NURBS-Python Documentation

Release 5.2.9

Onur Rauf Bingol

Sep 10, 2019
Introduction

1 Motivation
   1.1 References .......................................................... 4
   1.2 Author ............................................................... 4

2 Citing NURBS-Python
   2.1 Article .............................................................. 5
   2.2 BibTex ............................................................... 5
   2.3 Licenses ............................................................. 5

3 Questions and Answers
   3.1 What is NURBS? ....................................................... 7
   3.2 Why NURBS-Python? ................................................ 7
   3.3 Why two packages on PyPI? ........................................ 8
   3.4 Minimum Requirements ........................................... 8
   3.5 Help and Support .................................................. 8
   3.6 Issues and Reporting ............................................. 8
   3.7 How can I add a new feature? .................................... 9
   3.8 Why doesn’t NURBS-Python have XYZ feature? ............... 9
   3.9 NURBS-Python API changes ..................................... 9
   3.10 Plotly v4 API changes ........................................... 9

4 Installation and Testing
   4.1 Install via Pip ...................................................... 11
   4.2 Install via Conda .................................................. 11
   4.3 Manual Install ..................................................... 12
   4.4 Development Mode ............................................... 12
   4.5 Checking Installation ............................................ 12
   4.6 Testing .............................................................. 13
   4.7 Compile with Cython ............................................. 13
   4.8 Docker Containers ................................................ 14

5 Basics
   5.1 Working with the curves ......................................... 15
   5.2 Working with the surfaces ....................................... 20
   5.3 Working with the volumes ....................................... 20

6 Examples Repository .................................................. 21
# Table of Contents

7 Loading and Saving Data

8 Supported File Formats
   8.1 Text Files ........................................ 25
   8.2 Comma-Separated (CSV) .............................. 28
   8.3 OBJ Format ........................................ 28
   8.4 STL Format ......................................... 29
   8.5 Object File Format (OFF) ......................... 29
   8.6 Custom Formats (libconfig, YAML, JSON) ........ 29
   8.7 Using Templates ................................... 33

9 Compatibility .......................................... 35

10 Surface Generator .................................... 37

11 Knot Refinement ...................................... 39

12 Curve & Surface Fitting .............................. 43
   12.1 Interpolation ..................................... 43
   12.2 Approximation .................................... 43

13 Visualization ........................................ 45
   13.1 Examples ......................................... 45

14 Splitting and Decomposition ....................... 51
   14.1 Splitting ......................................... 51
   14.2 Bézier Decomposition ............................ 55

15 Exporting Plots as Image Files .................... 61

16 Core Modules ........................................ 63
   16.1 User API ......................................... 63
   16.2 Geometry Generators .............................. 184
   16.3 Advanced API ..................................... 193

17 Visualization Modules ............................... 265
   17.1 Visualization Base ............................... 265
   17.2 Matplotlib Implementation ...................... 266
   17.3 Plotly Implementation ............................ 274
   17.4 VTK Implementation .............................. 275

18 Command-line Application ......................... 281
   18.1 Installation ...................................... 281
   18.2 Documentation ................................... 281
   18.3 References ....................................... 281

19 Shapes Module ........................................ 283
   19.1 Installation ...................................... 283
   19.2 Documentation ................................... 283
   19.3 References ....................................... 283

20 Rhino Importer/Exporter ............................ 285
   20.1 Use Cases ........................................ 285
   20.2 Installation ...................................... 285
   20.3 Using with geomdl ............................... 285
   20.4 References ....................................... 286
Welcome to the NURBS-Python (geomdl) v5.x documentation!

NURBS-Python (geomdl) is a cross-platform (pure Python), object-oriented B-Spline and NURBS library. It is compatible with Python versions 2.7.x, 3.4.x and later. It supports rational and non-rational curves, surfaces and volumes.

NURBS-Python (geomdl) provides easy-to-use data structures for storing geometry descriptions in addition to the fundamental and advanced evaluation algorithms.

