".

data_size
[str, optional] Size of the data to be saved. The default is "minimal". If *data_size* is "minimal", then only the minimal information of the textile that can be used to reconstruct the textile object is saved. If *data_size* is "full", then all information of the textile is saved.

Returns

None.

Notes

TODO
[more storage can be saved. The Textile and Tow classes should be designed carefully] at next version.

`triangulate`()

Hexahedral mesh to tetrahedral mesh. Conformal meshing.

`property bounds`

Bounding box of the textile.

`property items`

Return tow names as a list. The tow names are reordered by the tow number if the tow name is in the format of "towType_towNumber". Otherwise, the tow names are ordered according to the order of adding the tows to the textile.

`property n_tows`

Number of tows in the textile.

`class polytex.tow.Tow`(*surf_points*, *order*, *rho_fiber*, *radius_fiber*, *length_scale*, *tex*, *name='Tow'*, *packing_fiber='Hex'*, *sort=True*, *resolution=None*, ***kwargs*)

Bases: `object`

Parameters

surf_points
[str, ndarray or DataFrame] The surface points of the tow should be an array of shape (n, 3) where n is the number of points. The points are extracted from volumetric images slice by slice. Please always put the column that indicates the slice number as the last column. It serves as the label for differentiate the points from different slices.

If surf_points is a string, it should be the path to the file that stores the surface points. The file should be a .pcd file as defined in the PolyTex library.

If surf_points is a numpy array, it should be an array of shape (n, 3) where n is the number of points.

If surf_points is a pandas DataFrame, it should be a DataFrame with 3 columns and n rows where n is the number of points.

order
[str] It is preferred to set the last column of the surf_points as the coordinate in the direction that is perpendicular to the image slices for geometry analysis, parametrization and kriging resampling. Hence, you may have reordered the columns. Here, you can specify the order of the columns in the reordered points. Default is "xyz". The other options are "xzy", "yxz", "yzx", "zxy", "zyx". This function will recover the original order of the columns when generating the surface mesh.

rho_fiber
[float, optional] The density of the fiber in kg/m^3.

radius_fiber
[float, optional] The radius of the fiber in m.

length_scale
[str, optional] The length scale of the coordinates. Default is "mm". The other options are "m", "cm", "um".

tex
[float, optional] The linear density of the tow in tex.

name
[str, optional] The name or type of the tow. Default is "Tow".

packing_fiber
[str, optional] The packing pattern of the fiber. Default is "Hex". The other option is "Square".

sort
[bool, optional] Whether to sort the points according to the slice number. Default is True.

resolution
[float, optional] The resolution of the MicroCT image used to generate the tow dataset. Default is None. This is only stored as an attribute for future use. It is not used in the current version.

kwargs
[dict] The keyword arguments for the PolyTex Tow class. The user can specify any keyword arguments for the Tow class. The keyword arguments will be stored as attributes of the Tow class. The user can access the attributes by tow.attribute_name.

Attributes

`attribute_names`
Get the attribute names of the Tow class.

Methods

axial_lines([save_path, plot])	Generate the axial line of the fiber tow surface.
from_file(path)	Initialize the tow from a saved tow file.
kde([bw, save_path])	Generate the kernel density estimation of the radial normalized distance for point cloud decomposition.
normal_cross_section([algorithm, save_path, ...])	Generate the normal cross section of the fiber tow surface.
radial_lines([save_path, plot, type])	Generate the radial line of the fiber tow surface.
resampling([krig_config, skip, ...])	Kriging the cross-sections of the tow to obtain the parametric equations for each cross-section (which is a closed curve).
save(save_path)	Save the fiber tow data.
smooth_window(size, h_res[, extend])	The window size of smoothing operation.
smoothing([name_drift, name_cov, ...])	Smooth the tow using parametric surface kriging.
surf_mesh([plot, save_path, end_closed])	Generate the surface mesh of the tow.
trajectory([krig_config, smooth, plot, ...])	Generate the trajectory of the tow and smooth it using kriging.
unit_vector(vector[, ax])	Returns the unit vector of the input vector along the given axis.

mw_kde	

axial_lines(*save_path=None*, *plot=True*)

Generate the axial line of the fiber tow surface.

Parameters

save_path
[str, optional] The path to save the axial line data as a vtk mesh file.

plot
[bool, optional] Whether to plot the axial line. Default is True.

Returns

axial_line
[vtkPolyData] The axial lines of the fiber tow.

classmethod from_file(*path*)

Initialize the tow from a saved tow file.

Parameters

path
[str] The path to the saved tow file. The file format is .tow.

Returns

tow
[Tow] The Tow object.

kde(*bw=None*, *save_path=None*)

Generate the kernel density estimation of the radial normalized distance for point cloud decomposition.

Parameters

bw
[float, optional] The bandwidth of the kernel density estimation. Default is None and the bandwidth will be estimated using the Scott's rule of thumb (one-fifth of the estimated value).

save_path
[str, optional] The path to save the kernel density estimation. Default is None. If None, the kernel density estimation will not be saved. The file format is .stat and can be loaded by the function *polytex.pk_load*.

Returns

clusters
[dict]

`normal_cross_section`(*algorithm='kriging'*, *save_path=None*, *plot=True*, *i_size=0.7*, *j_size=1*, *skip=10*, *max_dist=2*)

Generate the normal cross section of the fiber tow surface.

Parameters

algorithm
[str, optional] The algorithm to generate the cross section. Default is "kriging". The other option is "pyvista" which uses pyvista's mesh clip function (Polydata.clip_surface()).

save_path
[str, optional] The path to save the normal cross sections as a vtk mesh file.

plot
[bool, optional] Whether to plot the normal cross sections. Default is True.

i_size
[float, optional] The size of the i direction of the noraml plane. Default is 0.7.

j_size
[float, optional] The size of the j direction of the noraml plane. Default is 1.

skip
[int, optional] The number of cross sections to skip in plot. Default is 10. If skip is 1, all cross sections will be plotted.

Returns

cross_section
[pyvista.PolyData] The normal cross section of the fiber tows stored in a pyvista.PolyData object. Note that only the cross sections that are plotted are stored. If one wants to save all the cross sections, set skip=1.

planes
[pyvista.PolyData] The corresponding planes that the cross sections are generated from. Note that only the planes that are plotted are stored. If one wants to save all the planes, set skip=1.

`radial_lines`(*save_path=None*, *plot=True*, *type='resampled'*)

Generate the radial line of the fiber tow surface.

Parameters

save_path
[str, optional] The path to save the radial line data as a vtk mesh file.

plot
[bool, optional] Whether to plot the radial line. Default is True.

type
[str, optional] The type of radial line. Default is "resampled". The other option is "original".

Returns

radial_line
[vtkPolyData] The radial lines of the fiber tow.

resampling(*krig_config=('lin', 'cub')*, *skip=5*, *sample_position=[]*, *smooth=0*, *type='distance'*)

Kriging the cross-sections of the tow to obtain the parametric equations for each cross-section (which is a closed curve).

Parameters

krig_config
[tuple, optional] The kriging configuration. The first element is the kriging model for the drift and the second is the name of covariance. Default is ("lin", "cub").

skip
[int, optional] The number of cross-sections to be skipped for resampling to accelerate the operation. Default is 5. Namely, 1 of every 5 cross-sections will be resampled. If skip is 1, all the cross-sections will be kriged.

sample_position
[array_like, optional] The resampling positions of each cross-sections specified by the normalized distance in radial direction of each cross-section. Default is [].

smooth
[float, optional] The smoothing parameter for the kriging resampling. Default is 0. Also known as the nugget effect in geo-statistics and kriging theory.

type
[str, optional] The type of the kriging resampling. Default is "distance". The other option is "angular". If "distance", the resampling positions are specified by the normalized distance (between 0 and 1) in radial direction of each cross-section. If "angular", the resampling positions are specified by angular position (between 0 and 360) of control points in radial direction of each cross-section.

Returns

_Tow__kriged_vertices
[ndarray] The kriged points for each cross-section of the tow. It is an array of shape (n, 3) where n is the number of points. The kriged points is obtained according to the kriging configuration and the resampling positions. If the resampling positions are not specified, the kriged points are obtained by evenly sampling the cross-sections with a sampling interval of 0.05, namely, 20 points per cross- section.

expr
[dict]

The kriging expression for each cross-section. It contains two sub-dictionaries that
use the cross-section number as the key and the kriging expression as the value for the first two components of user input.

save(*save_path*)

Save the fiber tow data.

Parameters

save_path
[str] The path to save the fiber tow data.

`smooth_window`(*size*, *h_res*, *extend=2*)

The window size of smoothing operation. The window size is the number of slices in the axial direction of fiber tow that are smoothed per iteration.

A smaller window size will significantly decrease the computation time of smoothing operation.

Parameters

size
[int] The window size.

h_res
[int] The number of slices in the axial direction of fiber tow.

extend
[int, optional] The number of slices that are extended to the left and right of the smoothing window to ensure the smoothness of the surface at the boundary of the windows.

Returns

wins_interp
[ndarray] The window size for interpolation (with extensions at the two ends).

wins_result
[ndarray] The index of effective smoothing results (without extensions).

`smoothing`(*name_drift=['lin', 'lin']*, *name_cov=['cub', 'cub']*, *smooth_factor=[0, 0]*, *size=25*, *save_path=None*, *plot=False*)

Smooth the tow using parametric surface kriging. Anisotropic smoothing is applied to the tow by using different smoothing factors for the two directions.

Parameters

name_drift
[list, optional] The drift function for the parametric surface kriging. Default is [‘lin’, ‘lin’].

name_cov
[list, optional] The covariance function for the parametric surface kriging. Default is [‘cub’, ‘cub’].

smooth_factor
[list, optional] The smoothing factor for the parametric surface kriging. Default is [0, 0]. Also known as the nugget effect in geo-statistics and kriging theory. The smoothing factor is applied to the two directions separately. The first element is the smoothing factor for the radial direction and the second element is the smoothing factor for the axial direction.

size
[int, optional] # TODO : improve the description The size of the window for the moving average filter. Default is 25.

save_path
[str, optional] The path to save the smoothed tow. Default is None. If None, the smoothed tow will not be saved. The file format is .ply.

plot
[bool, optional] Whether to plot the smoothed tow. Default is False.

Returns

vertices
[ndarray] The smoothed tow vertices.

`surf_mesh`(*plot=False*, *save_path=None*, *end_closed=False*)

Generate the surface mesh of the tow.

Parameters

plot
[bool, optional] Whether to plot the surface mesh. Default is False.

save_path
[str, optional] The path to save the surface mesh. Default is None and the surface mesh will not be saved. The file format can be .ply or .vtk.

Returns

——-

surf_mesh
The surface mesh of the tow.

`trajectory`(*krig_config=('lin', 'cub')*, *smooth=0.0*, *plot=False*, *save_path=None*, *orientation=False*)

Generate the trajectory of the tow and smooth it using kriging.

Parameters

krig_config
[tuple, optional] The kriging configuration for the trajectory. It is a tuple of two strings that specify the drift and covariance function. Default is (“lin”, “cub”).

smooth
[float, optional] The smoothing parameter for the parametric curve kriging. Default is 0. Also known as the nugget effect in geo-statistics and kriging theory.

plot
[bool, optional] Whether to plot the trajectory. Default is False.

save_path
[str, optional] The path to save the trajectory as vtk file. Default is None and the trajectory will not be saved.

orientation
[bool, optional] Whether to calculate the orientation of the tow. Default is False. The orientation is the tangent vector of the trajectory.

Returns

traj
[np.ndarray] The trajectory of the tow in the form of (n, 3) where n is the number of points.

`unit_vector`(*vector*, *ax=1*)

Returns the unit vector of the input vector along the given axis.

Parameters

vector
[ndarray] The input vector.

ax
[int, optional] The axis along which the unit vector is calculated. Default is 1 (column).

Returns

unit_vector
[ndarray] The unit vector of the input vector along the given axis.

property attribute_names

Get the attribute names of the Tow class.

Returns

attribute_names
[list] The attribute names of the Tow class.

2.1.2 Primitive geometries

class polytex.geometry.basic.Point(*orig_3d=(0, 0, 0)*)

Bases: `ndarray`

Default constructor. If no arguments are given, the point is initialized to (0, 0, 0).

Parameters

cls
[class] The class of the object.

orig_3d
[tuple, list, or array_like] Defaults to 3d origin (0, 0, 0).

Returns

obj
[Point] The origin point of 3d space.

Examples

```
>>> p1 = Point()
>>> p1
Point([0, 0, 0])
```

Attributes

T
View of the transposed array.

base
Base object if memory is from some other object.

bounds
Returns ——- bounds : array_like The bounding box of the point.

ctypes
An object to simplify the interaction of the array with the ctypes module.

data
Python buffer object pointing to the start of the array's data.

dtype
Data-type of the array's elements.

flags
Information about the memory layout of the array.

flat
A 1-D iterator over the array.

imag
The imaginary part of the array.

itemsize
Length of one array element in bytes.

nbytes
Total bytes consumed by the elements of the array.

ndim
Number of array dimensions.

real
The real part of the array.

shape
Tuple of array dimensions.

size
Number of elements in the array.

strides
Tuple of bytes to step in each dimension when traversing an array.

x

xyz

y

z
Return —— z : float 3rd dimension element.

Methods

all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out, keepdims])	Return indices of the maximum values along the given axis.
argmin([axis, out, keepdims])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.

continues on next page

Table 1 – continued from previous page

cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
direction_ratio(other)	Gives the direction ratio between 2 points.
dist(other)	Both points must have the same dimensions :return: Euclidean distance
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save_as_vtk(filename[, color])	Save the point as a vtk file.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.

continues on next page

Table 1 – continued from previous page

`sum`([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
`swapaxes`(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
`take`(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
`tobytes`([order])	Construct Python bytes containing the raw data bytes in the array.
`tofile`(fid[, sep, format])	Write array to a file as text or binary (default).
`tolist`()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
`tostring`([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
`trace`([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
`transpose`(*axes)	Returns a view of the array with axes transposed.
`var`([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
`view`([dtype][, type])	New view of array with the same data.

dot	
set_x	
set_y	
set_z	

direction_ratio(*other*)

Gives the direction ratio between 2 points.

Parameters

other
[Point object] The other point to which the direction ratio is calculated.

Returns

direction_ratio
[list] The direction ratio between the 2 points.

Examples

```
>>> from polytex.geometry import Point
>>> p1 = Point(1, 2, 3)
>>> p1.direction_ratio(Point(2, 3, 5))
[1, 1, 2]
```

dist(*other*)

Both points must have the same dimensions :return: Euclidean distance

save_as_vtk(*filename*, *color=None*)

Save the point as a vtk file.

property bounds

Returns

bounds
[array_like] The bounding box of the point. The first row is the minimum values and the second row is the maximum values for each dimension.

property size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

property z

class polytex.geometry.basic.Vector(*orig_3d=(0, 0, 0)*)

Bases: *Point*

Default constructor. If no arguments are given, the point is initialized to (0, 0, 0).

Parameters

cls
[class] The class of the object.

orig_3d
[tuple, list, or array_like] Defaults to 3d origin (0, 0, 0).

Returns

obj
[Point] The origin point of 3d space.

Examples

```
>>> p1 = Point()
>>> p1
Point([0, 0, 0])
```

Attributes

T
View of the transposed array.

add

base

Base object if memory is from some other object.

bounds

Returns ——- bounds : array_like The bounding box of the point.

ctypes

An object to simplify the interaction of the array with the ctypes module.

data

Python buffer object pointing to the start of the array's data.

dtype

Data-type of the array's elements.

flags

Information about the memory layout of the array.

flat

A 1-D iterator over the array.

imag

The imaginary part of the array.

itemsize

Length of one array element in bytes.

nbytes

Total bytes consumed by the elements of the array.

ndim

Number of array dimensions.

norm

Return —— norm : float The norm of the vector.

real

The real part of the array.

shape

Tuple of array dimensions.

size

Number of elements in the array.

strides

Tuple of bytes to step in each dimension when traversing an array.

sub

x

xyz

y

z

Return —— z : float 3rd dimension element.

Methods

all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
angle_between(other[, radian])	Return the angle between 2 vectors.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out, keepdims])	Return indices of the maximum values along the given axis.
argmin([axis, out, keepdims])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
direction_ratio(other)	Gives the direction ratio between 2 points.
dist(other)	Both points must have the same dimensions :return: Euclidean distance
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.

continues on next page

Table 2 – continued from previous page

prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save_as_vtk(filename[, color])	Save the point as a vtk file.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.

cross	
dot	
set_x	
set_y	
set_z	

angle_between(*other*, *radian=False*)

Return the angle between 2 vectors.

$$\theta = \arccos \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|}$$

Parameters

other
[Vector object] The other vector to which the angle is calculated.

radian
[bool, optional] If True, return the angle in radians. The default is False.

Returns

angle
[float] The angle between the 2 vectors. If radian is True, the angle is in radians. Otherwise, the angle is in degrees.

`property norm`

`class polytex.geometry.basic.Line(`*p1*, *p2=None*, ***kwargs*`)`

Bases: `Line`

Parameters

p1
[Point object] The first point of the line.

p2
[Point object] The second point of the line.

Attributes

`ambient_dimension`
A property method that returns the dimension of LinearEntity object.

`args`
Returns a tuple of arguments of 'self'.

`assumptions0`
Return object *type* assumptions.

`boundary`
The boundary or frontier of a set.

`bounds`
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

`canonical_variables`
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

`closure`
Property method which returns the closure of a set.

`direction`
The direction vector of the LinearEntity.

expr_free_symbols

`free_symbols`
Return from the atoms of self those which are free symbols.

`func`
The top-level function in an expression.

`inf`
The infimum of `self`.

`interior`
Property method which returns the interior of a set.

is_Complement
is_EmptySet
is_Intersection
is_UniversalSet
is_algebraic
is_antihermitian
`is_closed`
A property method to check whether a set is closed.

is_commutative
`is_comparable`
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_empty
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_finite_set
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
`is_open`
Property method to check whether a set is open.

is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero
`kind`

The kind of a Set

length
The length of the line.

measure
The (Lebesgue) measure of `self`.

p1
The first defining point of a linear entity.

p2
The second defining point of a linear entity.

points
The two points used to define this linear entity.

sup
The supremum of `self`.