This documentation is organized into a couple sections:

- Introduction
- Using the Library
- Modules
NURBS-Python (geomdl) is a self-contained, object-oriented pure Python B-Spline and NURBS library with implementations of curve, surface and volume generation and evaluation algorithms. It also provides convenient and easy-to-use data structures for storing curve, surface and volume descriptions.

Some significant features of NURBS-Python (geomdl):

- Self-contained, object-oriented, extensible and highly customizable API
- Convenient data structures for storing curve, surface and volume descriptions
- Surface and curve fitting with interpolation and least squares approximation
- Knot vector and surface grid generators
- Support for common geometric algorithms: tessellation, voxelization, ray intersection, etc.
- Construct surfaces and volumes, extract isosurfaces via construct module
- Customizable visualization and animation options with Matplotlib, Plotly and VTK modules
- Import geometry data from common CAD formats, such as 3DM and SAT.
- Export geometry data into common CAD formats, such as 3DM, STL, OBJ and VTK
- Support importing/exporting in JSON, YAML and libconfig formats
- Jinja2 support for file imports
- Pure Python, no external C/C++ or FORTRAN library dependencies
- Python compatibility: 2.7.x, 3.4.x and later
- For higher performance, optional Compile with Cython options are also available
- Easy to install via pip or conda
- Docker images are available
- geomdl-shapes module for generating common spline and analytic geometries
- geomdl-cli module for using the library from the command line
NURBS-Python (geomdl) contains the following fundamental geometric algorithms:

- Point evaluation
- Derivative evaluation
- Knot insertion
- Knot removal
- Knot vector refinement
- Degree elevation
- Degree reduction

1.1 References

- Fletcher Dunn and Ian Parberry. 3D Math Primer for Graphics and Game Development. CRC Press, 2015.
- Erich Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

1.2 Author

- Onur R. Bingol (@orbingol)
2.1 Article

We have published an article outlining the design and features of NURBS-Python (geomdl) on an open-access Elsevier journal SoftwareX in the January-June 2019 issue.

Please refer to the following DOI link to access the article: https://doi.org/10.1016/j.softx.2018.12.005

2.2 BibTex

You can use the following BibTeX entry to cite the NURBS-Python paper:

```latex
@article{bingol2019geomdl,
    title={NURBS-Python: An open-source object-oriented (NURBS) modeling framework in Python},
    author={Bingol, Onur Rauf and Krishnamurthy, Adarsh},
    journal={SoftwareX},
    volume={9},
    pages={85--94},
    year={2019},
    publisher={Elsevier}
}
```

2.3 Licenses

- Source code is released under the terms of the MIT License
- Examples are released under the terms of the MIT License
• Documentation is released under the terms of CC BY 4.0
CHAPTER 3

Questions and Answers

3.1 What is NURBS?

NURBS is an acronym for Non-Uniform Rational Basis Spline and it represents a mathematical model for generation of geometric shapes in a flexible way. It is a well-accepted industry standard and used as a basis for nearly all of the 3-dimensional modeling and CAD/CAM software packages as well as modeling and visualization frameworks.

Although the mathematical theory of behind the splines dates back to early 1900s, the spline theory in the way we know is coined by Isaac (Iso) Schoenberg and developed further by various researchers around the world.

The following books are recommended for individuals who prefer to investigate the technical details of NURBS:

- A Practical Guide to Splines
- The NURBS Book
- Geometric Modeling with Splines: An Introduction

3.2 Why NURBS-Python?

NURBS-Python started as a final project for M E 625 Surface Modeling course offered in 2016 Spring semester at Iowa State University. The main purpose of the project was development of a free and open-source, object-oriented, pure Python NURBS library and releasing it on the public domain. As an added challenge to the project, everything was developed using Python Standard Library but no other external modules.

In years, NURBS-Python has grown up to a self-contained and extensible general-purpose pure Python spline library with support for various computational geometry and linear algebra algorithms. Apart from the computational side, user experience was also improved by introduction of visualization and CAD exchange modules.