Methods

`angle_between`(l2)	Return the non-reflex angle formed by rays emanating from the origin with directions the same as the direction vectors of the linear entities.
`arbitrary_point`([parameter])	A parameterized point on the Line.
`are_concurrent`(*lines)	Is a sequence of linear entities concurrent?
`as_content_primitive`([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
`as_dummy`()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
`atoms`(*types)	Returns the atoms that form the current object.
`bisectors`(other)	Returns the perpendicular lines which pass through the intersections of self and other that are in the same plane.
`class_key`()	Nice order of classes.
`compare`(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
`complement`(universe)	The complement of 'self' w.r.t the given universe.
`contains`(other)	Return True if *other* is on this Line, or False otherwise.
`count`(query)	Count the number of matching subexpressions.
`count_ops`([visual])	Wrapper for count_ops that returns the operation count.
`distance`(other)	Finds the shortest distance between a line and a point.
`doit`(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.
`dummy_eq`(other[, symbol])	Compare two expressions and handle dummy symbols.

continues on next page

Table 3 – continued from previous page

encloses(o)	Return True if o is inside (not on or outside) the boundaries of self.
equals(other)	Returns True if self and other are the same mathematical entities
evalf([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
find(query[, group])	Find all subexpressions matching a query.
fromiter(args, **assumptions)	Create a new object from an iterable.
has(*patterns)	Test whether any subexpression matches any of the patterns.
has_free(*patterns)	Return True if self has object(s) x as a free expression else False.
has_xfree(s)	Return True if self has any of the patterns in s as a free argument, else False.
intersect(other)	Returns the intersection of 'self' and 'other'.
intersection(other)	The intersection with another geometrical entity.
is_disjoint(other)	Returns True if self and other are disjoint.
is_parallel(l2)	Are two linear entities parallel?
is_perpendicular(l2)	Are two linear entities perpendicular?
is_proper_subset(other)	Returns True if self is a proper subset of other.
is_proper_superset(other)	Returns True if self is a proper superset of other.
is_similar(other)	Return True if self and other are contained in the same line.
is_subset(other)	Returns True if self is a subset of other.
is_superset(other)	Returns True if self is a superset of other.
isdisjoint(other)	Alias for is_disjoint()
issubset(other)	Alias for is_subset()
issuperset(other)	Alias for is_superset()
match(pattern[, old])	Pattern matching.
matches(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
n([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
parallel_line(p)	Create a new Line parallel to this linear entity which passes through the point *p*.
parameter_value(other, t)	Return the parameter corresponding to the given point.
perpendicular_line(p)	Create a new Line perpendicular to this linear entity which passes through the point *p*.
perpendicular_segment(p)	Create a perpendicular line segment from *p* to this line.
plot_interval([parameter])	The plot interval for the default geometric plot of line.
powerset()	Find the Power set of self.
projection(other)	Project a point, line, ray, or segment onto this linear entity.
random_point([seed])	A random point on a LinearEntity.
rcall(*args)	Apply on the argument recursively through the expression tree.
refine([assumption])	See the refine function in sympy.assumptions
reflect(line)	Reflects an object across a line.
replace(query, value[, map, simultaneous, exact])	Replace matching subexpressions of self with value.
rewrite(*args[, deep])	Rewrite *self* using a defined rule.

continues on next page

Table 3 – continued from previous page

rotate(angle[, pt])	Rotate angle radians counterclockwise about Point pt.
scale([x, y, pt])	Scale the object by multiplying the x,y-coordinates by x and y.
simplify(**kwargs)	See the simplify function in sympy.simplify
smallest_angle_between(l2)	Return the smallest angle formed at the intersection of the lines containing the linear entities.
sort_key([order])	Return a sort key.
subs(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
symmetric_difference(other)	Returns symmetric difference of self and other.
translate([x, y])	Shift the object by adding to the x,y-coordinates the values x and y.
union(other)	Returns the union of self and other.
xreplace(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

class polytex.geometry.basic.**Curve**(*points*)

Bases: object

A partial inheritance of polytex.geometry.Point class.

Parameters

points
[list, tuple or array_like] A list of Point objects.

Attributes

ambient_dimension
Return the dimension of the curve.

bounds
Return the bounds of the curve.

curvature
Return the curvature of the curve.

length
Return the length of the curve.

tangent
Return the tangent vector of the curve at each point.

Methods

plot()	Plot the curve.
save([save_path])	Save the curve to a vtk file.
to_polygon()	Convert the curve to a polygon.

plot()

Plot the curve.

Returns

None

save(*save_path=None*)

Save the curve to a vtk file.

Parameters

save_path
[str] The path to save the vtk file.

Returns

None

to_polygon()

Convert the curve to a polygon.

Returns

polygon
[Polygon object]

property ambient_dimension

Return the dimension of the curve.

Returns

ambient_dimension
[int]

property bounds

Return the bounds of the curve.

Returns

bounds
[tuple]

property curvature

Return the curvature of the curve.

TODO: curvature of a curve

Returns

curvature
[float]

property length

Return the length of the curve.

Returns

length
[float]

property tangent

Return the tangent vector of the curve at each point.

Returns

tangent
[Vector object]

class polytex.geometry.basic.Polygon(*points*)

Bases: *Curve*

A partial inheritance of polytex.geometry.Point class.

Parameters

points
[list, tuple or array_like] A list of Point objects.

Attributes

ambient_dimension
Return the dimension of the curve.

area
Return the area of the polygon.

bounds
Return the bounds of the curve.

centroid
Return the centroid of the polygon.

curvature
Return the curvature of the curve.

length
Return the length of the curve.

perimeter
Return the perimeter of the polygon.

tangent
Return the tangent vector of the curve at each point.

Methods

plot()	Plot the curve.
save([save_path])	Save the curve to a vtk file.
to_curve()	Convert the polygon to a curve.
to_polygon()	Convert the curve to a polygon.

to_curve()

Convert the polygon to a curve.

Returns

curve
[Curve object]

property area

Return the area of the polygon.

Returns

area
[float]

property centroid

Return the centroid of the polygon.

Returns

centroid
[Point object]

property perimeter

Return the perimeter of the polygon.

Returns

perimeter
[float]

class polytex.geometry.basic.Plane(*p1*, *a=None*, *b=None*, ***kwargs*)

Bases: object

Create a plane from 3 points or a point and a normal vector.

Parameters

p1
[Point object] A point on the plane.

a
[Point object, optional] A point on the plane. The default is None.

b
[Point object, optional] A point on the plane. The default is None.

****kwargs**
[dict, optional] *normal_vector* can be passed as a keyword argument when leaving a and b as None. If both a and b are not None, normal_vector will be ignored.

Returns

plane
[Plane object]

Methods

distance(point)	Return the signed distance between a point and the plane.
function()	Return the equation of the plane as a function of x, y and z.
intersection(obj, max_dist)	Return the intersection of the plane with a curve or a polygon.
show()	Plot the plane.

distance(*point*)

Return the signed distance between a point and the plane.

Parameters

point
[Point object] The point to calculate the distance. shape = (n, 3), where n is the number of points.

Returns

distance
[float] The distance between the point and the plane.

Examples

```
>>> from polytex.geometry import Point, Plane
>>> p1 = Point([5, 0, 0])
>>> normal = Point([1, 0, 0])
>>> plane = Plane(p1, normal_vector=normal)
>>> plane.show()
>>> plane.distance([[0, 0, 0], [1, 0, 0], [2, 0, 0]])
array([-5., -4., -3.])
```

function()

Return the equation of the plane as a function of x, y and z.

Parameters

normal
[array-like] normal vector of the plane.

point
[array-like] point on the plane.

Returns

A lambda function of x, y and z.
function of plane.

Notes

normal = (a, b, c) point = (x0, y0, z0) Equation of a plane: a(x-x0) + b(y-y0) + c(z-z0) = 0

Examples

Create a plane from a point and a normal vector >>> from polytex.geometry import Point, Plane >>> p1 = Point([1, 1, 1]) >>> normal = Point([1, 4, 7]) >>> plane2 = Plane(p1, normal_vector=normal) >>> f = plane2.function() >>> f <function polytex.geometry.basic.Plane.function.<locals>.<lambda>(x, y, z)> >>> f(1, 1, 1) 0

`intersection`(*obj*, *max_dist*)

Return the intersection of the plane with a curve or a polygon.

Parameters

obj
[Curve or Polygon object] The object to intersect with the plane.

max_dist
[float] The maximum distance of checked points from the plane.

Returns

intersection
[list or array] The intersection point of the plane with the obj in the shape of (1, 3). If the intersection is not found, none is returned.

`show`()

Plot the plane.

Returns

None

class `polytex.geometry.basic.Ellipse2D`(*n: int*, *a: float*, *b: float*, *center=[0, 0]*)

Bases: `object`

Parameters

n
[int] The number of points to generate.

a
[float] The length of the major axis.

b
[float] The length of the minor axis.

center
[list, tuple or array_like] The center of the ellipse.

Methods

elipse_2d()	Generate points on an ellipse.

elipse_2d() → ndarray

Generate points on an ellipse.

Returns

xy: array-like
Points on the ellipse with shape (n, 2).

class polytex.geometry.basic.**Tube**(*theta_res*, *h_res*, *vertices=None*, ***kwargs*)

Bases: `GeometryEntity`

Create a tubular surface.

Parameters

theta_res
[int] The number of points on each cross-section.

h_res
[int] The number of cross-sections.

vertices
[array_like] The points on the cross-sections. The shape of the array should be (h_res * theta_res, 3). The points should be ordered in the following way: [p1, p2, ..., p_theta_res, p1, p2, ..., p_theta_res, ..., p1, p2, ..., p_theta_res] where p1, p2, ..., p_theta_res are the points on each cross-section from the top to the bottom. The default value is None. If the value is None, the points will be generated automatically by assigning the height, major and minor radius to the tube.

Returns

tube
[Tube object]

Examples

```
>>> from polytex.geometry import Tube
>>> tube = Tube(4, 10, major=2, minor=1, h=5)
>>> mesh = tube.mesh(plot=True)
>>> tube.save_as_mesh('tube.vtk')
```

Attributes

args
Returns a tuple of arguments of 'self'.

assumptions0
Return object *type* assumptions.

bounds
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

`canonical_variables`
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

expr_free_symbols
`free_symbols`
Return from the atoms of self those which are free symbols.

`func`
The top-level function in an expression.

h_res
is_algebraic
is_antihermitian
is_commutative
`is_comparable`
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero
points
theta_res

Methods

as_content_primitive([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
as_dummy()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
atoms(*types)	Returns the atoms that form the current object.
class_key()	Nice order of classes.
compare(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
count(query)	Count the number of matching subexpressions.
count_ops([visual])	Wrapper for count_ops that returns the operation count.
doit(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.
dummy_eq(other[, symbol])	Compare two expressions and handle dummy symbols.
encloses(o)	Return True if o is inside (not on or outside) the boundaries of self.
evalf([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
find(query[, group])	Find all subexpressions matching a query.
fromiter(args, **assumptions)	Create a new object from an iterable.
has(*patterns)	Test whether any subexpression matches any of the patterns.
has_free(*patterns)	Return True if self has object(s) x as a free expression else False.
has_xfree(s)	Return True if self has any of the patterns in s as a free argument, else False.
intersection(o)	Returns a list of all of the intersections of self with o.
match(pattern[, old])	Pattern matching.
matches(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
mesh([plot, show_edges])	TODO : raise TypeError("Given points must be a sequence or an array.")
n([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
rcall(*args)	Apply on the argument recursively through the expression tree.
refine([assumption])	See the refine function in sympy.assumptions
replace(query, value[, map, simultaneous, exact])	Replace matching subexpressions of self with value.
rewrite(*args[, deep])	Rewrite *self* using a defined rule.
rotate(angle[, pt])	Rotate angle radians counterclockwise about Point pt.
save_as_mesh(save_path[, end_closed])	Save the tubular mesh to a file.
scale([x, y, pt])	Scale the object by multiplying the x,y-coordinates by x and y.
simplify(**kwargs)	See the simplify function in sympy.simplify

continues on next page

Table 4 – continued from previous page

`sort_key([order])`	Return a sort key.
`subs(*args, **kwargs)`	Substitutes old for new in an expression after sympifying args.
`translate([x, y])`	Shift the object by adding to the x,y-coordinates the values x and y.
`xreplace(rule)`	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

mesh(*plot=False*, *show_edges=True*)

TODO : raise TypeError("Given points must be a sequence or an array.")

Notes

theta_res should be 1 less then else where. To be fixed in the future.

save_as_mesh(*save_path*, *end_closed=True*)

Save the tubular mesh to a file. The file format is determined by the extension of the filename. The possible file formats are: [".ply", ".stl", ".vtk", ".vtu"].

TODO : There seems to be a bug in correction option of the to_meshio_data() method of the tubular mesh.

Parameters

save_path
[str] The path and the name of the file to be saved with the extension.

end_closed
[bool] If True, the ends of the tube will be closed. The default value is True.

Returns

mesh
[pyvista.UnstructuredGrid] The tubular mesh.

Examples

```
>>> from polytex.geometry import Tube
>>> tube = Tube(5,10,major=2, minor=1,h=5)
>>> tube.save_as_mesh('tube.vtu')
```

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

2.1.3 Parametric geometries

`class polytex.geometry.basic.ParamCurve(limits, function=[], dataset=None, krig_config=('lin', 'cub'), smooth=0.0, verbose=False)`

Bases: `object`

Parameters

limits
[3-tuple] Function parameter and lower and upper bounds.

function
[list] The function list for each coordinate component. The default value is [].

dataset
[array_like] The dataset of the curve. The default value is None. The first column is the parameter and the other columns are the value of coordinate components.

One of the function or dataset must be given. Please note that both are given, the dataset will be ignored.

krig_config
[tuple] The kriging interpolation configuration. The default value is ("lin", "cub"). The tuple should be in the form of (drift_name, covariance_name).

smooth
[float] The smoothing factor. The default value is 0.0.

verbose
[bool] If True, print the information of the kriging process. The default value is False.

Returns

curve
[ParamCurve object]

Methods

eval(t_value)	Evaluate the curve at a given parameter value.

bounds	

`eval(t_value)`

Evaluate the curve at a given parameter value. The parameter value should be within the limits. Otherwise, an error will be raised.

Parameters

t_value
[float or array_like] The parameter value for evaluation.

Returns

curve
[Curve object] The evaluated curve.

class polytex.geometry.basic.**ParamSurface**(*functions*, *limits*, ***kwargs*)

Bases: GeometryEntity

Parameters

functions
[list of functions] Function argument should be (x(s, t), y(s, t), z(s, t)) for a 3D surface.

limits
[2-tuple] Function parameter and lower and upper bounds of the two parameters. For example, ((s, 0, 1), (t, 0, 1)) is valid. The parameter should be the same for all three functions.

Attributes

args
Returns a tuple of arguments of 'self'.

assumptions0
Return object *type* assumptions.

bounds
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

canonical_variables
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

expr_free_symbols

free_symbols
Return from the atoms of self those which are free symbols.

func
The top-level function in an expression.

functions
The functions specifying the surface.

is_algebraic

is_antihermitian

is_commutative

is_comparable
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero

limits
The limits of the two parameters specifying the surface.

Methods

as_content_primitive([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
as_dummy()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
atoms(*types)	Returns the atoms that form the current object.
class_key()	Nice order of classes.
compare(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
count(query)	Count the number of matching subexpressions.
count_ops([visual])	Wrapper for count_ops that returns the operation count.
doit(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.

continues on next page

Table 5 – continued from previous page

dummy_eq(other[, symbol])	Compare two expressions and handle dummy symbols.
encloses(o)	Return True if o is inside (not on or outside) the boundaries of self.
eval(s_value, t_value)	Evaluate the surface at the given parameter values.
evalf([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
find(query[, group])	Find all subexpressions matching a query.
fromiter(args, **assumptions)	Create a new object from an iterable.
has(*patterns)	Test whether any subexpression matches any of the patterns.
has_free(*patterns)	Return True if self has object(s) `x` as a free expression else False.
has_xfree(s)	Return True if self has any of the patterns in s as a free argument, else False.
intersection(o)	Returns a list of all of the intersections of self with o.
match(pattern[, old])	Pattern matching.
matches(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
n([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
rcall(*args)	Apply on the argument recursively through the expression tree.
refine([assumption])	See the refine function in sympy.assumptions
replace(query, value[, map, simultaneous, exact])	Replace matching subexpressions of `self` with `value`.
rewrite(*args[, deep])	Rewrite *self* using a defined rule.
rotate(angle[, axis])	Rotate `angle` radians counterclockwise about Point `pt`.
scale([x, y, z])	Scale the object by multiplying the x,y-coordinates by x and y.
simplify(**kwargs)	See the simplify function in sympy.simplify
sort_key([order])	Return a sort key.
subs(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
translate([x, y, z])	Shift the object by adding to the x,y-coordinates the values x and y.
xreplace(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

eval(*s_value*, *t_value*)

Evaluate the surface at the given parameter values.

Parameters

s_value
[float or array_like] The parameter value for evaluation.

t_value
[float or array_like] The parameter value for evaluation.

Returns

Surface
The surface evaluated at the given parameter values. the shape of the returned surface is (len(s) * len(t), 3). The first column is s values, the second column is t values, and the following columns are the coordinates of the surface at the given parameter values (x, y, z).

rotate(*angle*, *axis='z'*)

Rotate `angle` radians counterclockwise about Point `pt`.

The default pt is the origin, Point(0, 0)

See also:

scale, *translate*

Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2, -1/2))
```

scale(*x=1*, *y=1*, *z=1*)

Scale the object by multiplying the x,y-coordinates by x and y.

If pt is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and shifted by pt.

See also:

rotate, *translate*

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/2))
>>> t.scale(2, 2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

translate(*x=0*, *y=0*, *z=0*)

Shift the object by adding to the x,y-coordinates the values x and y.

See also:

rotate, *scale*

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2), Point2D(3/2, 2 - sqrt(3)/
↪2))
```

property functions

The functions specifying the surface.

Returns

functions
list of parameterized coordinate functions.

Examples

```
>>> from sympy.abc import t
>>> from polytex.geometry import ParamCurve3D
>>> surface = ParamSurface((t, t, t), ((t, 0, 1), (t, 0, 1)))
>>> surface.functions
(t, t, t)
```

property limits

The limits of the two parameters specifying the surface.

Returns

list limits of the parameters.

Examples

```
>>> from sympy.abc import t
>>> from polytex.geometry import ParamCurve3D
>>> surface = ParamSurface((t, t, t), ((t, 0, 1), (t, 0, 1)))
>>> surface.limits
((t, 0, 1), (t, 0, 1))
```

2.2 Functions

2.2.1 polytex.io

Module contents

class polytex.io.**save_krig**(**args*, ***kwargs*)

Bases: dict

This class saves a dictonary of sympy expressions to a file in human readable form and then load as sympy expressions directly without other conversion. It is called by polytex.io.pk_save to save kriging expressions to a ".krig" file and by polytex.io.pk_load to load these files. Therefore, the class is not intended to be used directly by the user.

Notes

This class is taken from: https://github.com/sympy/sympy/issues/7974. A bug in exec() is fixed and some modifications are made to make it fit for the purpose of this project (store the kriging expression).

Examples

```
>>> import sympy
>>> from polytex.io import save_krig
>>> a, b = sympy.symbols('a, b')
>>> d = save_krig({'a':a, 'b':b})
>>> d.save('name.krig')
>>> del d
>>> d2 = save_krig.load('name.krig')
```

Methods

clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(key[, default])	If the key is not found, return the default if given; otherwise, raise a KeyError.
popitem(/)	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

load	
save	

classmethod **load**(*file_path*)

save(*file*)

polytex.io.**case_prepare**(*output_dir*)

Load openFoam case template and prepare the case for simulation.

Parameters

output_dir
[str] The output directory where the 0 and constant folders are located.

Returns

None.

polytex.io.**cell_faces**(*mesh*, *ind*, *neighbor=False*)

Get all faces of a 3D cell.

Parameters

mesh
[vtkUnstructuredGrid] The volume mesh.

ind
[int] The cell id.

neighbor
[bool, optional] If True, return the neighbor cell ids. The default is False.

Returns

faces
[list] A list containing all faces of the cell.

neighbors
[list] A list containing all neighbor cell ids of the cell. Not returned if neighbor is False.

`polytex.io.`**`choose_directory`**(*titl='Select the target directory:'*)

Choose a directory with GUI and return its path.

Parameters

titl: String.
The title of the open folder dialog window.

Returns

path: String.
The path of the selected directory.

`polytex.io.`**`choose_file`**(*titl='Select the target directory:'*, *format='csv'*)

Choose a file with GUI and return its path.

Parameters

titl: String.
The title of the window.

Returns

path: String.
The path of the file.

`polytex.io.`**`coo_to_ply`**(*file_coo*, *file_ply*, *interpolate=False*, *threshold=0.1*)

Convert a pcd file to ply file.

Parameters

file_coo
[str] The path of the coo file or pathlib.Path. File or filename to which the data is saved.

file_ply
[str] The path of the ply file or pathlib.Path. File or filename to which the data is to be saved.

interpolate
[bool, optional] Whether to interpolate the points. The default is False.

threshold
[float, optional] The threshold of the normalized distance between the neighboring points. The default is 0.1.

Returns

None

`polytex.io.`**`create_material_data_lines`**(*matname*, *rho*, *e*, *nu*, *sta*, *condition=True*, *materials=None*)

Creates lines for defining material data in the input file.

Parameters

matname
[str] Name of the material.

rho
[float] Density of the material.

e
[float] Young's modulus of the material.

nu
[float] Poisson's ratio of the material.

sta
[int] Material state variable.

condition
[bool] Condition for material type (default is True).

materials: list
List to store material data lines (default is None).

Returns

datalines
[list] A list of lines for material data.

polytex.io.**create_part_data_lines**(*prtname*, *nodes=[]*, *elements=[]*, *nodesets=[]*, *elemsets=[]*)

Creates lines for defining nodes, elements, and sets in the part data.

Parameters

prtname
[str] Name of the part.

nodes
[list] Node data. Each node is a list containing the node label and the x, y, and z coordinates. The number of nodes can be obtained using the len() function.

elements
[list] Element data. Each element is a list containing the element label and the node labels.

nodesets
[list] Node sets containing the node labels.

elemsets
[list] Element sets containing the element labels.

Returns

lines
[list] List of lines for the part data.

polytex.io.**create_solid_section_for_all_sets**(*Indices*, *fiber_material_name*, *orientation_name=None*, *controls=''*)

Creates solid section lines for all fiber element sets in the mesh.

Parameters

Indices: int
The indices of matrix and fiber.

fiber_material_name
[str] The name of the fiber material.

orientation_name
[str] The name of the orientation (default is None for the matrix).

controls
[str] Control parameters for the section (default is an empty string).

Returns

lines
[list] A list of lines for defining solid section properties for all fiber element sets.

`polytex.io.create_solid_section_lines`(*elset_name*, *material_name*, *orientation_name=None*, *controls=''*)

Creates lines for defining the solid section properties of a specified element set and material.

Parameters

elset_name
[str] The name of the element set.

material_name
[str] The name of the material.

orientation_name
[str] The name of the orientation (default is None for the matrix).

controls
[str] Control parameters for the section (default is an empty string).

Returns

lines
[list] A list containing lines for defining the solid section properties.

`polytex.io.create_yarn_element_sets`(*mesh*, *file_handle*, *Indices*, *verbose=False*)

Creates element sets for each unique fiber in the mesh.

Parameters

mesh
[pyvista.PolyData]

The input mesh.

file_handle
[file] The file handle to write element set lines.

Indices: int
The indices of matrix and fiber.

Returns

element_sets
[dict] Dictionary of element sets with the set name as the key and the lines as the value.

`polytex.io.cwd_chdir`(*path=''*)

Set given directory or the folder where the code file is as current working directory

Parameters

path:
the path of current working directory. if empty, the path of the code file is used.

Returns

cwd: the current working directory.

`polytex.io.filenames(path, filter='csv')`

Get the list of files in the given folder.

Parameters

path:
the path of the folder

filter:
filter for file selection.++

Returns

flst: the list of files in the given folder.

`polytex.io.get_boundary_faces(volume)`

Extract boundary faces from a vtkUnstructuredGrid object.

Parameters

volume
[pyvista.UnstructuredGrid] The volume mesh.

Returns

boundary_faces
[numpy.ndarray] A numpy array containing all boundary faces.

surf
[pyvista.PolyData] A pyvista surface mesh object.

`polytex.io.get_internel_faces(volume)`

Extract internel faces from a vtkUnstructuredGrid object.

Parameters

volume
[pyvista.UnstructuredGrid] The volume mesh.

Returns

internal_faces
[list] A list containing all internal faces.

owner
[list] A list containing all owner cell ids corresponding to the internal faces.

neighbour
[list] A list containing all neighbour cell ids corresponding to the internal faces.

`polytex.io.get_ply_property(mesh_path, column, skip=11, type='vertex', save_vtk=False)`

This function get a vertex property or cell property from a mesh stored as .ply format. It is intended to be used to get the user-defined properties that most of meshing and rendering software does not support.

Parameters

mesh_path
[str] The path of the mesh file with .ply extension.

column
[int or list of int] The column number of the property.

skip
[int, optional] The number of lines to skip in the header. The default is 11.

type
[str, optional] The type of the property. The default is "vertex" for vertex property. The other possible value is "cell" for cell property.

save_vtk
[bool, optional] If True, the mesh is saved as a vtk file. The default is False.

Returns

property
[numpy.ndarray] The property of the mesh.

Notes

The mesh must be saved as ASCII format.

Examples

```
>>> import polytex as pk
>>> mesh_path = "./weft_0_lin_lin_krig_30pts.ply"
>>> quality = ptx.get_ply_property(mesh_path, -2, skip=11, type="vertex", save_
↪vtk=False)
>>> quality
```

`polytex.io.meshio_save`(*file*, *vertices*, *cells=[]*, *point_data={}*, *cell_data={}*, *binary=False*)

Save surface mesh as a mesh file by definition of vertices and faces. Point data and cell data can be added. It is a wrapper of meshio.write() function.

Parameters

file
[str] The path of the ply file or pathlib.Path. File or filename to which the data is saved.

vertices
[numpy.ndarray] The vertices of the mesh. The shape of the array is (n, 3), where n is the number of vertices.

cells
[list, optional] The faces of the mesh stored as the connectivity between vertices. The default is [].

point_data
[dict, optional] The point data of the mesh. The default is {}.

cell_data
[dict, optional] The cell data of the mesh. The default is {}. Note that the cell data should be added as a list of arrays. Each array in the list corresponds to a cell type. For example, if the mesh has 2 triangles and 1 quad, namely, cells = [("triangle", [0, 1, 2], [1,2,3]), ("quad", [3, 4, 5, 6])], then the cell data should be added as cell_data = {"data": [[1, 2], [3]}.

binary
[bool, optional] If True, the data is written in binary format. The default is False.

Returns

None.

Examples

```
>>> import numpy as np
>>> import polytex as pk
>>> vertices = np.array([[0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]])
>>> cells = [("triangle", [[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]])]
>>> point_data = {"a": np.array([0, 1, 2, 3])}
>>> cell_data = {"b": np.array([[0, 1, 2, 3],])}
>>> ptx.meshio_save("test.ply", vertices, cells, point_data, cell_data)
>>> print("Done")
Done
```

polytex.io.**mkdir**(*path*)

Create the output directory if it does not exist.

Parameters

path: str
The path of the output directory.

Returns

output_dir: str
The path of the output directory.

polytex.io.**pcd_to_ply**(*file_pcd*, *file_ply*, *binary=False*)

Convert a pcd file to ply file.

Parameters

file_pcd
[str] The path of the pcd file or pathlib.Path. File or filename to which the data is saved.

file_ply
[str] The path of the ply file or pathlib.Path. File or filename to which the data is to be saved.

binary
[bool, optional] TODO

Returns

None

polytex.io.**pk_load**(*file*)

Load a file format defined in polytex (.coo, .geo, or .stat) file and return as a pandas dataframe or a numpy.array object.

Parameters

file: str, or pathlib.Path.
File path and name to which the data is stored.

Returns

df: pandas.DataFrame or numpy.ndarray
The data to be loaded. It is a pandas dataframe if the file is a .coo/geo file. Otherwise, it is a numpy array or dict and a warning will be raised.

polytex.io.**pk_save**(*fp*, *data*, *check_format=True*)

Save a Python dict or pandas dataframe as a file format defined in polytex (.coo, geo) file

Parameters

fp: str
File path and name to which the data is saved. If the file name does not end with a supported file extension, a ValueError will be raised.

data: Tow, Tex, or dict
The data to be saved. It can be several customised file formats for polytex.

Returns

None

`polytex.io.`**`read_explicit_data`**(*filename*, *type='zip'*, *sort=True*, *resolution=1.0*, *max_pts=100*, *verbose=False*)

Read ROI data from csv files exported from manual segmentation in ImageJ/FIJI. See https://www.binyang.fun/manual-segmentation-in-imagej-fiji/ for more details.

Parameters

filename
[str] The path of the roi file. The file should be either a zip of csv files or a directory containing multiple csv files. Each csv file contains the coordinates of the segmented points on a slice. see https://www.binyang.fun/manual-segmentation-in-imagej-fiji/ for more details. The parameter "type" should be set accordingly ("zip" or "dir").

type
[str, optional] The type of saved file. The default is "zip". The other option is "dir".

sort
[bool, optional] Whether to sort the coordinates according to the slice number. The default is True. Note that the coordinates on the same slice are not sorted. The sorting is only applied to the slices.

resolution
[float, optional] The resolution of the image. The default is 1.0, the coordinates are not converted to the physical coordinates (namely the unit is pixel).

max_pts
[int, optional] The maximum number of points on each slice. The default is 100. If the number of points on a slice is larger than max_pts, the points will be uniformly sampled to max_pts (approximately).

Returns

surf_points
[numpy.ndarray] The coordinates of the segmented points on the surface of the tow in shape (N, 3), where N is the total number of points.

`polytex.io.`**`read_imagej_roi`**(*filename*, *type='zip'*, *sort=True*, *resolution=1.0*, *max_pts=100*, *verbose=False*)

`polytex.io.`**`save`**(*file*, *arr*, *allow_pickle=True*, *fix_imports=True*)

This is an exact copy of numpy.save, except that it does not check the extensions.

Parameters

file
[file, str, or pathlib.Path. File or filename to which the data is saved.]

arr
[array_like. Array data to be saved.]

allow_pickle
[bool, optional] Allow saving object arrays using Python pickles. Reasons for disallowing

pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3). Default: True

fix_imports
[bool, optional] Only useful in forcing objects in object arrays on Python 3 to be pickled in a Python 2 compatible way. If *fix_imports* is True, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

`polytex.io.save_csv`(*filename*, *dataset*, *csv_head*)

Save numpy array to csv file with given info in the first row.

Parameters

filename:
The path and name of the csv file.

dataset: List or numpy.ndarray
The dataset to be saved in the csv file

csv_head:
A list of headers of the csv file. The length of the list should be the same as the number of columns in the dataset.

Returns

None.

`polytex.io.save_nrrd`(*cell_label*, *file_name*, *file_path='./'*)

Save the labels of a hexahedral mesh to a nrrd file. The labels should be starting from 0 and increasing by 1.

Parameters

cell_label: numpy array(int, int, int)
The cell label of the mesh.

file_name: String
The name of the .nrrd file.

file_path: String
The save path of the .nrrd file.

Returns

None

`polytex.io.save_ply`(*file*, *vertices*, *cells=[]*, *point_data={}*, *cell_data={}*, *binary=False*)

`polytex.io.texgen_voxel`(*mesh*, *rf*, *perm_model='Gebart'*, *fiber_packing='Hex'*, *plot=False*, *scalar='YarnIndex'*, *progress_bar=True*)

Read the vtu voxel mesh exported from TexGen and calculate necessary information for OpenFOAM polyMesh conversion.

Parameters

mesh
[pyvista.DataSet] The voxel mesh exported from TexGen.

rf
[float] The fiber radius (m).

perm_model
[str, optional] The yarn permeability model. The default is "Gebart".

fiber_packing
[str, optional] The fiber packing pattern used for yarn permeability calculation. The default is "Hex". Valid options are "Quad" and "Hex".

plot
[bool, optional] If True, plot the mesh. The default is False.

scalar
[str, optional] The scalar to plot. The default is "YarnIndex".

Returns

mesh
[pyvista.UnstructuredGrid] The voxel mesh with the new data.

`polytex.io.`**`voxel2foam`**(*mesh*, *scale=1*, *outputDir='./'*, *boundary_type=None*, *cell_data_list=None*) → None

Convert a voxel mesh to OpenFOAM mesh. The cell data is converted to OpenFOAM initial conditions and saved in the 0 timestep folder.

Parameters

mesh
[pyvista.UnstructuredGrid or pyvista.DataSet] The voxel mesh.

scale
[float, optional] The scale factor to convert the unit of points. The default is 1.0.

outputDir
[str, optional] The output directory. The default is './'.

boundary_type
[dict, optional] The type of each boundary. The default is None. If None, the type of the boundary is set as "patch". The key is the boundary name and the value is the boundary type. The key should be the same as the face_boundary_dict.

The boundary type can be "patch", "wall", "empty", "symmetryPlane", "wedge", "cyclic", etc. See OpenFOAM user guide for more details.

cell_data_list
[list, optional] A list containing the names of the cell data to be written. The default is None.

Returns

None.

`polytex.io.`**`voxel2img`**(*mesh*, *mesh_shape*, *dataset='YarnIndex'*, *save_path='./img/'*, *scale=None*, *img_name='img'*, *format='tif'*, *scale_algrithm='linear'*)

Convert a voxel mesh to a series of images.

Parameters

mesh
[pyvista.UnstructuredGrid] The voxel mesh to convert.

mesh_shape
[list] The number of cells in each direction of the mesh [nx, ny, nz].

dataset
[str, optional] The name of the cell data to convert. The default is "YarnIndex".

save_path
[str, optional] The path to save the images. The default is "./img/".

scale
[int] The scale factor of the image. The default is None.

img_name
[str, optional] The name of the output image. The default is "img". The slice number will be added to the end of the name and separated by an underscore.

format
[str, optional] The format of the output image. The default is "tif".

scale_algrithm
[str, optional] The algorithm used to scale the pixel numbers of the image. The default is "linear". The other option is "spline".

TODO: The "spline" algorithm is only working for x and y directions yet.
The z direction is to be implemented.

Returns

None

Examples

```
>>> import pyvista as pv
>>> import polytex as ptx
>>> mesh = pv.read("./v2i.vtu")
>>> mesh_shape = [20, 20, 5]
>>> ptx.io.voxel2img(mesh, mesh_shape, dataset="YarnIndex",
                    save_path="./img/",
                    scale=50, img_name="img", format="tif",
                    scale_algrithm="linear")
```

`polytex.io.`**`voxel2inp`**(*mesh*, *scale=1*, *outputDir='./mesh-C3D8R.inp'*, *orientation=True*) → None

Convert a voxel mesh to an Abaqus input file.

Parameters

mesh
[pyvista.UnstructuredGrid] The voxel mesh.

scale
[float, optional] The scale factor to convert the unit of points. The default is 1.0.

outputDir
[str, optional] The output directory and filename. The default is './mesh-C3D8R.inp'. The file extension is automatically added if not provided.

Returns

None.

Notes

voxel2inp is developed by Chao Yang (yangchaogg@whut.edu.cn) & Bin Yang (bin.yang@polymtl.ca) jointly. Please contact us if you have any questions.

polytex.io.**write_FoamFile**(*ver*, *fmt*, *cls*, *location*, *obj*, *top_separator*)

polytex.io.**write_boundary**(*face_boundary_dict*, *start_face*, *output_dir='./constant/polyMesh/'*, *type=None*)

Boundary file writing.

Parameters

face_boundary_dict
[dict] A dict contains boundary category. The key is the boundary name and the value is a numpy array containing all boundary face node indices. The boundary name should be the same as the boundary patch name in the boundary file.

start_face
[int] The start face index of the boundary faces. equal to the number of internal faces.

output_dir
[str, optional] The output directory. The default is './constant/polyMesh/'.

type
[dict, optional] The type of each boundary. The default is None. If None, the type of the boundary is set as "patch". The key is the boundary name and the value is the boundary type. The key should be the same as the face_boundary_dict.

The boundary type can be "patch", "wall", "empty", "symmetryPlane", "wedge", "cyclic", etc. See OpenFOAM user guide for more details.

Returns

None.

polytex.io.**write_cell_data**(*cellDataDict*, *outputDir='./0/'*, *array_list=None*)

Write cell data to OpenFOAM format

Parameters

cellDataDict
[dict] A dictionary to store all cell data sets in vtu file using the data set name as key.

outputDir
[str, optional] The directory to store the converted data. The default is './0/'.

array_list
[list, optional] A list containing the names of the cell data to be written. The default is None,

polytex.io.**write_cell_zone**(*cell_zone*, *output_dir='./constant/polyMesh/'*)

Write the cells to a file for porous properties setting.

Parameters

cell_zone
[dict] A dict containing all cell ids corresponding to the cell zones. The key is the cell zone name and the value (a list) is the cell ids in the zone.

output_dir
[str, optional] The output directory. The default is './constant/polyMesh/'.

Returns

None.

`polytex.io.write_face`(*face_points*)

Parameters

face_points
[list] A list containing all face node indices.

`polytex.io.write_faces`(*internal_faces*, *face_boundary*, *output_dir='./constant/polyMesh/'*)

Write the faces file.

Parameters

internal_faces
[list] A list containing all internal face node indices.

face_boundary
[dict] A dict containing all boundary faces. The key is the boundary patch name, and the value is a numpy array containing all boundary face node indices.

output_dir
[str, optional] The output directory. The default is './constant/polyMesh/'.

Returns

None.

`polytex.io.write_fiber_orientation_to_file`(*mesh*, *Indices*, *file_header=''*, *output_file='fabrictest.ori'*)

Writes fiber orientation information to a file.

Parameters

mesh: pyvista.PolyData
The input mesh.

Indices: int
The indices of matrix and fiber.

file_header: str
The header lines for the ori file (default is '').

output_file: str
The output file path for writing fiber orientation information (default is 'fabrictest.ori').

Returns

None

`polytex.io.write_neighbors`(*neighbour*, *output_dir='./constant/polyMesh/'*)

Write the neighbors file.

Parameters

neighbour
[list] A list containing all neighbor cell ids corresponding to the internal faces.

output_dir
[str, optional] The output directory. The default is './constant/polyMesh/'.

Returns

None.

`polytex.io.write_owner`(*owner_internal*, *owner_boundary*, *output_dir='./constant/polyMesh/'*)

Write the owner file.

Parameters

owner_internal
[array-like] A list containing all owner cell ids corresponding to the internal faces.

owner_boundary
[dict] A list containing all owner cell ids corresponding to the boundary faces.

output_dir
[str, optional] The output directory. The default is './constant/polyMesh/'.

Returns

None.

`polytex.io.write_points`(*points*, *output_dir='./constant/polyMesh/'*, *scale=1.0*)

Write points to OpenFOAM format

Parameters

points
[array-like] The points to be written. The shape of the array should be (n, 3).

output_dir
[str, optional] The directory to store the converted data. The default is './', which means the current directory.

scale
[float, optional] The scale factor to convert the unit of points. The default is 1.0.

Returns

: int
1 if the writing is successful.

`polytex.io.zip_files`(*directory*, *file_list*, *filename*, *remove='True'*)

Add multiple files to a zip file.

Parameters

directory: String.
The directory of the files to be added to zip file. Therefore, all the files in the file_list should be in the same directory.

file_list
[List.] The list of file names to be added to the zip file (without directory).

filename: String.
The name of the zip file. The zip file is saved in the same directory

remove:
Whether to remove original files after adding to zip file. Default is True. If False, the original files will not be removed.