NURBS-Python is a user-friendly library, regardless of the mathematical complexity of the splines. To give a head start, it comes with 40+ examples for various use cases. It also provides several extension modules for

- Using the library directly from the command-line
- Generating common spline shapes
• Rhino .3dm file import/export support
• ACIS .sat file import support

Moreover, NURBS-Python and its extensions are free and open-source projects distributed under the MIT license.

NURBS-Python is not an another NURBS library but it is mostly considered as one of its kind. Please see the Motivation page for more details.

3.3 Why two packages on PyPI?

Prior to NURBS-Python v4.0.0, the PyPI project name was NURBS-Python. The latest version of this package is v3.9.0 which is an alias for the geomdl package. To get the latest features and bug fixes, please use geomdl package and update whenever a new version is released. The simplest way to check if you are using the latest version is

```
$ pip list --outdated
```

3.4 Minimum Requirements

NURBS-Python (geomdl) is tested with Python versions 2.7.x, 3.4.x and higher.

3.5 Help and Support

Please join the email list on Google Groups. It is open for NURBS-Python users to ask questions, request new features and submit any other comments you may have.

Alternatively, you may send an email to nurbs-python@googlegroups.com.

3.6 Issues and Reporting

3.6.1 Bugs and Feature Requests

NURBS-Python project uses the issue tracker on GitHub for reporting bugs and requesting for a new feature. Please use the provided templates on GitHub.

3.6.2 Contributions

All contributions to NURBS-Python are welcomed and I appreciate your time and efforts in advance. I have posted some guidelines for contributing and I would be really happy if you could follow these guidelines if you would like to contribute to NURBS-Python.

It is suggested to open a new ticket GitHub to discuss what you would like to fix or add as a new feature, as it may already been fixed/implemented in some of the development branches.
3.7 How can I add a new feature?

The library is designed to be extensible in mind. It provides a set of abstract classes for creating new geometry types. All classes use evaluators which contain the evaluation algorithms. Evaluator classes can be extended for new type of algorithms. Please refer to BSpline and NURBS modules for implementation examples. It would be also a good idea to refer to the constructors of the abstract classes for more details.

3.8 Why doesn’t NURBS-Python have XYZ feature?

NURBS-Python tries to keep the geometric operations on the parametric space without any conversion to other representations. This approach makes some operations and queries hard to implement. Keeping NURBS-Python independent of libraries that require compilation caused including implementations some well-known geometric queries and computations, as well as a simple linear algebra module. However, the main purpose is providing a base for NURBS data and fundamental operations while keeping the external dependencies at minimum. It is users’ choice to extend the library and add new more advanced features (e.g. intersection computations) or capabilities (e.g. a new file format import/export support).

All advanced features should be packaged separately. If you are developing a feature to replace an existing feature, it might be a good idea to package it separately.

NURBS-Python may seem to keep very high standards by means of accepting contributions. For instance, if you implement a feature applicable to curves but not surfaces and volumes, such a pull request won’t be accepted till you add that feature to surfaces and volumes. Similarly, if you change a single module and/or the function you use most frequently, but that change is affecting the library as a whole, your pull request will be put on hold.

If you are not interested in such level of contributions, it is suggested to create a separate module and add geomdl as its dependency. If you create a module which uses geomdl, please let the developers know via emailing nurbs-python@googlegroups.com and you will be credited as a contributor.

3.9 NURBS-Python API changes

Please refer to CHANGELOG file for details.

3.10 Plotly v4 API changes

As of Plotly release v4.0, the package plotly is now an offline-only package (which is all fine for geomdl). However, The online functionality, e.g. uploading charts to Plotly servers, has been moved to chart-studio package.