Returns

None.

2.2.2 polytex.geometry

Submodules

polytex.geometry.basic

`class polytex.geometry.basic.Curve(points)`

Bases: `object`

A partial inheritance of polytex.geometry.Point class.

Parameters

points
[list, tuple or array_like] A list of Point objects.

Attributes

ambient_dimension
Return the dimension of the curve.

bounds
Return the bounds of the curve.

curvature
Return the curvature of the curve.

length
Return the length of the curve.

tangent
Return the tangent vector of the curve at each point.

Methods

plot()	Plot the curve.
save([save_path])	Save the curve to a vtk file.
to_polygon()	Convert the curve to a polygon.

`plot()`

Plot the curve.

Returns

None

`save(save_path=None)`

Save the curve to a vtk file.

Parameters

save_path
[str] The path to save the vtk file.

Returns

None

`to_polygon()`

Convert the curve to a polygon.

Returns

polygon
[Polygon object]

`property ambient_dimension`

Return the dimension of the curve.

Returns

ambient_dimension
[int]

`property bounds`

Return the bounds of the curve.

Returns

bounds
[tuple]

`property curvature`

Return the curvature of the curve.

TODO: curvature of a curve

Returns

curvature
[float]

`property length`

Return the length of the curve.

Returns

length
[float]

`property tangent`

Return the tangent vector of the curve at each point.

Returns

tangent
[Vector object]

`class polytex.geometry.basic.Ellipse2D`(*n: int*, *a: float*, *b: float*, *center=[0, 0]*)

Bases: `object`

Parameters

n
[int] The number of points to generate.

a
[float] The length of the major axis.

b
[float] The length of the minor axis.

center
[list, tuple or array_like] The center of the ellipse.

Methods

elipse_2d()	Generate points on an ellipse.

`elipse_2d()` → ndarray

Generate points on an ellipse.

Returns

xy: array-like
Points on the ellipse with shape (n, 2).

class `polytex.geometry.basic.Line`(*p1*, *p2=None*, ***kwargs*)

Bases: `Line`

Parameters

p1
[Point object] The first point of the line.

p2
[Point object] The second point of the line.

Attributes

`ambient_dimension`
A property method that returns the dimension of LinearEntity object.

`args`
Returns a tuple of arguments of 'self'.

`assumptions0`
Return object *type* assumptions.

`boundary`
The boundary or frontier of a set.

`bounds`
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

`canonical_variables`
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

`closure`
Property method which returns the closure of a set.

`direction`
The direction vector of the LinearEntity.

expr_free_symbols
`free_symbols`
Return from the atoms of self those which are free symbols.

`func`
The top-level function in an expression.

inf
The infimum of `self`.

interior
Property method which returns the interior of a set.

is_Complement
is_EmptySet
is_Intersection
is_UniversalSet
is_algebraic
is_antihermitian
is_closed
A property method to check whether a set is closed.

is_commutative
is_comparable
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_empty
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_finite_set
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_open
Property method to check whether a set is open.

is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero
kind
The kind of a Set

length

The length of the line.

measure
The (Lebesgue) measure of `self`.

p1
The first defining point of a linear entity.

p2
The second defining point of a linear entity.

points
The two points used to define this linear entity.

sup
The supremum of `self`.

Methods

`angle_between`(l2)	Return the non-reflex angle formed by rays emanating from the origin with directions the same as the direction vectors of the linear entities.
`arbitrary_point`([parameter])	A parameterized point on the Line.
`are_concurrent`(*lines)	Is a sequence of linear entities concurrent?
`as_content_primitive`([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
`as_dummy`()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
`atoms`(*types)	Returns the atoms that form the current object.
`bisectors`(other)	Returns the perpendicular lines which pass through the intersections of self and other that are in the same plane.
`class_key`()	Nice order of classes.
`compare`(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
`complement`(universe)	The complement of 'self' w.r.t the given universe.
`contains`(other)	Return True if *other* is on this Line, or False otherwise.
`count`(query)	Count the number of matching subexpressions.
`count_ops`([visual])	Wrapper for count_ops that returns the operation count.
`distance`(other)	Finds the shortest distance between a line and a point.
`doit`(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.
`dummy_eq`(other[, symbol])	Compare two expressions and handle dummy symbols.
`encloses`(o)	Return True if o is inside (not on or outside) the boundaries of self.

continues on next page

Table 6 – continued from previous page

`equals`(other)	Returns True if self and other are the same mathematical entities
`evalf`([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
`find`(query[, group])	Find all subexpressions matching a query.
`fromiter`(args, **assumptions)	Create a new object from an iterable.
`has`(*patterns)	Test whether any subexpression matches any of the patterns.
`has_free`(*patterns)	Return True if self has object(s) `x` as a free expression else False.
`has_xfree`(s)	Return True if self has any of the patterns in s as a free argument, else False.
`intersect`(other)	Returns the intersection of 'self' and 'other'.
`intersection`(other)	The intersection with another geometrical entity.
`is_disjoint`(other)	Returns True if `self` and `other` are disjoint.
`is_parallel`(l2)	Are two linear entities parallel?
`is_perpendicular`(l2)	Are two linear entities perpendicular?
`is_proper_subset`(other)	Returns True if `self` is a proper subset of `other`.
`is_proper_superset`(other)	Returns True if `self` is a proper superset of `other`.
`is_similar`(other)	Return True if self and other are contained in the same line.
`is_subset`(other)	Returns True if `self` is a subset of `other`.
`is_superset`(other)	Returns True if `self` is a superset of `other`.
`isdisjoint`(other)	Alias for `is_disjoint()`
`issubset`(other)	Alias for `is_subset()`
`issuperset`(other)	Alias for `is_superset()`
`match`(pattern[, old])	Pattern matching.
`matches`(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
`n`([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
`parallel_line`(p)	Create a new Line parallel to this linear entity which passes through the point *p*.
`parameter_value`(other, t)	Return the parameter corresponding to the given point.
`perpendicular_line`(p)	Create a new Line perpendicular to this linear entity which passes through the point *p*.
`perpendicular_segment`(p)	Create a perpendicular line segment from *p* to this line.
`plot_interval`([parameter])	The plot interval for the default geometric plot of line.
`powerset`()	Find the Power set of `self`.
`projection`(other)	Project a point, line, ray, or segment onto this linear entity.
`random_point`([seed])	A random point on a LinearEntity.
`rcall`(*args)	Apply on the argument recursively through the expression tree.
`refine`([assumption])	See the refine function in sympy.assumptions
`reflect`(line)	Reflects an object across a line.
`replace`(query, value[, map, simultaneous, exact])	Replace matching subexpressions of `self` with `value`.
`rewrite`(*args[, deep])	Rewrite *self* using a defined rule.
`rotate`(angle[, pt])	Rotate `angle` radians counterclockwise about Point `pt`.

continues on next page

Table 6 – continued from previous page

`scale`([x, y, pt])	Scale the object by multiplying the x,y-coordinates by x and y.
`simplify`(**kwargs)	See the simplify function in sympy.simplify
`smallest_angle_between`(l2)	Return the smallest angle formed at the intersection of the lines containing the linear entities.
`sort_key`([order])	Return a sort key.
`subs`(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
`symmetric_difference`(other)	Returns symmetric difference of `self` and `other`.
`translate`([x, y])	Shift the object by adding to the x,y-coordinates the values x and y.
`union`(other)	Returns the union of `self` and `other`.
`xreplace`(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

default_assumptions = {}

class polytex.geometry.basic.**ParamCurve**(*limits*, *function=[]*, *dataset=None*, *krig_config=('lin', 'cub')*, *smooth=0.0*, *verbose=False*)

Bases: `object`

Parameters

limits
[3-tuple] Function parameter and lower and upper bounds.

function
[list] The function list for each coordinate component. The default value is [].

dataset
[array_like] The dataset of the curve. The default value is None. The first column is the parameter and the other columns are the value of coordinate components.

One of the function or dataset must be given. Please note that both are given, the dataset will be ignored.

krig_config
[tuple] The kriging interpolation configuration. The default value is ("lin", "cub"). The tuple should be in the form of (drift_name, covariance_name).

smooth
[float] The smoothing factor. The default value is 0.0.

verbose
[bool] If True, print the information of the kriging process. The default value is False.

Returns

curve
[ParamCurve object]

Methods

eval(t_value)	Evaluate the curve at a given parameter value.

bounds	

bounds()

eval(*t_value*)

Evaluate the curve at a given parameter value. The parameter value should be within the limits. Otherwise, an error will be raised.

Parameters

t_value
[float or array_like] The parameter value for evaluation.

Returns

curve
[Curve object] The evaluated curve.

class polytex.geometry.basic.**ParamCurve3D**(*functions*, *limits*, ***kwargs*)

Bases: Curve

Parameters

function
[list of functions] Function argument should be (x(t), y(t), z(t)) for a 3D curve.

limits
[3-tuple] Function parameter and lower and upper bounds. The parameter should be the same for all three functions. For example, (t, 0, 1) is valid but ((t, 0, 1), (s, 0, 1), (u, 0, 1)) is not.

Returns

curve
[ParamCurve3D object]

Attributes

ambient_dimension
The dimension of the curve.

args
Returns a tuple of arguments of 'self'.

assumptions0
Return object *type* assumptions.

boundary
The boundary or frontier of a set.

bounds
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

canonical_variables
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

closure
Property method which returns the closure of a set.

expr_free_symbols
free_symbols
Return a set of symbols other than the bound symbols used to parametrically define the Curve.

func
The top-level function in an expression.

functions
The functions specifying the curve.

inf
The infimum of `self`.

interior
Property method which returns the interior of a set.

is_Complement
is_EmptySet
is_Intersection
is_UniversalSet
is_algebraic
is_antihermitian
is_closed
A property method to check whether a set is closed.

is_commutative
is_comparable
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_empty
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_finite_set
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_open

Property method to check whether a set is open.

is_polar

is_positive

is_prime

is_rational

is_real

is_transcendental

is_zero

kind

The kind of a Set

length

The length of the curve.

limits

The limits for the curve.

measure

The (Lebesgue) measure of `self`.

parameter

The curve function variable.

sup

The supremum of `self`.

Methods

`__call__`(f)	Call self as a function.
`arbitrary_point`([parameter])	A parameterized point on the curve.
`as_content_primitive`([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
`as_dummy`()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
`atoms`(*types)	Returns the atoms that form the current object.
`class_key`()	Nice order of classes.
`compare`(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
`complement`(universe)	The complement of 'self' w.r.t the given universe.
`contains`(other)	Returns a SymPy value indicating whether `other` is contained in `self`: `true` if it is, `false` if it is not, else an unevaluated `Contains` expression (or, as in the case of ConditionSet and a union of FiniteSet/Intervals, an expression indicating the conditions for containment).
`count`(query)	Count the number of matching subexpressions.
`count_ops`([visual])	Wrapper for count_ops that returns the operation count.

continues on next page

Table 7 – continued from previous page

doit(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.
dummy_eq(other[, symbol])	Compare two expressions and handle dummy symbols.
encloses(o)	Return True if o is inside (not on or outside) the boundaries of self.
eval(t_value)	Evaluate the curve at a given parameter value.
evalf([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
find(query[, group])	Find all subexpressions matching a query.
fromiter(args, **assumptions)	Create a new object from an iterable.
has(*patterns)	Test whether any subexpression matches any of the patterns.
has_free(*patterns)	Return True if self has object(s) `x` as a free expression else False.
has_xfree(s)	Return True if self has any of the patterns in s as a free argument, else False.
intersect(other)	Returns the intersection of 'self' and 'other'.
intersection(o)	Returns a list of all of the intersections of self with o.
is_disjoint(other)	Returns True if `self` and `other` are disjoint.
is_proper_subset(other)	Returns True if `self` is a proper subset of `other`.
is_proper_superset(other)	Returns True if `self` is a proper superset of `other`.
is_similar(other)	Is this geometrical entity similar to another geometrical entity?
is_subset(other)	Returns True if `self` is a subset of `other`.
is_superset(other)	Returns True if `self` is a superset of `other`.
isdisjoint(other)	Alias for `is_disjoint()`
issubset(other)	Alias for `is_subset()`
issuperset(other)	Alias for `is_superset()`
match(pattern[, old])	Pattern matching.
matches(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
n([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
parameter_value(other, t)	Return the parameter corresponding to the given point.
plot_interval([parameter])	The plot interval for the default geometric plot of the curve.
powerset()	Find the Power set of `self`.
rcall(*args)	Apply on the argument recursively through the expression tree.
refine([assumption])	See the refine function in sympy.assumptions
reflect(line)	Reflects an object across a line.
replace(query, value[, map, simultaneous, exact])	Replace matching subexpressions of `self` with `value`.
rewrite(*args[, deep])	Rewrite *self* using a defined rule.
rotate(angle[, axis])	This function is used to rotate a curve along given point `pt` at given angle(in radian).
scale([x, y, z])	Override GeometryEntity.scale since Curve is not made up of Points.
simplify(**kwargs)	See the simplify function in sympy.simplify
sort_key([order])	Return a sort key.

continues on next page

Table 7 – continued from previous page

subs(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
symmetric_difference(other)	Returns symmetric difference of self and other.
translate([x, y, z])	Translate the Curve by (x, y, z).
union(other)	Returns the union of self and other.
xreplace(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
equals	
is_hypergeometric	

eval(*t_value*)

Evaluate the curve at a given parameter value. The parameter value should be within the limits. Otherwise, an error will be raised.

Parameters

t_value
[float or array_like] The parameter value for evaluation.

Returns

curve
[Curve object] The evaluated curve.

rotate(*angle*, *axis='z'*)

This function is used to rotate a curve along given point pt at given angle(in radian).

Parameters

angle
the angle at which the curve will be rotated(in radian) in counterclockwise direction. default value of angle is 0.

pt
[Point] the point along which the curve will be rotated. If no point given, the curve will be rotated around origin.

Returns

Curve
returns a curve rotated at given angle along given point.

Examples

```
>>> from sympy import Curve, pi
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))
```

scale(*x=1*, *y=1*, *z=1*)

Override GeometryEntity.scale since Curve is not made up of Points.

Returns

Curve
returns scaled curve.

Examples

```
>>> from sympy import Curve
>>> from sympy.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))
```

translate(*x=0*, *y=0*, *z=0*)

Translate the Curve by (x, y, z).

Parameters

x
[float] The translation in the x direction.

y
[float] The translation in the y direction.

z
[float] The translation in the z direction.

Returns

Curve
returns a translated curve.

Examples

```
>>> from polytex.geometry import ParamCurve3D
>>> from sympy.abc import x
>>> ParamCurve3D((x, x), (x, 0, 1)).translate(1, 2)
ParamCurve3D((x + 1, x + 2), (x, 0, 1))
```

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

default_assumptions = {}

property length

The length of the curve.

class polytex.geometry.basic.ParamSurface(*functions*, *limits*, ***kwargs*)

Bases: `GeometryEntity`

Parameters

functions
[list of functions] Function argument should be (x(s, t), y(s, t), z(s, t)) for a 3D surface.

limits
[2-tuple] Function parameter and lower and upper bounds of the two parameters. For example, ((s, 0, 1), (t, 0, 1)) is valid. The parameter should be the same for all three functions.

Attributes

args
Returns a tuple of arguments of 'self'.

assumptions0
Return object *type* assumptions.

bounds
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

canonical_variables
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

expr_free_symbols
free_symbols
Return from the atoms of self those which are free symbols.

func
The top-level function in an expression.

functions
The functions specifying the surface.

is_algebraic
is_antihermitian
is_commutative
is_comparable
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero

limits
The limits of the two parameters specifying the surface.

Methods

`as_content_primitive`([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
`as_dummy`()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
`atoms`(*types)	Returns the atoms that form the current object.
`class_key`()	Nice order of classes.
`compare`(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
`count`(query)	Count the number of matching subexpressions.
`count_ops`([visual])	Wrapper for count_ops that returns the operation count.
`doit`(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.

continues on next page

Table 8 – continued from previous page

`dummy_eq`(other[, symbol])	Compare two expressions and handle dummy symbols.
`encloses`(o)	Return True if o is inside (not on or outside) the boundaries of self.
`eval`(s_value, t_value)	Evaluate the surface at the given parameter values.
`evalf`([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
`find`(query[, group])	Find all subexpressions matching a query.
`fromiter`(args, **assumptions)	Create a new object from an iterable.
`has`(*patterns)	Test whether any subexpression matches any of the patterns.
`has_free`(*patterns)	Return True if self has object(s) `x` as a free expression else False.
`has_xfree`(s)	Return True if self has any of the patterns in s as a free argument, else False.
`intersection`(o)	Returns a list of all of the intersections of self with o.
`match`(pattern[, old])	Pattern matching.
`matches`(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
`n`([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
`rcall`(*args)	Apply on the argument recursively through the expression tree.
`refine`([assumption])	See the refine function in sympy.assumptions
`replace`(query, value[, map, simultaneous, exact])	Replace matching subexpressions of `self` with `value`.
`rewrite`(*args[, deep])	Rewrite *self* using a defined rule.
`rotate`(angle[, axis])	Rotate `angle` radians counterclockwise about Point `pt`.
`scale`([x, y, z])	Scale the object by multiplying the x,y-coordinates by x and y.
`simplify`(**kwargs)	See the simplify function in sympy.simplify
`sort_key`([order])	Return a sort key.
`subs`(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
`translate`([x, y, z])	Shift the object by adding to the x,y-coordinates the values x and y.
`xreplace`(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

eval(*s_value*, *t_value*)

Evaluate the surface at the given parameter values.

Parameters

s_value
[float or array_like] The parameter value for evaluation.

t_value
[float or array_like] The parameter value for evaluation.

Returns

Surface
The surface evaluated at the given parameter values. the shape of the returned surface is (len(s) * len(t), 3). The first column is s values, the second column is t values, and the following columns are the coordinates of the surface at the given parameter values (x, y, z).

rotate(*angle*, *axis='z'*)

Rotate `angle` radians counterclockwise about Point `pt`.

The default pt is the origin, Point(0, 0)

See also:

scale, *translate*

Examples

```
>>> from sympy import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2, -1/2))
```

scale(*x=1*, *y=1*, *z=1*)

Scale the object by multiplying the x,y-coordinates by x and y.

If pt is given, the scaling is done relative to that point; the object is shifted by -pt, scaled, and shifted by pt.

See also:

rotate, *translate*

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/2))
>>> t.scale(2, 2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

translate(*x=0*, *y=0*, *z=0*)

Shift the object by adding to the x,y-coordinates the values x and y.

See also:

rotate, *scale*

Examples

```
>>> from sympy import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2), Point2D(3/2, 2 - sqrt(3)/
↪2))
```

default_assumptions = {}

property functions

The functions specifying the surface.

Returns

functions
list of parameterized coordinate functions.

Examples

```
>>> from sympy.abc import t
>>> from polytex.geometry import ParamCurve3D
>>> surface = ParamSurface((t, t, t), ((t, 0, 1), (t, 0, 1)))
>>> surface.functions
(t, t, t)
```

property limits

The limits of the two parameters specifying the surface.

Returns

list limits of the parameters.

Examples

```
>>> from sympy.abc import t
>>> from polytex.geometry import ParamCurve3D
>>> surface = ParamSurface((t, t, t), ((t, 0, 1), (t, 0, 1)))
>>> surface.limits
((t, 0, 1), (t, 0, 1))
```

class polytex.geometry.basic.**Plane**(*p1*, *a=None*, *b=None*, ***kwargs*)

Bases: `object`

Create a plane from 3 points or a point and a normal vector.