To install Plotly v4.x, please follow the instructions below or refer to Plotly website:

3.10.1 Using pip

```bash
$ pip install plotly chart-studio
```
3.10.2 Using conda

```bash
$ conda install -c plotly plotly chart-studio
```

3.10.3 Activating online mode

`geomdl` comes with the offline functionality by default. It also supports the online functionality as an option. A keyword argument `online` should be passed while initializing `VisPlotly.VisConfig` class.

```python
from geomdl.visualization import VisPlotly

# Enable Plotly online functionality
vconf = VisPlotly.VisConfig(online=True)

# Alternatively, the keyword argument may be used during the initialization of the visualization class
vmodule = VisPlotly.VisSurface(online=True)

# Update a hypothetical "surf" object which corresponds to a B-spline or NURBS surface
surf.vis = vmodule
```
CHAPTER 4

Installation and Testing

Installation via pip or conda is the recommended method for all users. Manual method is only recommended for advanced users. Please note that if you have used any of these methods to install NURBS-Python, please use the same method to upgrade to the latest version.

Note: On some Linux and MacOS systems, you may encounter 2 different versions of Python installed. In that case Python 2.x package would use python2 and pip2, whereas Python 3.x package would use python3 and pip3. The default python and pip commands could be linked to one of those. Please check your installed Python version via python -V to make sure that you are using the correct Python package.

4.1 Install via Pip

The easiest method to install/upgrade NURBS-Python is using pip. The following commands will download and install NURBS-Python from Python Package Index.

$ pip install --user geomdl

Upgrading to the latest version:

$ pip install geomdl --upgrade

Installing a specific version:

$ pip install --user geomdl==5.0.0

4.2 Install via Conda

NURBS-Python can also be installed/upgraded via conda package manager from the Anaconda Cloud repository.
Installing:

$ conda install -c orbingol geomdl

Upgrading to the latest version:

$ conda upgrade -c orbingol geomdl

If you are experiencing problems with this method, you can try to upgrade conda package itself before installing the NURBS-Python library.

4.3 Manual Install

The initial step of the manual install is cloning the repository via git or downloading the ZIP archive from the repository page on GitHub. The package includes a setup.py script which will take care of the installation and automatically copy/link the required files to your Python distribution’s site-packages directory.

The most convenient method to install NURBS-Python manually is using pip:

$ pip install --user .

To upgrade, please pull the latest commits from the repository via git pull --rebase and then execute the above command.

4.4 Development Mode

The following command enables development mode by creating a link from the directory where you cloned NURBS-Python repository to your Python distribution’s site-packages directory:

$ pip install --user -e .

Since this command only generates a link to the library directory, pulling the latest commits from the repository would be enough to update the library to the latest version.

4.5 Checking Installation

If you would like to check if you have installed the package correctly, you may try to print geomdl.__version__ variable after import. The following example illustrates installation check on a Windows PowerShell instance:

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\> python
Python 3.6.2 (v3.6.2:5fd33b8, Jul 8 2017, 04:57:36) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import geomdl
>>> geomdl.__version__
'4.0.2'

12 Chapter 4. Installation and Testing
4.6 Testing

The package includes tests/ directory which contains all the automated testing scripts. These scripts require pytest installed on your Python distribution. Then, you can execute the following from your favorite IDE or from the command line:

```
$ pytest
```

pytest will automatically find the tests under tests/ directory, execute them and show the results.

4.7 Compile with Cython

To improve performance, the Core Library of NURBS-Python can be compiled and installed using the following command along with the pure Python version.

```
$ pip install --user . --install-option="--use-cython"
```

This command will generate .c files (i.e. cythonization) and compile the .c files into binary Python modules.

The following command can be used to directly compile and install from the existing .c files, skipping the cythonization step:

```
$ pip install --user . --install-option="--use-source"
```

To update the compiled module with the latest changes, you need to re-cythonize the code.

To enable Cython-compiled module in development mode:

```
$ python setup.py build_ext --use-cython --inplace
```

After the successful execution of the command, you can import and use the compiled library as follows:

```python
# Importing NURBS module
from geomdl.core import NURBS
# Importing visualization module
from geomdl.visualization import VisMPL as vis

# Creating a curve instance
crv = NURBS.Curve()
# Make a quadratic curve
crv.degree = 2

# Skipping control points and knot vector assignments#
# Skipping control points and knot vector assignments#
# Skipping control points and knot vector assignments#
# Set the visualization component and render the curve
crv.vis = vis.VisCurve3D()
crv.render()
```

Before Cython compilation, please make sure that you have Cython module and a valid compiler installed for your operating system.
4.8 Docker Containers

A collection of Docker containers is provided on Docker Hub containing NURBS-Python, Cython-compiled core and the command-line application. To get started, first install Docker and then run the following on the Docker command prompt to pull the image prepared with Python v3.5:

```
$ docker pull idealabisu/nurbs-python:py35
```

On the Docker Repository page, you can find containers tagged for Python versions and Debian (no suffix) and Alpine Linux (-alpine suffix) operating systems. Please change the tag of the pull command above for downloading your preferred image.

After pulling your preferred image, run the following command:

```
$ docker run --rm -it --name geomdl -p 8000:8000 idealabisu/nurbs-python:py35
```

In all images, Matplotlib is set to use `webagg` backend by default. Please follow the instructions on the command line to view your figures.

Please refer to the Docker documentation for details on using Docker.
In order to generate a spline shape with NURBS-Python, you need 3 components:

- degree
- knot vector
- control points

The number of components depend on the parametric dimensionality of the shape regardless of the spatial dimensionality.

- **curve** is parametrically 1-dimensional (or 1-manifold)
- **surface** is parametrically 2-dimensional (or 2-manifold)
- **volume** is parametrically 3-dimensional (or 3-manifold)

Parametric dimensions are defined by $u, v, w$ and spatial dimensions are defined by $x, y, z$.

### 5.1 Working with the curves

In this section, we will cover the basics of spline curve generation using NURBS-Python. The following code snippet is an example to a 3-dimensional curve.

```python
from geomdl import BSpline

# Create the curve instance
crv = BSpline.Curve()

# Set degree
crv.degree = 2

# Set control points
crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]
```

(continues on next page)
As described in the introduction text, we set the 3 required components to generate a 3-dimensional spline curve.

### 5.1.1 Evaluating the curve points

The code snippet is updated to retrieve evaluated curve points.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

points = crv.evalpts

for pt in points:
    print(pt)
```

The `evalpts` property will automatically call `evaluate()` function.

### 5.1.2 Getting the curve point at a specific parameter

`evaluate_single` method will return the point evaluated as the specified parameter.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

point = crv.evaluate_single(0.5)
```
5.1.3 Setting the evaluation delta

Evaluation delta is used to change the number of evaluated points. Increasing the number of points will result in a bigger evaluated points array, as described with \texttt{evalpts} property and decreasing will reduce the size of the \texttt{evalpts} array. Therefore, evaluation delta can also be used to change smoothness of the plots generated using the visualization modules.

\texttt{delta} property will set the evaluation delta. It is also possible to use \texttt{sample\_size} property to set the number of evaluated points.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

crv.delta = 0.005

points_a = crv.evalpts

# Update delta

crv.delta = 0.1

# The curve will be automatically re-evaluated

points_b = crv.evalpts
```

5.1.4 Inserting a knot

\texttt{insert\_knot} method is recommended for this purpose.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

crv.insert_knot(0.5)
```
5.1.5 Plotting

To plot the curve, a visualization module should be imported and curve should be updated to use the visualization module.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

from geomdl.visualization import VisMPL

crv.vis = VisMPL.VisCurve3D()

crv.render()
```

5.1.6 Convert non-rational to rational curve

The following code snippet generates a B-Spline (non-rational) curve and converts it into a NURBS (rational) curve.