Parameters

p1
[Point object] A point on the plane.

a
[Point object, optional] A point on the plane. The default is None.

b
[Point object, optional] A point on the plane. The default is None.

****kwargs**
[dict, optional] *normal_vector* can be passed as a keyword argument when leaving a and b as None. If both a and b are not None, normal_vector will be ignored.

Returns

plane
[Plane object]

Methods

distance(point)	Return the signed distance between a point and the plane.
function()	Return the equation of the plane as a function of x, y and z.
intersection(obj, max_dist)	Return the intersection of the plane with a curve or a polygon.
show()	Plot the plane.

distance(*point*)

Return the signed distance between a point and the plane.

Parameters

point
[Point object] The point to calculate the distance. shape = (n, 3), where n is the number of points.

Returns

distance
[float] The distance between the point and the plane.

Examples

```
>>> from polytex.geometry import Point, Plane
>>> p1 = Point([5, 0, 0])
>>> normal = Point([1, 0, 0])
>>> plane = Plane(p1, normal_vector=normal)
>>> plane.show()
>>> plane.distance([[0, 0, 0], [1, 0, 0], [2, 0, 0]])
array([-5., -4., -3.])
```

function()

Return the equation of the plane as a function of x, y and z.

Parameters

normal
[array-like] normal vector of the plane.

point
[array-like] point on the plane.

Returns

A lambda function of x, y and z.
function of plane.

Notes

normal = (a, b, c) point = (x0, y0, z0) Equation of a plane: a(x-x0) + b(y-y0) + c(z-z0) = 0

Examples

Create a plane from a point and a normal vector >>> from polytex.geometry import Point, Plane >>> p1 = Point([1, 1, 1]) >>> normal = Point([1, 4, 7]) >>> plane2 = Plane(p1, normal_vector=normal) >>> f = plane2.function() >>> f <function polytex.geometry.basic.Plane.function.<locals>.<lambda>(x, y, z)> >>> f(1, 1, 1) 0

`intersection`(*obj*, *max_dist*)

Return the intersection of the plane with a curve or a polygon.

Parameters

obj
[Curve or Polygon object] The object to intersect with the plane.

max_dist
[float] The maximum distance of checked points from the plane.

Returns

intersection
[list or array] The intersection point of the plane with the obj in the shape of (1, 3). If the intersection is not found, none is returned.

`show`()

Plot the plane.

Returns

None

`class polytex.geometry.basic.Point`(*orig_3d=(0, 0, 0)*)

Bases: `ndarray`

Default constructor. If no arguments are given, the point is initialized to (0, 0, 0).

Parameters

cls
[class] The class of the object.

orig_3d
[tuple, list, or array_like] Defaults to 3d origin (0, 0, 0).

Returns

obj
[Point] The origin point of 3d space.

Examples

```
>>> p1 = Point()
>>> p1
Point([0, 0, 0])
```

Attributes

T
View of the transposed array.

base
Base object if memory is from some other object.

bounds
Returns ——- bounds : array_like The bounding box of the point.

ctypes
An object to simplify the interaction of the array with the ctypes module.

data
Python buffer object pointing to the start of the array's data.

dtype
Data-type of the array's elements.

flags
Information about the memory layout of the array.

flat
A 1-D iterator over the array.

imag
The imaginary part of the array.

itemsize
Length of one array element in bytes.

nbytes
Total bytes consumed by the elements of the array.

ndim
Number of array dimensions.

real
The real part of the array.

shape
Tuple of array dimensions.

size
Number of elements in the array.

strides
Tuple of bytes to step in each dimension when traversing an array.

x

xyz

y

z
Return —— z : float 3rd dimension element.

Methods

`all`([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
`any`([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
`argmax`([axis, out, keepdims])	Return indices of the maximum values along the given axis.
`argmin`([axis, out, keepdims])	Return indices of the minimum values along the given axis.
`argpartition`(kth[, axis, kind, order])	Returns the indices that would partition this array.
`argsort`([axis, kind, order])	Returns the indices that would sort this array.
`astype`(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
`byteswap`([inplace])	Swap the bytes of the array elements
`choose`(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
`clip`([min, max, out])	Return an array whose values are limited to `[min, max]`.
`compress`(condition[, axis, out])	Return selected slices of this array along given axis.
`conj`()	Complex-conjugate all elements.
`conjugate`()	Return the complex conjugate, element-wise.
`copy`([order])	Return a copy of the array.
`cumprod`([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
`cumsum`([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
`diagonal`([offset, axis1, axis2])	Return specified diagonals.
`direction_ratio`(other)	Gives the direction ratio between 2 points.
`dist`(other)	Both points must have the same dimensions :return: Euclidean distance
`dump`(file)	Dump a pickle of the array to the specified file.
`dumps`()	Returns the pickle of the array as a string.
`fill`(value)	Fill the array with a scalar value.
`flatten`([order])	Return a copy of the array collapsed into one dimension.
`getfield`(dtype[, offset])	Returns a field of the given array as a certain type.
`item`(*args)	Copy an element of an array to a standard Python scalar and return it.
`itemset`(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
`max`([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
`mean`([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
`min`([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
`newbyteorder`([new_order])	Return the array with the same data viewed with a different byte order.
`nonzero`()	Return the indices of the elements that are non-zero.
`partition`(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
`prod`([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis

continues on next page

Table 9 – continued from previous page

ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set a.flat[n] = values[n] for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save_as_vtk(filename[, color])	Save the point as a vtk file.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITE-BACKIFCOPY, respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an a.ndim-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.

dot	
set_x	
set_y	
set_z	

direction_ratio(*other*)

Gives the direction ratio between 2 points.

Parameters

other
[Point object] The other point to which the direction ratio is calculated.

Returns

direction_ratio
[list] The direction ratio between the 2 points.

Examples

```
>>> from polytex.geometry import Point
>>> p1 = Point(1, 2, 3)
>>> p1.direction_ratio(Point(2, 3, 5))
[1, 1, 2]
```

dist(*other*)

Both points must have the same dimensions :return: Euclidean distance

save_as_vtk(*filename*, *color=None*)

Save the point as a vtk file.

set_x(*x*)

set_y(*y*)

set_z(*z*)

property bounds

Returns

bounds
[array_like] The bounding box of the point. The first row is the minimum values and the second row is the maximum values for each dimension.

property size

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

Notes

a.size returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

property x

property xyz

`property y`

`property z`

`class polytex.geometry.basic.Polygon(points)`

Bases: `Curve`

A partial inheritance of polytex.geometry.Point class.

Parameters

points
[list, tuple or array_like] A list of Point objects.

Attributes

`ambient_dimension`
Return the dimension of the curve.

`area`
Return the area of the polygon.

`bounds`
Return the bounds of the curve.

`centroid`
Return the centroid of the polygon.

`curvature`
Return the curvature of the curve.

`length`
Return the length of the curve.

`perimeter`
Return the perimeter of the polygon.

`tangent`
Return the tangent vector of the curve at each point.

Methods

`plot`()	Plot the curve.
`save`([save_path])	Save the curve to a vtk file.
`to_curve`()	Convert the polygon to a curve.
`to_polygon`()	Convert the curve to a polygon.

`to_curve()`

Convert the polygon to a curve.

Returns

curve
[Curve object]

`property area`

Return the area of the polygon.

Returns

area
[float]

property centroid
Return the centroid of the polygon.

Returns

centroid
[Point object]

property perimeter
Return the perimeter of the polygon.

Returns

perimeter
[float]

class polytex.geometry.basic.Tube(*theta_res*, *h_res*, *vertices=None*, ***kwargs*)

Bases: `GeometryEntity`

Create a tubular surface.

Parameters

theta_res
[int] The number of points on each cross-section.

h_res
[int] The number of cross-sections.

vertices
[array_like] The points on the cross-sections. The shape of the array should be (h_res * theta_res, 3). The points should be ordered in the following way: [p1, p2, …, p_theta_res, p1, p2, …, p_theta_res, …, p1, p2, …, p_theta_res] where p1, p2, …, p_theta_res are the points on each cross-section from the top to the bottom. The default value is None. If the value is None, the points will be generated automatically by assigning the height, major and minor radius to the tube.

Returns

tube
[Tube object]

Examples

```
>>> from polytex.geometry import Tube
>>> tube = Tube(4, 10, major=2, minor=1, h=5)
>>> mesh = tube.mesh(plot=True)
>>> tube.save_as_mesh('tube.vtk')
```

Attributes

args
Returns a tuple of arguments of 'self'.

assumptions0
Return object *type* assumptions.

bounds
Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

`canonical_variables`
Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any free symbols in the expression.

expr_free_symbols
`free_symbols`
Return from the atoms of self those which are free symbols.

`func`
The top-level function in an expression.

h_res
is_algebraic
is_antihermitian
is_commutative
`is_comparable`
Return True if self can be computed to a real number (or already is a real number) with precision, else False.

is_complex
is_composite
is_even
is_extended_negative
is_extended_nonnegative
is_extended_nonpositive
is_extended_nonzero
is_extended_positive
is_extended_real
is_finite
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero
points
theta_res

Methods

as_content_primitive([radical, clear])	A stub to allow Basic args (like Tuple) to be skipped when computing the content and primitive components of an expression.
as_dummy()	Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.
atoms(*types)	Returns the atoms that form the current object.
class_key()	Nice order of classes.
compare(other)	Return -1, 0, 1 if the object is smaller, equal, or greater than other.
count(query)	Count the number of matching subexpressions.
count_ops([visual])	Wrapper for count_ops that returns the operation count.
doit(**hints)	Evaluate objects that are not evaluated by default like limits, integrals, sums and products.
dummy_eq(other[, symbol])	Compare two expressions and handle dummy symbols.
encloses(o)	Return True if o is inside (not on or outside) the boundaries of self.
evalf([n, subs, maxn, chop, strict, quad, ...])	Evaluate the given formula to an accuracy of *n* digits.
find(query[, group])	Find all subexpressions matching a query.
fromiter(args, **assumptions)	Create a new object from an iterable.
has(*patterns)	Test whether any subexpression matches any of the patterns.
has_free(*patterns)	Return True if self has object(s) x as a free expression else False.
has_xfree(s)	Return True if self has any of the patterns in s as a free argument, else False.
intersection(o)	Returns a list of all of the intersections of self with o.
match(pattern[, old])	Pattern matching.
matches(expr[, repl_dict, old])	Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.
mesh([plot, show_edges])	TODO : raise TypeError("Given points must be a sequence or an array.")
n([n, subs, maxn, chop, strict, quad, verbose])	Evaluate the given formula to an accuracy of *n* digits.
rcall(*args)	Apply on the argument recursively through the expression tree.
refine([assumption])	See the refine function in sympy.assumptions
replace(query, value[, map, simultaneous, exact])	Replace matching subexpressions of `self` with `value`.
rewrite(*args[, deep])	Rewrite *self* using a defined rule.
rotate(angle[, pt])	Rotate `angle` radians counterclockwise about Point `pt`.
save_as_mesh(save_path[, end_closed])	Save the tubular mesh to a file.
scale([x, y, pt])	Scale the object by multiplying the x,y-coordinates by x and y.
simplify(**kwargs)	See the simplify function in sympy.simplify

continues on next page

Table 10 – continued from previous page

sort_key([order])	Return a sort key.
subs(*args, **kwargs)	Substitutes old for new in an expression after sympifying args.
translate([x, y])	Shift the object by adding to the x,y-coordinates the values x and y.
xreplace(rule)	Replace occurrences of objects within the expression.

copy	
could_extract_minus_sign	
is_hypergeometric	

mesh(*plot=False*, *show_edges=True*)

TODO : raise TypeError("Given points must be a sequence or an array.")

Notes

theta_res should be 1 less then else where. To be fixed in the future.

save_as_mesh(*save_path*, *end_closed=True*)

Save the tubular mesh to a file. The file format is determined by the extension of the filename. The possible file formats are: [".ply", ".stl", ".vtk", ".vtu"].

TODO : There seems to be a bug in correction option of the to_meshio_data() method of the tubular mesh.

Parameters

save_path
[str] The path and the name of the file to be saved with the extension.

end_closed
[bool] If True, the ends of the tube will be closed. The default value is True.

Returns

mesh
[pyvista.UnstructuredGrid] The tubular mesh.

Examples

```
>>> from polytex.geometry import Tube
>>> tube = Tube(5,10,major=2, minor=1,h=5)
>>> tube.save_as_mesh('tube.vtu')
```

property bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

default_assumptions = {}

property h_res: int

property points

property theta_res: int

class polytex.geometry.basic.**Vector**(*orig_3d=(0, 0, 0)*)

Bases: *Point*

Default constructor. If no arguments are given, the point is initialized to (0, 0, 0).

Parameters

cls
[class] The class of the object.

orig_3d
[tuple, list, or array_like] Defaults to 3d origin (0, 0, 0).

Returns

obj
[Point] The origin point of 3d space.

Examples

```
>>> p1 = Point()
>>> p1
Point([0, 0, 0])
```

Attributes

T
View of the transposed array.

add

base
Base object if memory is from some other object.

bounds
Returns ——- bounds : array_like The bounding box of the point.

ctypes
An object to simplify the interaction of the array with the ctypes module.

data
Python buffer object pointing to the start of the array's data.

dtype
Data-type of the array's elements.

flags
Information about the memory layout of the array.

flat
A 1-D iterator over the array.

imag
The imaginary part of the array.

itemsize
Length of one array element in bytes.

nbytes
Total bytes consumed by the elements of the array.

ndim
Number of array dimensions.

norm
Return —— norm : float The norm of the vector.

real
The real part of the array.

shape
Tuple of array dimensions.

size
Number of elements in the array.

strides
Tuple of bytes to step in each dimension when traversing an array.

sub

x

xyz

y

z
Return —— z : float 3rd dimension element.

Methods

all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
angle_between(other[, radian])	Return the angle between 2 vectors.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out, keepdims])	Return indices of the maximum values along the given axis.
argmin([axis, out, keepdims])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to `[min, max]`.
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
diagonal([offset, axis1, axis2])	Return specified diagonals.
direction_ratio(other)	Gives the direction ratio between 2 points.

continues on next page

Table 11 – continued from previous page

dist(other)	Both points must have the same dimensions :return: Euclidean distance
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
save_as_vtk(filename[, color])	Save the point as a vtk file.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.

continues on next page

Table 11 – continued from previous page

`tobytes`([order])	Construct Python bytes containing the raw data bytes in the array.
`tofile`(fid[, sep, format])	Write array to a file as text or binary (default).
`tolist`()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
`tostring`([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
`trace`([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
`transpose`(*axes)	Returns a view of the array with axes transposed.
`var`([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
`view`([dtype][, type])	New view of array with the same data.

cross	
dot	
set_x	
set_y	
set_z	

`angle_between`(*other*, *radian=False*)

Return the angle between 2 vectors.

$$\theta = \arccos \frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|}$$

Parameters

other
[Vector object] The other vector to which the angle is calculated.

radian
[bool, optional] If True, return the angle in radians. The default is False.

Returns

angle
[float] The angle between the 2 vectors. If radian is True, the angle is in radians. Otherwise, the angle is in degrees.

`cross`(*other*)

`dot`(*other*)

`property add`

`property norm`

`property sub`

`polytex.geometry.basic.find_intersect`(*f*, *curve*, *niterations=5*, *mSegments=5*)

Find the intersection of a curve with a plane

Parameters

f
[lambda function] function of plane.

curve
[array-like] points on the curve in shape of (n, 3).

niterations: int
number of iterations.

mSegments: int
number of segments for each iteration.

Returns

intersection: array-like
intersection points with shape (n, 3).

polytex.geometry.geometry

`polytex.geometry.geometry.`**`angularSort`**(*localCo*, *centroid*, *sort=True*)

Sort the vertices of a 2D polygon in angular order. It can be a convex or concave polygon.

Parameters

localCo
[Numpy array] with 2 columns. The x, y coordinate components of the vertices of the polygon (For the cross-section of fiber tows, it is the coordinate in the local coordinate system with its center at the centroid of the polygon).

centroid
[Numpy array] with 2 columns. The x, y coordinate components of the centroid of the polygon.

sort
[Boolean] If True, the vertices are sorted in angular order. If False, the vertices are not sorted and returned following the original order with angular position for each input vertices.

Returns

coorSort
[Numpy array] with 3 columns. The x, y coordinate components of the vertices of the polygon sorted in angular order. The third column is the z coordinate in 3D case.

angle
[Numpy array] with 1 column. The angular position of the vertices of the polygon in degrees. The two returns are sorted in the same order if sort is True. Otherwise, the two returns are not sorted, and are following the original order of the input vertices.

`polytex.geometry.geometry.`**`area_signed`**(*points: ndarray*) → float

Return the signed area of a simple polygon given the 2D coordinates of its veritces.

The signed area is computed using the shoelace algorithm. A positive area is returned for a polygon whose vertices are given by a counter-clockwise sequence of points.

Parameters

points
[array_like] Input 2D points of the polygon in the shape (n, 2). If the polygon is 3D, An error is raised. Note that the points have to be ordered in a clockwise or counter-clockwise manner. Otherwise, the area will be non-sense.

Returns

area_signed
[float] The signed area of the polygon.

Raises

ValueError
If the points are not 2D.

Notes

- If the number of points is less than 3, a warning is raised and the area is

returned as 0. - This function is modified from open source Python library scikit-spatial: skspatial.measurement — scikit-spatial documentation https://scikit-spatial.readthedocs.io/en/stable/_modules/skspatial/measurement.html#area_signed)

Examples

```
>>> from polytex.geometry import area_signed
```

```
>>> area_signed([[0, 0], [1, 0], [0, 1]])
0.5
```

```
>>> area_signed([[0, 0], [0, 1], [1, 0]])
-0.5
```

```
>>> area_signed([[0, 0], [0, 1], [1, 2], [2, 1], [2, 0]])
-3.0
```

`polytex.geometry.geometry.`**`edgeLen`**(*localCo*, *boundType='rotated'*)

the width and height of rotated_rectangle

Parameters

localCo
[Numpy array] with 2 columns. The x, y coordinate components of the vertices of the polygon (For the cross-section of fiber tows, it is the coordinate in the local coordinate system with its center at the centroid of the polygon).

boundType: string
rotated or parallel

Returns

width
[float] The width of the minimum rotated rectangle that contains the polygon.

height
[float] The height of the minimum rotated rectangle that contains the polygon.

angleRotated
[float] The angle of rotation of the minimum rotated rectangle that contains the polygon.

`polytex.geometry.geometry.`**`geom_cs`**(*coordinate*, *message='OFF'*, *sort=True*)

Geometry analysis and points sorting for a cross-section of a fiber tow.

Parameters

coordinate
[Numpy array] with 3 colums. The x, y, z coordinate components of the vertices of the polygon. Note that only the first two columns are used for the 2D polygon.

Returns

geometry file
[x,y,z of points, and x,y,z of centerline]

properties: area… …

`polytex.geometry.geometry.geom_tow(surf_points, sort=True)`

The surface points for each cross-section. the last column (z-axis) should be along the extension direction of the cross-sections. It also serves as the label of each cross-section.

Parameters

surf_points
[array_like] The surface points for each cross-section. the last column (z-axis) should be along the extension direction of the cross-sections. It also serves as the label of each cross-section.

sort
[Boolean] If True, the vertices are sorted in angular order. If False, the vertices are not sorted and returned following the original order with angular position for each input vertices.