```python
from geomdl import BSpline

crv = BSpline.Curve()

crv.degree = 2

crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

crv.knotvector = [0, 0, 0, 1, 1, 1]

from geomdl import convert

crv_rat = convert.bspline_to_nurbs(crv)
```

5.1.7 Using knot vector generator

Knot vector generator is located in the knotvector module.
from geomdl import BSpline
from geomdl import knotvector

# Create the curve instance
crv = BSpline.Curve()

# Set degree
crv.degree = 2

# Set control points
crv.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

# Generate a uniform knot vector
crv.knotvector = knotvector.generate(crv.degree, crv.ctrlpts_size)

5.1.8 Plotting multiple curves

multi module can be used to plot multiple curves on the same figure.

from geomdl import BSpline
from geomdl import multi
from geomdl import knotvector

# Create the curve instance #1
crv1 = BSpline.Curve()

# Set degree
crv1.degree = 2

# Set control points
crv1.ctrlpts = [[1, 0, 0], [1, 1, 0], [0, 1, 0]]

# Generate a uniform knot vector
crv1.knotvector = knotvector.generate(crv1.degree, crv1.ctrlpts_size)

# Create the curve instance #2
crv2 = BSpline.Curve()

# Set degree
crv2.degree = 3

# Set control points
crv2.ctrlpts = [[1, 0, 0], [1, 1, 0], [2, 1, 0], [1, 1, 0]]

# Generate a uniform knot vector
crv2.knotvector = knotvector.generate(crv2.degree, crv2.ctrlpts_size)

# Create a curve container
mcrv = multi.CurveContainer(crv1, crv2)

# Import Matplotlib visualization module
from geomdl.visualization import VisMPL

# Set the visualization component of the curve container
mcrv.vis = VisMPL.VisCurve3D()
Please refer to the Examples Repository for more curve examples.

## 5.2 Working with the surfaces

The majority of the surface API is very similar to the curve API. Since a surface is defined on a 2-dimensional parametric space, the getters/setters have a suffix of \_u and \_v; such as knotvector\_u and knotvector\_v.

For setting up the control points, please refer to the control points manager documentation.

Please refer to the Examples Repository for surface examples.

## 5.3 Working with the volumes

Volumes are defined on a 3-dimensional parametric space. Working with the volumes are very similar to working with the surfaces. The only difference is the 3rd parametric dimension, \( w \). For instance, to access the knot vectors, the properties you will use are knotvector\_u, knotvector\_v and knotvector\_w.

For setting up the control points, please refer to the control points manager documentation.

Please refer to the Examples Repository for volume examples.
Examples Repository

Although using NURBS-Python is straightforward, it is always confusing to do the initial start with a new library. To give you a headstart on working with NURBS-Python, an Examples repository over 50 example scripts which describe usage scenarios of the library and its modules is provided. You can run the scripts from the command line, inside from favorite IDE or copy them to a Jupyter notebook.

The Examples repository contains examples on

- Bézier curves and surfaces
- B-Spline & NURBS curves, surfaces and volumes
- Spline algorithms, e.g. knot insertion and removal, degree elevation and reduction
- Curve & surface splitting and Bézier decomposition (info)
- Surface and curve fitting using interpolation and least squares approximation (docs)
- Geometrical operations, e.g. tangent, normal, binormal (docs)
- Importing & exporting spline geometries into supported formats (docs)
- Compatibility module for control points conversion (docs)
- Surface grid generators (info and docs)
- Geometry containers (docs)
- Automatic uniform knot vector generation via knotvector.generate()
- Visualization components (info, Matplotlib, Plotly and VTK)
- Ray operations (docs)
- Voxelization (docs)

Matplotlib and Plotly visualization modules are compatible with Jupyter notebooks but VTK visualization module is not. Please refer to the NURBS-Python wiki for more details on using NURBS-Python Matplotlib and Plotly visualization modules with Jupyter notebooks.
CHAPTER 7

Loading and Saving Data

NURBS-Python provides the following API calls for exporting and importing spline geometry data:

- `exchange.import_json()`
- `exchange.export_json()`

JSON import/export works with all spline geometry and container objects. Please refer to File Formats for more details.

The following code snippet illustrates a B-spline curve generation and its JSON export:

```python
from geomdl import BSpline
from geomdl import utilities
from geomdl import exchange