Returns

df_geom
[DataFrame] The geometrical features of each cross-section. The columns are: [Area, Perimeter, Width, Height, AngleRotated, Circularity, centroidX, centroidY, centroidZ]

df_coo
[DataFrame] The coordinates of each cross-section. The columns are: [distance, normalized distance, angular position (degree), X, Y, Z)]

`polytex.geometry.geometry.normDist(localCo)`

The normalized distance of the vertices of a polygon

Parameters

localCo
[Numpy array] with 2 columns. The x, y coordinate components of the vertices of the polygon (For the cross-section of fiber tows, it is the coordinate in the local coordinate system with its center at the centroid of the polygon).

polytex.geometry.transform

`polytex.geometry.transform.d2r(degrees: float) → float`

Convert degrees to radians.

Parameters

degrees
[float] Angle in degrees.

Returns

float
Angle in radians.

`polytex.geometry.transform.e123_dcm(psi: float, theta: float, phi: float) → ndarray`

This function chaining the rotation matrices for the Euler 123 sequence. The rotation matrix is defined as:

$$R = R_3(psi)R_2(theta)R_1(phi)$$

where R_1 is the rotation matrix about the x-axis, R_2 is the rotation matrix about the y-axis, and R_3 is the rotation matrix about the z-axis.

Parameters

psi
[float] The rotation angle about the z-axis in radians.

theta
[float] The rotation angle about the y-axis in radians.

phi
[float] The rotation angle about the x-axis in radians.

Returns

dcm
[numpy.ndarray] The direction cosine matrix for the Euler 123 sequence.

Examples

```
>>> import polytex.geometry.transform as tf
```

`polytex.geometry.transform.euler_z_noraml(normal, *args) → list`

This function returns the euler angles (phi, theta, psi) for rotating the global coordinate system to align its z-axis with a normal vector from the origin to a point (namely, no translation is considered).

Parameters

normal
[list or array] The normal vector from the origin to a point.

Returns

euler_angles
[list] The euler angles (psi, theta, phi), where psi is the rotation angle about the z-axis, theta is the rotation angle about the y-axis, and phi is the rotation angle about the x-axis in radians. Note that the rotation should be performed in the order of e123 by pre-multiplying the rotation matrices.

Notes

No translation is considered. The origin of the global coordinate system is assumed to be the origin of the local coordinate system. The user should translate the local coordinate system to the origin before calling this function and then re-translate the local coordinate system to the desired location.

Examples

```
>>> import polytex.geometry.transform as tf
>>> import numpy as np
>>> normal = [0.43583834, -0.00777955, -0.89999134]
>>> euler_angles = tf.euler_z_noraml(normal)
>>> print(np.allclose(euler_angles, [0, 0.4509695318910846, 3.132948841252596]))
True
>>> print("As we are rotating the global coordinate system to align its z-axis with␣
↪the normal vector,")
>>> print("the normal vector should be [0, 0, 1] after the rotation.")
>>> tf.e123_dcm(*euler_angles) @ normal
>>> print(np.allclose(tf.e123_dcm(*euler_angles) @ normal, [0, 0, 1]))
True
```

`polytex.geometry.transform.`**`euler_zx_coordinate`**(*z_new*, *x_new*) → list

This function returns the euler angles (phi, theta, psi) for rotating the global coordinate system to align its z-axis with the z_new vector and its x-axis with the x_new vector.

Parameters

z_new
[list or array] The coordinate of the new z-axis in the original coordinate system.

x_new
[list or array] The coordinate of the new x-axis in the original coordinate system.

Returns

euler_angles
[list] The euler angles (psi, theta, phi), where psi is the rotation angle about the z-axis, theta is the rotation angle about the y-axis, and phi is the rotation angle about the x-axis in radians. Note that the rotation should be done in e123 sequence by pre-multiplying the rotation matrices.

Notes

No translation is considered. The origin of the global coordinate system is assumed to be the origin of the local coordinate system. The user should translate the local coordinate system to the origin before calling this function and then re-translate the local coordinate system to the desired location.

Examples

```
>>> import polytex.geometry.transform as tf
```

`polytex.geometry.transform.`**`rx`**(*phi: float*) → ndarray

Single axis frame rotation about the X-axis.

Parameters

phi
[float] The angle between z-axis (or y-axis) of the initial and final frames in radian. The rotation is positive if the frame rotates in the counter-clockwise direction when viewed from the positive end of x-axis.

Returns

numpy.ndarray
Rotation matrix.

`polytex.geometry.transform.ry(theta: float)` → ndarray

Single axis frame rotation about the Y-axis.

Parameters

theta
[float] The angle between z-axis (or x-axis) of the initial and final frames in radian. The rotation is positive if the frame rotates in the counter-clockwise direction when viewed from the positive end of y-axis.

`polytex.geometry.transform.rz(psi: float)` → ndarray

Single axis frame rotation about the Z-axis.

Parameters

psi
[float] The angle between x-axis (or y-axis) of the initial and final frames in radian. The rotation is positive if the frame rotates in the counter-clockwise direction when viewed from the positive end of z axis.

Returns

numpy.ndarray
Rotation matrix.

2.2.3 polytex.kriging

polytex.kriging.curve2D

Implementation of 2D Curve Kriging.

Bin Yang 2021-9-2

`polytex.kriging.curve2D.addPoints(coordinate, threshold=0.03)`

Linearly interpolate the points between each of two points that further than the threshold.

Parameters

coordinate
[numpy array] The coordinates of the points. [normalized distance, X, Y, Z]

threshold
[float, optional] The distance between two points. The default is 0.03.

Returns

coordinate
[numpy array] The coordinates of the points after linear interpolation. [normalized distance, X, Y, Z]

`polytex.kriging.curve2D.bd_Deriv_kriging_func(x, y, xDeriv, yDeriv, choixDerive, choixCov, plot_x_pts, nugg)`

Derivative kriging function.

Parameters

x
[array] x points

y
[array] y points

xDeriv
[array] x points for derivative

yDeriv
[array] the derivative of xDeriv points

choixDerive
[string] the name of the derivative function. Possible values are 'cst', 'lin' or 'quad'.

choixCov
[string] the name of the covariance function. Possible values are 'lin' or 'cub'.

plot_x_pts: array
number of points for plot

nugg: float
nugget effect (variance)

Returns

kringFunctionStr: string
String of the kriging function.

x_var_sym: string
string of the x variable

`polytex.kriging.curve2D.`**`buildKrigFunc_deriv`**(*x*, *xKnown*, *xKnown_deriv*, *B*, *deriveFuncs*, *covFuncs*, *covFuncs_deriv*)

`polytex.kriging.curve2D.`**`buildM_deriv`**(*x*, *x_deriv*, *name_drift*, *name_cov*, *covFuncs_deriv*, *covFuncs_deriv2*, *nugg*)

Build the matrix M for the derivative kriging system

Parameters

x
[array] x points

xDeriv
[array] x points for derivative

name_drift
[function] derivative functions

name_cov
[function] covariance functions

covFuncs_deriv
[function] derivative of covariance functions

covFuncs_deriv2
[function] second derivative of covariance functions

nugg
[float] nugget effect (variance)

`polytex.kriging.curve2D.`**`buildU_deriv`**(*y*, *y_deriv*, *deriveFuncs*) `polytex.kriging.curve2D.`**`curve_krig_2D`**`(`*dataset*, *name_drift*, *name_cov*, *nugget_effect=0*`)`

Parameters

dataset: numpy array
X-Y.

name_drift
[String] Name of drift.

name_cov
[String] Name of covariance.

nugget_effect
[Float]

Returns

expr
[Expression] The kriging expression.

`polytex.kriging.curve2D.`**`func_select`**`(`*drift_name*, *cov_name*`)`

This is the function for definition of drift function and covariance function in dictionary drif_funcs and cov_funcs.

Parameters

drift_name
[string] The name of the drift function. Possible values are: “const”, “lin”, and “quad” for “constant”, “linear”, “quadratic”, respectively.

cov_name
[string] The name of the covariance function. Possible values are: “lin”, “cub”, and “log” for “linear”, “cubic”, “logarithmic”, respectively.

Returns

drift_func
[function] The drift function.

cov_func
[function] The covariance function.

`polytex.kriging.curve2D.`**`func_var`**`(`*X*, *K_h*, *lambda_*, *nugget_effect=0*`)`

Calculate the variance of the prediction.

Parameters

X
[numpy array] The locations where function values are unknown and to be predicted.

K_h
[numpy array] The kriging matrix.

lambda_
[numpy array] The weight of gloabl combination. Note that it is different from the vector_ba in dual Kriging formulation.

nugget_effect
[float] The nugget effect. Default is 0.

Returns

std_prediction
[numpy array] the standard deviation of the prediction

`polytex.kriging.curve2D.h(x1, x2)`

The function h(x1, x2) = abs(x1 - x2).

Parameters

x1
[float] The first point.

x2
[float] The second point.

Returns

h
[float]

`polytex.kriging.curve2D.interpolate(dataset, name_drift, name_cov, nugget_effect=0, interp=' ', return_std=False)`

Parameters

dataset
[numpy array] X-Y.

name_drift
[String] Name of drift.

name_cov
[String] Name of covariance.

nugget_effect
[Float] smoothing strength control

interp: Numpy array
The points that need to be interpolated, 1D numpy array. If interp is not given, the x-coordinate of the sample points is used.

Returns

expr
[Expression] The kriging expression.

`polytex.kriging.curve2D.krig_expression(len_b, func_drift, func_cov, adef, dataset, vector_ba)`

return the Kriging function expression.

Parameters

len_b
[int] The length of the b vector (the coefficient of linear combination).

func_drift
[function] The drift function.

func_cov
[function] The covariance function.

adef
[numpy array] The coefficients of the drift function.

dataset
[numpy array] The sample points. X-Y.

vector_ba
[numpy array] The kriging vector.

Returns

expr
[sympy expression] The analytical expression of the function created by kriging.

`polytex.kriging.curve2D.lambda_weight`(*X*, *X_train*, *func_drift*, *func_cov*, *mat_krig*)

Calculate the weight of the lambda matrix

Parameters

X
[numpy array] The locations where function values are unknown and to be predicted.

X_train
[numpy array] The training data. The locations where function values are known.

func_drift
[function] The drift function.

func_cov
[function] The covariance function.

mat_krig
[numpy array] The kriging matrix.

Returns

K_h
[numpy array] The kriging matrix.

lambda_
[numpy array] The weight of gloabl combination. Note that it is different from the vector_ba in dual Kriging formulation.

`polytex.kriging.curve2D.solve`(*dataset*, *krig_len*, *mat_krig*, *inverse_type='inverse'*)

Solve the kriging equation: [Matrix_kriging] [b_a] = [u.. 0..]

dataset: numpy array.
The sample points. X-Y.

krig_len: int.
The length of the kriging vector.

mat_krig: numpy array.
The kriging matrix.

inverse_type: String.
The type of the inverse matrix. "inverse" or "pseudoinverse". "inverse" : inverse matrix "pseudoinverse" : generalized inverse/pseudoinverse

Returns

b_a: numpy array.
The kriging vector.

mat_krig_inv: numpy array.
The inverse matrix of the kriging matrix.

`polytex.kriging.curve2D.solveB`(*M*, *U*)

Solve the linear equation system M * B = U TODO: check the comments

Parameters

M
[numpy array] The kriging matrix.

U
[numpy array] The vector of the right hand side of the linear equation system.

Returns

B
[numpy array] The solution of the linear equation system.

polytex.kriging.intersect

1. the intersection of a curve with a plane def curve_plane()
2. the intersection of a plane with a plane def plane_plane()
3. the intersection of a line with a surface def line_surf()
4. the intersection of a curve with a surface def curve_surf()
5. the intersection of two surfaces def surf_surf()

`polytex.kriging.intersect.`**`ray_plane_intersect`**`(`*plane_normal*, *plane_point*, *ray_direction*, *ray_point*, *epsilon=1e-06*`)`

Find the intersection between a plane and a ray.

Parameters

plane_normal
[array-like] normal vector of the plane for defining the plane.

plane_point
[array-like] point on the plane for defining the plane.

ray_direction
[array-like] direction of the ray.

ray_point
[array-like] The endpoint of the ray.

epsilon
[float] tolerance for determining if an intersection point exists.

Returns

Psi: array-like
intersection point.

polytex.kriging.mdKrig

`polytex.kriging.mdKrig.`**`buildKriging`**`(`*xy*, *z*, *drift_name*, *cov_name*, *nugg=0*`)`

Build the kriging model and return the expression in string format.

Parameters

xy: array like. The coordinates of the points. The shape is (m, 2).
z: array like. The values of the target function. The shape is (m,).
drift_name: str. The name of the drift function.
The possible values are: 'const', 'lin', 'cub'.

cov_name: str. The name of the covariance function.
The possible values are: 'lin', 'cub', 'log'.

nugg: float. The nugget effect (variance).

Returns

return
The expression of kriging function in string format. ..

`polytex.kriging.mdKrig.buildM(xy, drift_name, cov_name)`

Build the kriging matrix.

Parameters

xy: The coordinates of the points. The shape is (m, 2).
drift_name: str. The name of the drift function.
Possible values are: "const", "lin", "quad".

cov_name: str. The name of the covariance function.
Possible values are: "lin", "cub", "log".

Returns

drift_func: The drift function.
cov_func: The covariance function.
a_len: The length of the drift function.
M: The matrix of the kriging system. The shape is (n,n).

`polytex.kriging.mdKrig.buildU(z, a_len)`

Build the result vector of the kriging linear system.

Parameters

z:
The values of the target function. The shape is (m,).

a_len:
The length of the drift function.

Returns

U: The result vector of the kriging linear system. The shape is (n,).

`polytex.kriging.mdKrig.dist(xy, type='Euclidean')`

Calculate the distance between each pair of points.

Parameters

xy: numpy array. The coordinates of the points. The shape is (m, 2).
type: str. The type of the distance. The default is "Euclidean".

Other possible values are:
"1-norm" : The 1-norm distance. "inf-norm" : The infinity-norm distance.

Returns

distance
[numpy array] The distance between each pair of points. The shape is (m, m).

`polytex.kriging.mdKrig.func_select(drift_name, cov_name)`

Function for definition of drift and covariance function in dictionary drif_funcs and cov_funcs.

Parameters

drift_name: str. The name of the drift function.
Possible values are: "const", "lin", "quad".

cov_name: str. The name of the covariance function.
Possible values are: "lin", "cub", "log".

Returns

drift_func: Function.
The drift function.

cov_func: Function.
The covariance function.

a_len: int.
The length of the drift function.

`polytex.kriging.mdKrig.`**`interp`**`(`*xy*, *expr*`)`

TODO: add description

Parameters

xy: numpy array.
The coordinates of the points. The shape is (m, 2).

expr: String.
The expression of the target function.

Returns

yinter: The values of the kriging function. The shape is (m,).

`polytex.kriging.mdKrig.`**`nugget`**`(`*M*, *nugg*, *b_len*`)`

Introduce the nugget effect to the kriging matrix.

Parameters

M
[numpy array.] The kriging matrix. The shape is (n,n).

nugg
[float.] The nugget effect.

Returns

M: numpy array.
The kriging matrix with nugget effect.

`polytex.kriging.mdKrig.`**`solveB`**`(`*M*, *U*`)`

Solve the kriging linear system.

Parameters

M: numpy array.
The kriging matrix.

U: numpy array.
The result vector of the kriging linear system.

Returns

B: numpy array.
The solution of the kriging linear system (vector contains b_i and a_i).

polytex.kriging.paraSurface

`polytex.kriging.paraSurface.buildKriging`(*s*, *t*, *x*, *drift_names*, *cov_names*, *nugg=[ ]*)

Build the kriging model and return the expression in string format.

Parameters

s, t: numpy array.
The parameters of the two profiles for surface parametric kriging in S and T direction (1d array).

x: array like.
The known values of the variables in parametric space.

drift_names: list.
The name of the drift functions for profile 1 and profile 2 in the following format: [drift_name1, drift_name2]. The possible values are: 'const', 'lin', 'cub'.

cov_names: list.
The name of the covariance functions in the following format: [covariance_name1, covariance_name2]. The possible values are: 'lin', 'cub', 'log'.

nugg: list.
The nugget effects (variance) for each profile contained in a list.

Returns

expr: The expression of kriging function in string format.

`polytex.kriging.paraSurface.buildM`(*x*, *drift_name*, *cov_name*)

Build the kriging matrix.

Parameters

x: The coordinates of the points. The shape is (m, 2).
drift_name: str. The name of the drift function.
Possible values are: "const", "lin", "quad".

cov_name: str. The name of the covariance function.
Possible values are: "lin", "cub", "log".

Returns

return drift_func
The drift function. ..

return cov_func
The covariance function. ..

return a_len
The length of the drift function. ..

return M
The matrix of the kriging system. The shape is (n,n). ..

`polytex.kriging.paraSurface.buildP`(*x*, *a_lenS*, *a_lenT*)

Build the result matrix of the kriging linear system.

Parameters

x: numpy array.
The values of the target function. The shape is (m,n).

a_lenS: int.
The size of the result matrix in S direction.

a_lenT: int.
The length of the result matrix in T direction.

Returns

P

[numpy array.] The result vector of the kriging linear system. The shape is (n,).

`polytex.kriging.paraSurface.dist1D(x)`

Calculate the distance between each pair of points.

Parameters

x: numpy array.
The coordinates in parametric space of the points. The shape is (m, 1).

Returns

numpy array.
The distance between each pair of points. The shape is (m, m).

`polytex.kriging.paraSurface.func_select(drift_name, cov_name)`

This is the function for definition of drift function and covariance function in dictionary drif_funcs and cov_funcs.

Parameters

drift_name: str.
The name of the drift function. Possible values are: "const", "lin", "quad".

cov_name: str.
The name of the covariance function. Possible values are: "lin", "cub", "log".

Returns

drift_func:
The drift function.

cov_func:
The covariance function.

`polytex.kriging.paraSurface.interp(s, t, expr, split_complexity=1)`

Interpolation (substitute the symbolic variables in the expression).

Parameters

s, t: numpy array.
The parameters of the two profiles for surface parametric kriging (1d array). s has size that same as the number of rows and t the same as the number of columns.

expr: String.
The expression of the target function.

Returns

xinterp: numpy array.
The values of the kriging function. The shape is (s.size,t.size).

`polytex.kriging.paraSurface.kVector(x, symVar, drift_name, cov_name)`

Calculate the kriging matrix.

Parameters

x: numpy array.
The coordinates in parametric space of the points. The shape is (m, 1).

symVar: String.
The variable in parametric space.

cov_name: String.
The name of covariance function.

Returns

return
numpy array. The kriging matrix. The shape is (m, 1).

`polytex.kriging.paraSurface.nugget(M, nugg, b_len)`

Introduce the nugget effect to the kriging matrix.

Parameters

M
[numpy array.] The kriging matrix.

nugg: float.
The nugget effect.

Returns

M
[numpy array.] The kriging matrix with nugget effect.

`polytex.kriging.paraSurface.surface3Dinterp(x, y, z, name_drift, name_cov, nug, return_dict=None, label=None)`

Build the kriging model and interpolate the results

Parameters

x: numpy array.
The coordinates in parametric space of the points. The shape is (m, 1).

y: numpy array.
The coordinates in parametric space of the points. The shape is (n, 1).

z: numpy array.
The coordinates in parametric space of the points. The shape is (m, n).

name_drift: String.
The name of drift function.

name_cov: String.
The name of covariance function.

nug: float.
The nugget effect.

return_dict: dict.
The dictionary to store the results. It is used for multiprocessing purpose.

label: String.
The label of the result.