# Create a B-Spline curve instance
curve = BSpline.Curve()

# Set the degree
curve.degree = 3

# Load control points from a text file
curve.ctrlpts = exchange.import_txt("control_points.txt")

# Auto-generate the knot vector
curve.knotvector = utilities.generate_knot_vector(curve.degree, len(curve.ctrlpts))

# Export the curve as a JSON file
exchange.export_json(curve, "curve.json")
```

The following code snippet illustrates importing from a JSON file and adding the result to a container object:

```python
from geomdl import multi
from geomdl import exchange

# Import curve from a JSON file
```

(continues on next page)
curve_list = exchange.import_json("curve.json")

# Add curve list to the container
curve_container = multi.CurveContainer(curve_list)
NURBS-Python supports several input and output formats for importing and exporting B-Spline/NURBS curves and surfaces. Please note that NURBS-Python uses right-handed notation on input and output files.

### 8.1 Text Files

NURBS-Python provides a simple way to import and export the control points and the evaluated control points as ASCII text files. The details of the file format for curves and surfaces is described below:

#### 8.1.1 NURBS-Python Custom Format

NURBS-Python provides `import_txt()` function for reading control points of curves and surfaces from a text file. For saving the control points `export_txt()` function may be used.

The format of the text file depends on the type of the geometric element, i.e. curve or surface. The following sections explain this custom format.

#### 2D Curves

To generate a 2D B-Spline Curve, you need a list of \((x, y)\) coordinates representing the control points \((P)\), where

- \(x\): value representing the \(x\)-coordinate
- \(y\): value representing the \(y\)-coordinate

The format of the control points file for generating 2D B-Spline curves is as follows:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>(y_1)</td>
</tr>
<tr>
<td>(x_2)</td>
<td>(y_2)</td>
</tr>
<tr>
<td>(x_3)</td>
<td>(y_3)</td>
</tr>
</tbody>
</table>
The control points file format of the NURBS curves are very similar to B-Spline ones with the difference of weights. To generate a 2D NURBS curve, you need a list of \((x*_{w}, y*_{w}, w)\) coordinates representing the weighted control points \((P_{w})\) where,

- \(x\): value representing the x-coordinate
- \(y\): value representing the y-coordinate
- \(w\): value representing the weight

The format of the control points file for generating 2D NURBS curves is as follows:

\[
\begin{array}{ccc}
  x^*_{w} & y^*_{w} & w \\
  x_{1}^*_{w_{1}} & y_{1}^*_{w_{1}} & w_{1} \\
  x_{2}^*_{w_{2}} & y_{2}^*_{w_{2}} & w_{2} \\
  x_{3}^*_{w_{3}} & y_{3}^*_{w_{3}} & w_{3} \\
\end{array}
\]

**Note:** The `compatibility` module provides several functions to manipulate & convert control point arrays into NURBS-Python compatible ones and more.

### 3D Curves

To generate a 3D B-Spline curve, you need a list of \((x, y, z)\) coordinates representing the control points \((P)\), where

- \(x\): value representing the x-coordinate
- \(y\): value representing the y-coordinate
- \(z\): value representing the z-coordinate

The format of the control points file for generating 3D B-Spline curves is as follows:

\[
\begin{array}{ccc}
  x & y & z \\
  x_{1} & y_{1} & z_{1} \\
  x_{2} & y_{2} & z_{2} \\
  x_{3} & y_{3} & z_{3} \\
\end{array}
\]

To generate a 3D NURBS curve, you need a list of \((x*_{w}, y*_{w}, z*_{w}, w)\) coordinates representing the weighted control points \((P_{w})\) where,

- \(x\): value representing the x-coordinate
- \(y\): value representing the y-coordinate
- \(z\): value representing the z-coordinate
- \(w\): value representing the weight

The format of the control points file for generating 3D NURBS curves is as follows:

\[
\begin{array}{cccc}
  x^*_{w} & y^*_{w} & z^*_{w} & w \\