Returns

z_krig: numpy array.
The interpolated results. The shape is (m, n). It is the return value if return_dict is None (namely, the function is not used in parallel).

return_dict: dict.
The dictionary to store the results. It is used for multiprocessing purpose.

polytex.kriging.projection

This will be an implementation of Kriging: Chapter 15.

The following functions will be included:

1. orthogonal projection of a point onto a curve def point_curve()
2. orthogonal projection of a point onto a surface def point_surf()

polytex.kriging.tool

`polytex.kriging.tool.data_compr(`*matXC*, *data_norm*, *max_err*, *skip_comp*`)`

Data compression by kriging using linear drift and linear covariance.

Parameters

data_norm
[numpy array] Time-Temperature-Alpha-dadt

max_err
[float] The criterion for data compression, which is the maximum local error.

skip_comp
[int] skip (skip_comp-1) data point for data compression. skip_comp >=1.

Returns

data_norm_comp
[TYPE] Data points .

extre
[numpy array] Index of data_norm_comp or extrema choosed according to kriging compression.

`polytex.kriging.tool.fun_crva(`*data_norm*, *drift_para*, *cov_para*`)`

Parameters

data_norm
[numpy array] Time-Temperature-Alpha-dadt.

drift_para
[list] List of string elements.

cov_para
[list] List of string elements.

Returns

expr
[Expression] The kriging expression.

`polytex.kriging.tool.norm`(*data_krig*, *norm_type='axial'*)

This is the normalization function. After input the data of DSC test, this function will normalize temperature, degree of cure and rate of cure.

Parameters

data_krig
[numpy array] Time-Temperature-Alpha-dadt

norm_type
[string, optional] The type of normalization. The default is 'axial'. The other option is 'global' (TODO).

polytex.kriging.volumeKrig

2.2.4 polytex.mesh

polytex.mesh.decimation

`polytex.mesh.decimation.adjacent_from_edge`(*cells*, *edges*, *cell_idx=None*, *return_dict={}*)

Returns the adjacent cells of the edges with the index of the edge as key.

Parameters

cells: (n, 4) array
cell list of the mesh expressed in node connectivity

edges: (m, 2) array
edge list to be collapsed

cell_idx: (n,) array
cell index. If None, it will be generated as np.arange(n). (default: None)

Returns

return_dict: dictionary
a dictionary of adjacent cells with key as the edge

`polytex.mesh.decimation.adjacent_from_edge_parallel`(*cells*, *edge_collapse*, *n_cores=4*)

Get the adjacent cells of the edges to be collapsed.

Parameters

cells: (n, 4) array
cell list of the mesh expressed in node connectivity

edge_collapse: (m, 2) array
edge list to be collapsed

n_cores: int
number of cores to use for multiprocessing (default: 4)

Returns

return_dict: a dictionary of adjacent cells with key as the edge

`polytex.mesh.decimation.construct_tetra_vtk`(*points*, *cells*, *save=False*, *filename='tetra.vtk'*, *path='./'*, *binary=True*)

Construct a UnstructuredGrid tetrahedral mesh from vertices and connectivity.

Parameters

points: (n, 3) array
vertices

cells: (m, 4) array
connectivity

save: bool
whether to save the mesh

filename: str
if save=True, provide a file name

path: str
if save=True, provide a path to save the mesh

binary: bool
whether to save the mesh in binary format

Returns

grid: pyvista.UnstructuredGrid
UnstructuredGrid tetrahedral mesh

`polytex.mesh.decimation.`**`edge_collapse_pipeline`**`(`*mesh*, *surf*, *iteration=1*, *threshold=2*, *n_cores=4*`)`

Edge collapse pipeline. Edges containing boundary and independent points will not be collapsed.

Parameters

points: (n, 3) array
vertices

cells: (m, 4) array
connectivity

surf: a surface mesh of interfaces between materials (subdomains)
iteration: int
number of iterations for edge collapse

threshold: float
threshold for edge collapse

Returns

new_points: (n', 3) array
vertices after edge collapse

new_cells: (m', 4) array
connectivity after edge collapse

`polytex.mesh.decimation.`**`get_boundary_points`**`(`*mesh*`)`

Returns a list of boundary points from this mesh.

Parameters

mesh: pyvista mesh object
boundary_points: A list of boundary points

Returns

pts_boundary_idx: NumPy array
A numpy array in the shape of [n_boundary_points] containing the index of boundary points.

`polytex.mesh.decimation.get_cells(mesh)`

Returns a list of the cells from this mesh with mixed cell types. This properly unpacks the VTK cells array.(safe but now so fast)

Parameters

mesh: pyvista unstructured mesh object
A pyvista mesh object.

Returns

cells: list
A list of cells. The first element of each cell is the number of nodes in the cell.

`polytex.mesh.decimation.get_collapse_direction(edge_indicator)`

Get the direction of edge collapse. The direction is determined by the indicator function of the two vertices.

Parameters

edge_indicator: (n, 2) array
indicator function of the two vertices of an edge

Returns

collapse_indicator: (n,) array
direction of edge collapse. possible values are "forward", "backward", "bilateral", and "neither"

`polytex.mesh.decimation.get_edge_collapse(points, edges, surf_dist, edge_indicator, threshold=1.2)`

Get the edge collapse list.

Parameters

points: (n, 3) array
vertex list

edges: (n', 2) array
edge list

surf_dist: (n,) array
surface distance of the vertices to interfaces

edge_indicator: (n', 2) array
indicator function of the two vertices of an edge. possible values are 0, 1, and 2 for each containing boundary, independent, and free vertices respectively.

threshold: float
 # TODO: explain this parameter
threshold for the surface distance to filter out the edges to be collapsed. The edges to be collapsed are the ones with edge length less than the threshold and surface distance of the two vertices are both greater than the threshold.

Returns

edge_collapse: (n", 2) array
edge collapse list

collapse_indicator: (n", 2) array
indicator function of the two vertices of an edge to be collapsed. possible values are "forward", "backward", "bilateral", and "neither"

`polytex.mesh.decimation.get_edge_length(points, edges)`

Returns the length of an edge given node position and the edge.

Parameters

points: A numpy array in the shape of [n_points, 3]
The points array containing node position

edges: A numpy array in the shape of [n_edges, 2]
The edges array containing node connectivity

Returns

edge_length: A numpy array in the shape of [n_edges]

`polytex.mesh.decimation.get_edges_from_tetra(cells)`

Given cells of tetrahedral mesh, return all the edges.

Parameters

cells: A numpy array in the shape of [n_cells, 4]
The cells array containing node connectivity.

Returns

edges_for_cell: A list of edges
The edges are sorted so that the first node is always smaller than the second. This is to ensure easy searching neighbors. The edges of a cell can be retrieved by edges[cell_index]

edges: A numpy array in the shape of [n_edges, 2]
The edges array containing node connectivity.

`polytex.mesh.decimation.get_maximal_independent_node(edges)`

Get the maximal independent node set from the edge list.

Parameters

edges: (n, 2) array
edge list

Returns

pts_independent: (m,) array
maximal independent node set

`polytex.mesh.decimation.get_surf_dist(surf, points)`

Get the distance from a point to the surface mesh by finding the closest point on the surface mesh with KDTree.

Parameters

surf: A pyvista triangular mesh object
points: (n, 3) array
points to be measured

Returns

surf_dist: (n,) array of float
distance from the points to the surface mesh

idx: (n,) array of int
index of the closest point on the surface mesh

`polytex.mesh.decimation.get_vertex_indicator(n_points, pts_boundary_idx, pts_independent, edges)`

Get the indicator function of the vertices:
0 for boundary vertices, 1 for independent vertices, 2 for free vertices.

The indicator function is used to determine the nodes to be collapsed. The nodes to be collapsed are the ones with indicator function equal to 2 while 0 and 1 are fixed.

Parameters

n_points: int
number of vertices

pts_boundary_idx: (m,) array
indices of boundary points

pts_independent: (n,) array
indices of independent points

edges: (n', 2) array
edge list

Returns

vertex_indicator: (n_points,) array
indicator function of the vertices

edge_indicator: (n', 2) array
indicator function of the two vertices of an edge

`polytex.mesh.decimation.renumber_points(pts_del, cells, proc_num, return_dict={})`

Renumber the points in cells after some points are deleted.

Parameters

pts_del: A list of points to be deleted
cells: A numpy array in the shape of [n_cells, 4]
The cells array containing node connectivity

proc_num: Int, the number of process for multiprocessing.
return_dict: A dictionary to store the result of each process.

Returns

num_diff: A numpy array in the shape of [n_cells, 4]
The difference between the original node index and the new node index

`polytex.mesh.decimation.tetra_edge_collapse(edges, collapse_indicator, edge_adjacent, points, cells, n_cores)`

Collapse edges of tetrahedral mesh.

Parameters

edges: (n, 2) array
collapse_indicator: (n,) array
direction of edge collapse. possible values are "forward", "backward", "bilateral", and "neither"

edge_adjacent: dict
adjacent cells of each edge

points: (n, 3) array
node positions

cells: (n, 4) array
node connectivity

Returns

points: (n, 3) array
node positions after edge collapse

cells: (n, 4) array
node connectivity after edge collapse

polytex.mesh.features

polytex.mesh.features.**voxelize**(*mesh*, *density=None*, *check_surface=True*, *density_type='cell_number'*, *contained_cells=False*)

Voxelize surface mesh to UnstructuredGrid. The bounding box of the voxelized mesh possibly smaller than the bounding box of the surface mesh when cell_size type of density is used.

Parameters

mesh
[pyvista.PolyData] Surface mesh to be voxelized.

density
[float, int, or list of float or int] Uniform size of the voxels when single float passed. A list of densities along x,y,z directions. Defaults to 1/100th of the mesh length for cell_size (float or list) flavor density and 50 cells in each direction for cell_number density (int or list).

check_surface
[bool] Specify whether to check the surface for closure. If on, then the algorithm first checks to see if the surface is closed and manifold. If the surface is not closed and manifold, a runtime error is raised.

density_type
[str] Specify the type of density to use. Options are 'cell_number' or 'cell_size'. When 'cell_number' is used, the density is the number of cells in each direction. When 'cell_size' is used, the density is the size of cells in each direction.

contained_cells
[bool] If True, only cells that fully are contained in the surface mesh will be selected. If False, extract the cells that contain at least one of the extracted points.

Returns

vox
[pyvista.UnstructuredGrid] Voxelized unstructured grid of the original mesh.

ugrid
[pyvista.UnstructuredGrid] The backgrond mesh for voxelization

Examples:
Create an equal density voxelized mesh using cell_size density.

```
>>> import pyvista as pv
    ..
```

```
>>> from pyvista import examples
    ..
```

```
>>> import polytex.mesh as ms
    ..
```

```
>>> mesh = pv.PolyData(examples.load_uniform().points)
    ..
```

```
>>> vox, _ = ms.voxelize(mesh, density=0.5, density_type='cell_size')
    ..
```

```
>>> vox.plot(show_edges = True)
    ..
```

Create a voxelized mesh with specified number of elements in x, y, and z dimensions.

```
>>> mesh = pv.PolyData(examples.load_uniform().points)
    ..
```

```
>>> vox, _ = ms.voxelize(mesh, density=[50, 50, 50], density_type='cell_
↪number', contained_cells=False)
    ..
```

```
>>> vox.plot(show_edges = True)
    ..
```

polytex.mesh.from_image

To check the use of this module, please refer to the example "mesh_from_image.py"
in the test folder of PolyTex.

polytex.mesh.from_image.**get_vcut_plane**(*surf_mesh*, *direction='x'*, *skip=1*)

Get the vertical cut plane of the surf mesh in the direction of x, y, or z axis through (boundary) cutting edge extraction.

Parameters

surf_mesh
[pyvista.PolyData] The surface mesh.

direction
[str, optional] The direction of the vertical cut plane, by default 'x'. The direction can be 'x', 'y', or 'z'.

skip
[int, optional] The number of cut planes to skip. The unit is slice in the direction of the vertical cut plane, by default 1.

Returns

vcut_plane
[numpy.ndarray] The vertical cut planes of the surf mesh.

trajectory
[numpy.ndarray] The trajectory (centroid) of the vertical cut plane calculated by averaging the coordinates of the points on the cutting edge.

polytex.mesh.from_image.**im_to_ugrid**(*im*)

Convert image or image sequence to an unstructured grid.

Parameters

im
[image object] The image sequence stored as a single tif file.

Returns

ugrid
[pyvista.UnstructuredGrid] The unstructured grid discretized from the image with voxels.

im_dim
[numpy.ndarray] The image dimension.

Examples

```
>>> import polytex as ptx
>>> im = ptx.example("image")
>>> mesh, mesh_dim = ptx.mesh.im_to_ugrid(im)
```

polytex.mesh.from_image.**mesh_extract**(*ugrid*, *threshold*, *pointdata='Tiff Scalars'*, *type='foreground'*)

Extract part of the mesh from the unstructured grid according to the value of point data.

Parameters

ugrid
[pyvista.UnstructuredGrid] The unstructured grid discretized from the image with voxels.

threshold
[float] The threshold value of the point data specified by the parameter 'pointdata'.

pointdata
[str, optional] The point data name, by default 'Tiff Scalars'.

type
[str, optional] The type of the extracted mesh, by default "forground". The type can be "forground" or "background". If the type is "foreground", the extracted mesh is where the point data is greater than the threshold value. If the type is "background", the extracted mesh is where the point data is less than the threshold value.

Returns

subset_final
[pyvista.UnstructuredGrid] The extracted volume mesh.

surf
[pyvista.PolyData] The extracted surface mesh.

polytex.mesh.from_image.**mesh_separation**(*mesh*, *plot=False*)

Separate the mesh object into different regions according to the connectivity of the mesh. It may not work for mesh with multiple regions that are connected.

Parameters

mesh
[pyvista.UnstructuredGrid] The mesh object.

Returns

mesh_dict
[dict] The dictionary of the separated mesh objects. The key is the region number and the value is the mesh object.

polytex.mesh.from_image.**slice_plot**(*vcut_planes*, *skip=10*, *marker='o'*, *marker_size=0.1*, *direction='z'*, *dpi=300*, *save=False*, *save_path=None*)

Plot the vertical cut planes.

Parameters

vcut_planes
[numpy.ndarray] The vertical cut planes of the surf mesh stored in a numpy array. The shape of the array is (n_points, 3).

skip
[int, optional] The number of cut planes to skip when plotting the vertical cut planes, by default 10.

marker
[str, optional] The marker type, by default 'o'.

marker_size
[float, optional] The marker size, by default 0.1.

dpi
[int, optional] The resolution of the figure, by default 300.

save
[bool, optional] Whether to save the figure, by default False.

save_path
[str, optional] The path to save the figure, by default None. If save is True, the save_path must be specified.

Returns

None

polytex.mesh.mesh

polytex.mesh.mesh.**background_mesh**(*bbox*, *voxel_size=None*)

Generate a voxel background mesh.

Parameters

bbox: bounding box of the background mesh specified through a numpy array
contains the minimum and maximum coordinates of the bounding box [xmin, xmax, ymin, ymax, zmin, zmax]

voxel_size: voxel size of the background mesh, type: None, float, or numpy.ndarray
if *None*, the voxel size is set to the 1/20 of the diagonal length of the bounding box;

if *float*, the voxel size is set to the float value in x, y, z directions;

if *list*, *set*, or *tuple* of size 3, the voxel size is set to the values for the x-, y- and z- directions.

Returns

grid
[pyvista mesh object (UnstructuredGrid)]

mesh_shape
[tuple] shape of the mesh

`polytex.mesh.mesh.construct_tetra_vtk`(*points*, *cells*, *save=None*, *binary=True*)

Construct a UnstructuredGrid tetrahedral mesh from vertices and connectivity.

Parameters

points: (n, 3) array
vertices

cells: (m, 4) array
connectivity

save: str
The path and file name of the vtk file to be saved ("./tetra.vtk"). If None, the vtk file will not be saved.

binary: bool
whether to save the mesh in binary format

Returns

grid: pyvista.UnstructuredGrid
UnstructuredGrid tetrahedral mesh

`polytex.mesh.mesh.find_cells_within_bounds`(*mesh*, *bounds*)

Find the index of cells in this mesh within bounds.

Parameters

mesh: pyvista mesh
bounds: type:iterable(float)
list of 6 values, [xmin, xmax, ymin, ymax, zmin, zmax]

Returns

type: numpy.ndarray
array of cell indices within bounds.

Examples

>> mesh = pv.PolyData(np.random.rand(10, 3)) >> indices = find_cells_within_bounds(mesh, [0, 1, 0, 1, 0, 1])

`polytex.mesh.mesh.intersection_detect`(*label_set_dict*)

Find the intersection of fiber tows from implicit surface.

Parameters

label_set_dict: dictioanry
dictionary of the label sets of the fiber tows (key: yarn indices, value: sparse matrix of cell queries)

Returns

type: dictionary of the indices of intersected cell
key: yarn indices 1_yarn indices 2, value: sparse matrix of cell indices

`polytex.mesh.mesh.isInbBox`(*bbox*, *point*)

Determine if point is within a bounding box (bbox) that parallel to the principle axes of global Cartesian coordinate.

Parameters

bbox: list
bounding box, [xmin, xmax, ymin, ymax, zmin, zmax].

point: list
[x, y, z]

Returns

True or False

`polytex.mesh.mesh.label_mask`(*mesh_background*, *mesh_tri*, *tolerance=1e-07*, *check_surface=False*)

Store the label of each fiber tow for intersection detection.

Parameters

mesh_background: pyvista.UnstructuredGrid
background mesh

mesh_tri: pyvista.PolyData
tubular mesh of the fiber tows

tolerance: float
tolerance for the enclosed point detection

Returns

mask: type: numpy.ndarray (bool)
mask of the background mesh, True for the cells that are within the bounds of the tubular mesh

label_yarn: type: numpy.ndarray (int) (1D)

`polytex.mesh.mesh.mesh_correction`(*cells*, *points*, *theta_res*)

Close the ends of the tubular mesh with triangles.

Parameters

cells: list
list of cells

points: numpy.ndarray
vertices of the tubular mesh

theta_res: int
number of points in the radial direction of the tubular mesh. The first and the last vertex of each radial direction point list are repeated. However, they are considered as two different points when considering theta resolution (theta_res).

Returns

points: numpy.ndarray
vertices of the tubular mesh

cells: list
list of cells

`polytex.mesh.mesh.structured_cylinder_vertices`(*a*, *b*, *h*, *theta_res=5*, *h_res=5*)

Generate points on an ellipse.

Parameters

a
[float] semi-major axis

b
[float] semi-minor axis

h
[float] height

theta_res
[int, optional] number of points, by default 5.

h_res: int, optional
number of points. by default 5.

Returns

points: numpy.ndarray
vertices on the ellipse surface (x, y, z).

`polytex.mesh.mesh.`**`to_meshio_data`**(*mesh*, *theta_res*, *correction=True*)

Convert PyVista flavor data structure to meshio.

Parameters

mesh: PyVista.DataSet
Any PyVista mesh/spatial data type.

theta_res:
number of points in the radial direction

correction: boolean
if True, tubular mesh will be closed at the ends with triangles.

Returns

points: numpy.ndarray
vertices of the tubular mesh

cells: list
list of cells

point_data: numpy.ndarray
point data

cell_data: numpy.ndarray
cell data

`polytex.mesh.mesh.`**`tubular_mesh_generator`**(*theta_res*, *h_res*, *vertices*, *plot=True*)

Generate a tubular mesh.

Parameters

theta_res: int
number of points

h_res: int
number of points

vertices: numpy.ndarray
vertices of the tubular mesh, shape (n, 3) The vertices of the tubular mesh are sorted in the radial direction first, then in the vertical direction. The first vertex is repeated at the end of each radial direction point list.

Returns

mesh
[points on the tubular mesh]

2.2.5 polytex.plot

polytex.plot.color_cluster

`polytex.plot.color_cluster.`**`color_cluster`**(*clusters*)

polytex.plot.image_plot

`polytex.plot.image_plot.`**`lighten_color`**(*color*, *amount=0.5*, *alpha=1*)

Lightens the given color by multiplying (1-luminosity) by the given amount. Input can be matplotlib color string, hex string, or RGB tuple.

url : https://stackoverflow.com/questions/37765197/darken-or-lighten-a-color-in-matplotlib

Parameters

color
[str or tuple] color to lighten

amount
[float] amount to lighten the color. Value less than 1 produces a lighter color, value greater than 1 produces a darker color.

alpha
[float] alpha value of the color. Default is 1. The alpha value is a float between 0 and 1.

Returns

tuple
modified color in RGBA tuple (float values in the range 0-1).

Examples:
>> lighten_color('g', 0.3, 1)
(0.5500000000000002, 0.9999999999999999, 0.5500000000000002, 1)
>> lighten_color('#F034A3', 0.6, 0.5)
(0.9647058823529411, 0.5223529411764707, 0.783529411764706, 0.5)
>> lighten_color((.3,.55,.1), 0.5)
(0.6365384615384615, 0.8961538461538462, 0.42884615384615377, 1)

`polytex.plot.image_plot.`**`para_plot`**()

This function is used to describe the parameters of the plot.

`polytex.plot.image_plot.`**`plot_on_img`**(*x*, *y*, *backgroundImg*, *labels=[]*, *save=False*)

This function is used to plot the image.

Parameters

x,y: numpy array
if x.shape[1]>1,

labels: list of string
, legend

img:
a image as the background

save: bool
if True, save the image, default is False.

Returns

None.

`polytex.plot.image_plot.vert_sub_plot(num_plots, vspace, x, y, labels)`

This function is used to plot multiple subplots vertically.

Parameters

num_plots
[int] The number of subplots.

vspace
[float] The vertical space between subplots.

x
[numpy array] The x-axis data. The shape of x should be (num_points, num_plots).

y
[numpy array] The y-axis data. The shape of y should be (num_points, num_plots).

labels
[list] The labels of subplots.

Returns

fig
[matplotlib.figure.Figure] The figure object.

`polytex.plot.image_plot.xy_interp(*axis_list, num=100, raw=False)`

Interpolate the axis_list to the same x-axis and calculate the mean y-axis for all the input x-y pairs (midline).

TODO: check what happens if the range of x-axis is not the same for all the input axis_list.

Parameters

axis_list
[list of np.ndarray] Each element is a 2D array with shape (n, 2), where n is the number of points. The first column is x-axis and the second column is y-axis.

num
[int, optional] The number of points to interpolate. The default is 100.

raw
[bool, optional] If True, return the raw interpolated axis_list. The default is False.

Returns

mid
[np.ndarray] The interpolated midline with shape (n, 2), where n is the number of points. The first column is x-axis and the second column is y-axis.

raw_interp
[np.ndarray] The raw interpolated axis_list with shape (n, m), where n is the number of points and m is the number of input axis_list. The first m columns are x-axis and the last m columns are y-axis.

Notes

[algorithm - How to interpolate a line between two other lines in python] (https://stackoverflow.com/questions/49037902/how-to-interpolate-a-line-between-two-other-lines-in-python/49041142#49041142)

Examples

```
>>> x1 = np.linspace(0, 10, 10)
>>> y1 = np.linspace(0, 15, 10)
>>> x2 = np.linspace(0, 10, 20)
>>> y2 = np.linspace(0, 12, 20)
>>> x3 = np.linspace(0, 9, 30)
>>> y3 = np.linspace(0, 18, 30)
>>> interp(np.vstack((x1, y1)).T, np.vstack((x2, y2)).T, np.vstack((x3, y3)).T)
```

2.2.6 polytex.stats

polytex.stats.bw_opt

`polytex.stats.bw_opt.`**`bw_scott`**(*sigma*, *n=''*)

Scott's rule for bandwidth selection.

Parameters

sigma
[float] The standard deviation of the data.

n
[int] The number of data points.

Returns

bw
[float] The bandwidth of the kernel.

`polytex.stats.bw_opt.`**`log_likelihood`**(*pdf*)

Calculate the likelihood of the given probability density function. The likelihood is:

$$L =$$

rac{1}{N}sum_{i=1}^{N} f(x_i)

Parameters

pdf
[Numpy array] The probability density function.

Returns

LL
[float] The log-likelihood of the given probability density function.

`polytex.stats.bw_opt.`**`opt_bandwidth`**(*variable*, *x_test*, *bw*)

Find the optimal bandwidth by tuning of the *bandwidth* parameter via cross-validation and returns the parameter value that maximizes the log-likelihood of data.

Parameters

variable
[Numpy array] A N x 1 dimension numpy array. The data to apply the kernel density estimation.

x_test
[Numpy array] Test data to get the density distribution.

bw
[list of float] The bandwidth of the kernels to be tested.

Returns

kde.bandwidth
[float] The optimal bandwidth of the kernel.

polytex.stats.kde

`polytex.stats.kde.kdePlot`(*xkde*, *ykde*, *cluster_center_idx*)

Parameters

xkde
[Numpy array] The normalized distance.

ykde
[Numpy array] The probability density distribution corresponding to the normalized distance.

cluster_center_idx
[Numpy array] The index of the cluster centers.

Returns

None.

`polytex.stats.kde.kdeScreen`(*variable*, *x_test*, *bw*, *kernels='gaussian'*, *plot='False'*)

This function estimates the probability density distribution of the input variable with the non-parametric kernel density estimation (KDE) method. The local maxima and minima of the probability density distribution are identified to decompose the input variable into a set of clusters. The former is used as the cluster centers and the latter is used as the cluster boundaries.

Parameters

variable
[Numpy array] A N x 1 dimension numpy array to apply the kernel density estimation.

x_test
[Numpy array] Test data to get the density distribution. It has the same shape as the given variable. It should cover the whole range of the variable.

bw
[float] The bandwidth of the kernel.

kernel
[string, optional] The kernel to use. The default is 'gaussian'. The possible values are {'gaussian', 'tophat', 'epanechnikov', 'exponential', 'linear', 'cosine'}.

plot
[bool, optional] Whether plot the probability density distribution. The default is False.

Returns

clusters
[dictionary] The index of the cluster centers, cluster boundary and the probability density distribution (pdf).

`polytex.stats.kde.movingKDE`(*dataset*, *bw=0.002*, *windows=1*, *n_clusters=20*, *x_test=None*)

This function applies the kernel density estimation (KDE) method to the input dataset with a moving window. Namely, the dataset is divided into a set of windows and the KDE method is applied to each window. This allows to capture more details of geometry changes of a fiber tow.

Parameters

dataset
[Numpy array] A N x 2 dimension numpy array for kernel density estimation. The first colum should be the variable under analysis, the second is the label of cross-sections that the variable belongs to.

bw
[Numpy array or float, optional] A range of bandwidth values for kde operation usually generated with np.arange(). The optimal bandwidth will be identified within this range and be used for kernel density estimation. If a number is given, the number will be used as the bandwidth for kernel estimation.

windows
[int,] The number of windows (segmentations) for KDE analysis. The default is 1, namely, the whole dataset is used for KDE analysis and gives the same result as using the function kdeScreen() directly.

n_clusters
[int] The target number of cluster_center. The default is 20.

x_test
[Numpy array] Test data to get the density distribution. The default is None.

Returns

kdeOutput
[Numpy array] A N x 3 dimension numpy array. The first column is the label of the window under analysis, the second is normlized distance, the third is the probability density.

cluster_center
[Numpy array] A M x N dimension numpy array. M is the number of windows and N-1 is the number of cluster centers. The first column is the maximum index for each window, the following columns are the cluster centers.

2.2.7 polytex.misc

Module contents

`polytex.misc.cai_berdichevsky`(*vf*, *rf*, *packing='Quad'*, *tensorial=False*)

Calculate the fiber tow permeability for a given fiber packing pattern according to cai's model.

$$
\begin{aligned}
K_L =& 0.211 r^2 \left((V_a - 0.605) \left(\frac{0.907 V_f}{V_a} \right)^{(-0.181)} * \left(\frac{1 - 0.907 V_f}{V_a} \right)^{(2.66)} \right. \\
& \left. + 0.292 (0.907 - V_a) (V_f)^{(-1.57)} (1 - V_f)^{(1.55)} \right) \\
K_T =& 0.229 r^2 \left(\frac{1.814}{V_a} - 1 \right) \left(\frac{\left(1 - \sqrt{\frac{V_f}{V_a}} \right)}{\sqrt{\frac{V_f}{V_a}}} \right)^{2.5}
\end{aligned}
$$

Parameters

vf
[float or array_like] Fiber volume fraction.

rf
[float or array_like] Fiber radius (m). If vf and rf are arrays, they must have the same shape.

packing
[string] Fiber packing pattern. Valid options are "Quad" and "Hex".

tensorial
[bool] If True, return the permeability tensor (a list with 9 elements). Otherwise, return the permeability components that parallel and perpendicular to the fiber tow as a list of 3 floats. The default is False.

Returns

k
[array-like] Fiber tow permeability. If tensorial is True, return a list with 9 elements. Otherwise, return a list of 3 floats (k11, k22, k33), corresponding to the permeability components that parallel and perpendicular to the fiber tow. The units are m^2. Note that the principal permeability components k22 and k33 are equal.

References

Cai, Z. and A. Berdichevsky, An improved self-consistent method for estimating the permeability of a fiber assembly. Polymer composites, 1993. 14(4): p. 314-323

Examples

```
>>> rf = 6.5e-6
>>> k = cai_berdichevsky(vf=0.3, rf=rf, packing='Hex')
>>> k / rf**2
array([[0.04415829, 0.10741589, 0.10741589]])
>>> k = cai_berdichevsky(vf=0.7, rf=rf, packing='Hex')
>>> k / rf**2
array([[0.00604229, 0.00162723, 0.00162723]])
>>> k = cai_berdichevsky(vf=0.3, rf=rf, packing='Hex', tensorial=True)
>>> k / rf**2
array([[0.04415829, 0.        , 0.        , 0.        , 0.10741589,
        0.        , 0.        , 0.        , 0.10741589]])
```

polytex.misc.**compress_file**(*zipfilename*, *dirname*)

Compresses all files and subdirectories in the specified directory.

Parameters

zipfilename
[str] The name of the zip file, including the path.

dirname
[str] The name of the directory to be compressed, including the path.

Returns

int
1 if the compression is successful, otherwise 0.

Examples

```
>>> compress_file("test.zip", "./test")
```

polytex.misc.**drummond_tahir**(*vf*, *rf*, *packing='Quad'*, *tensorial=False*)

Calculate the fiber tow permeability for a given fiber packing pattern according to Drummond and Tahir's model.

$$K_l = \frac{r^2}{4V_f}\left(-lnV_f - 1.476 + 2V_f - 0.5V_f^2\right)$$
$$K_{tQuad} = \frac{r^2}{8V_f}\left(-lnV_f - 1.476 + \frac{2V_f - 0.796V_f}{1 + 0.489V_f - 1.605{V_f}^2}\right)$$
$$K_{tHex} = \frac{r^2}{8V_f}\left(-lnV_f - 1.497 + 2V_f - \frac{V_f^2}{2} - 0.739V_f^4 + \frac{2.534V_f^5}{1 + 1.2758V_f}\right)$$

Parameters

vf
[float or array_like] Fiber volume fraction.

rf
[float or array_like] Fiber radius (m). If vf and rf are arrays, they must have the same shape.

packing
[string] Fiber packing pattern. Valid options are "Quad" and "Hex".

tensorial
[bool] If True, return the permeability tensor (a list with 9 elements). Otherwise, return the permeability components that parallel and perpendicular to the fiber tow as a list of 3 floats. The default is False.

Returns

k
[array-like] Fiber tow permeability. If tensorial is True, return a list with 9 elements. Otherwise, return a list of 3 floats (k11, k22, k33), corresponding to the permeability components that parallel and perpendicular to the fiber tow. The units are m^2. Note that the principal permeability components k22 and k33 are equal.

References

Drummond J E, Tahir M I. Laminar viscous flow through regular arrays of parallel solid cylinders[J]. International Journal of Multiphase Flow, 1984, 10(5): 515-540..

Examples

```
>>> rf = 6.5e-6
>>> k = drummond_tahir(vf=0.3, rf=rf, packing='Hex')
>>> k / rf**2
array([[0.23581067, 0.10851671, 0.10851671]])
>>> k = drummond_tahir(vf=0.7, rf=rf, packing='Hex')
>>> k / rf**2
array([[0.01274105, 0.01110984, 0.01110984]])
>>> k = drummond_tahir(vf=0.3, rf=rf, packing='Hex', tensorial=True)
>>> k / rf**2
array([[0.23581067, 0.        , 0.        , 0.        , 0.10851671,
        0.        , 0.        , 0.        , 0.10851671]])
```

`polytex.misc.`**`gebart`**(*vf*, *rf*, *packing='Quad'*, *tensorial=False*)

Calculate the fiber tow permeability for a given fiber packing pattern according to Gebart's model.

$$k_l = \frac{8r_f^2}{c}\frac{(1-V_f)^3}{V_f^2}$$

$$k_t = c_1 r_f^2 \sqrt{\left(\sqrt{\frac{V_{f_{max}}}{V_f}} - 1\right)^5}$$

Parameters

vf
[float or array_like] Fiber volume fraction.

rf
[float or array_like] Fiber radius (m). If vf and rf are arrays, they must have the same shape.

packing
[string] Fiber packing pattern. Valid options are "Quad" and "Hex".

tensorial
[bool] If True, return the permeability tensor (a list with 9 elements). Otherwise, return the permeability components that parallel and perpendicular to the fiber tow as a list of 3 floats. The default is False.

Returns

k
[array-like] Fiber tow permeability. If tensorial is True, return a list with 9 elements. Otherwise, return a list of 3 floats (k11, k22, k33), corresponding to the permeability components that parallel and perpendicular to the fiber tow. The units are m^2. Note that the principal permeability components k22 and k33 are equal.

References

Gebart BR. Permeability of Unidirectional Reinforcements for RTM. Journal of Composite Materials. 1992;26(8):1100-33.

Examples

```
>>> print(gebart(vf=0.5, rf=1.7e-5, packing="Quad", tensorial=False))
[2.02807018e-11 3.73472833e-12 3.73472833e-12]
>>> print(gebart(vf=0.5, rf=1.7e-5, packing="Quad", tensorial=True))
[2.02807018e-11 0.00000000e+00 0.00000000e+00 0.00000000e+00
 3.73472833e-12 0.00000000e+00 0.00000000e+00 0.00000000e+00
 3.73472833e-12]
>>> print(gebart(vf=0.5, rf=1.7e-5, packing="Hex", tensorial=False))
[2.18113208e-11 4.72786599e-12 4.72786599e-12]
>>> print(gebart(vf=0.5, rf=1.7e-5, packing="Hex", tensorial=True))
[2.18113208e-11 0.00000000e+00 0.00000000e+00 0.00000000e+00
 4.72786599e-12 0.00000000e+00 0.00000000e+00 0.00000000e+00
 4.72786599e-12]
```

`polytex.misc.`**`perm_rotation`**(*permeability*, *orientation*, *inverse=False*, *disable_tqdm=True*)

Rotate the permeability tensor according to the yarn orientation in the world coordinate system.

Parameters

permeability: ndarray
The principal permeability tensor of the yarn in the local coordinate system of the yarn. Shape: (n, 9)

orientation: ndarray
The orientation of the yarn in the world coordinate system. Shape: (n, 3)

inverse: bool
If True, the inverse of permeability tensor is returned.

Returns

perm_rot: ndarray
The rotated permeability tensor. Shape: (n, 9)

D
[ndarray] The inverse of the rotated permeability tensor. Shape: (n, 9)

`polytex.misc.`**`porosity_tow`**(*rho_lin*, *area_xs*, *rho_fiber=2550*, *fvf=False*)

Calculate local porosity of a tow based on its cross-sectional area and linear density.

Parameters

rho_lin: float
Linear density of the tow. Unit: Tex (g/1000m)

area_xs: array-like
Cross-sectional area of the tow. Unit: m^2. Shape: (n cross-sections, 1).

rho_fiber: float
Volume density of the fiber. Unit: kg/m^3. Default: 2550 (glass fiber).

fvf: bool
Whether to return fiber volume fraction. Default: False. If True, return fiber volume fraction instead of porosity.

Returns

porosity: array-like
Local porosity of the tow. Shape: (n cross-sections, 1). Unit: 1. The fiber volume fraction is returned if fvf is True.

Examples

```
>>> rho_lin = 275 # 275 Tex
>>> area_xs = np.array([0.16, 0.22, 0.15])/1e6 # mm^2 to m^2
>>> rho_fiber = 2550 # kg/m^3
>>> porosity = porosity_tow(rho_lin, area_xs, rho_fiber, fvf=True)
>>> print(porosity)
[0.67401961 0.49019608 0.71895425]
```

2.2.8 polytex.thirdparty

Subpackages

Submodules

polytex.thirdparty.bcolors module

class `polytex.thirdparty.bcolors.`**`bcolors`**

Bases: `object`

[Simple Python class to create colored messages for command line printing] (https://gist.github.com/tuvokki/14deb97bef6df9bc6553)

Helper class to print colored output

To use code like this, you can do something like

print bcolors.WARNING

- "Warning: No active frommets remain. Continue?"
- bcolors.ENDC

you can also use the convenience method bcolors.colored like this

```
>>> print(bcolors.colored("This frumble is underlined", bcolors.UNDERLINE))
```

or use one of the following convenience methods:
warning, fail, ok, okblue, header

Examples

```
>>> print(bcolors.warning("This is dangerous"))
```

Method calls can be nested too, print an underlined header do this:

```
>>> print(bcolors.header(bcolors.colored("The line under this text is purple too ...
↪ ", bcolors.UNDERLINE)))
```

Methods

colored(message, color)
fail(message)
header(message)
ok(message)
okblue(message)
warning(message)

static colored(*message*, *color*)

static fail(*message*)

static header(*message*)

static ok(*message*)

static okblue(*message*)

static warning(*message*)

BOLD = '\x1b[1m'

ENDC = '\x1b[0m'

FAIL = '\x1b[91m'

HEADER = '\x1b[95m'

OKBLUE = '\x1b[94m'

OKCYAN = '\x1b[96m'

OKGREEN = '\x1b[92m'

UNDERLINE = '\x1b[4m'

WARNING = '\x1b[93m'

CHAPTER THREE

LICENSE

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it. For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary.To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

> a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
>
> b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
>
> c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
>
> d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

> a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
>
> b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
>
> c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
>
> d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
>
> e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

> a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
>
> b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
>
> c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
>
> d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
>
> e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
>
> f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES

PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

> <one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands *show w* and *show c* should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <https://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <https://www.gnu.org/licenses/why-not-lgpl.html>.

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z