  x_{1}^*_{w_{1}} & y_{1}^*_{w_{1}} & z_{1}^*_{w_{1}} & w_{1} \\
  x_{2}^*_{w_{2}} & y_{2}^*_{w_{2}} & z_{2}^*_{w_{2}} & w_{2} \\
  x_{3}^*_{w_{3}} & y_{3}^*_{w_{3}} & z_{3}^*_{w_{3}} & w_{3} \\
\end{array}
\]
Note: *compatibility* module provides several functions to manipulate & convert control point arrays into NURBS-Python compatible ones and more.

## Surfaces

Control points file for generating B-Spline and NURBS has 2 options:

First option is very similar to the curve control points files with one noticeable difference to process $u$ and $v$ indices. In this list, the $v$ index varies first. That is, a row of $v$ control points for the first $u$ value is found first. Then, the row of $v$ control points for the next $u$ value.

The second option sets the rows as $v$ and columns as $u$. To generate a **B-Spline surface** using this option, you need a list of $(x, y, z)$ coordinates representing the control points ($P$) where,

- $x$: value representing the x-coordinate
- $y$: value representing the y-coordinate
- $z$: value representing the z-coordinate

The format of the control points file for generating B-Spline surfaces is as follows:

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
</tr>
<tr>
<td>u1</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
</tr>
<tr>
<td>u2</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
<td>$(x, y, z)$</td>
</tr>
</tbody>
</table>

To generate a **NURBS surface** using the 2nd option, you need a list of $(x*w, y*w, z*w, w)$ coordinates representing the weighted control points ($P_w$) where,

- $x$: value representing the x-coordinate
- $y$: value representing the y-coordinate
- $z$: value representing the z-coordinate
- $w$: value representing the weight

The format of the control points file for generating NURBS surfaces is as follows:

<table>
<thead>
<tr>
<th></th>
<th>v0</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
</tr>
</thead>
<tbody>
<tr>
<td>u0</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
</tr>
<tr>
<td>u1</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
</tr>
<tr>
<td>u2</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
<td>$(x<em>w, y</em>w, z*w, w)$</td>
</tr>
</tbody>
</table>

Note: *compatibility* module provides several functions to manipulate & convert control point arrays into NURBS-Python compatible ones and more.

## Volumes

Parametric volumes can be considered as a stacked surfaces, which means that w-parametric axis comes the first and then other parametric axes come.
8.2 Comma-Separated (CSV)

You may use `export_csv()` and `import_csv()` functions to save/load control points and/or evaluated points as a CSV file. This function works with both curves and surfaces.

8.3 OBJ Format

You may use `export_obj()` function to export a NURBS surface as a Wavefront .obj file.

8.3.1 Example 1

The following example demonstrates saving surfaces as .obj files:

```python
# ex_bezier_surface.py
from geomdl import BSpline
from geomdl import utilities
from geomdl import exchange

# Create a BSpline surface instance
surf = BSpline.Surface()

# Set evaluation delta
surf.delta = 0.01

# Set up the surface
surf.degree_u = 3
surf.degree_v = 2
control_points = [[0, 0, 0], [0, 1, 0], [0, 2, -3],
                  [1, 0, 6], [1, 1, 0], [1, 2, 0],
                  [2, 0, 0], [2, 1, 0], [2, 2, 3],
                  [3, 0, 0], [3, 1, -3], [3, 2, 0]]
surf.set_ctrlpts(control_points, 4, 3)

# Evaluate surface
surf.evaluate()

# Save surface as a .obj file
exchange.export_obj(surf, "bezier_surf.obj")
```

8.3.2 Example 2

The following example combines `shapes` module together with `exchange` module:

```python
from geomdl.shapes import surface
from geomdl import exchange

# Generate cylindrical surface
surf = surface.cylinder(radius=5, height=12.5)

# Set evaluation delta
```
surf.delta = 0.01

# Evaluate the surface
surf.evaluate()

# Save surface as a .obj file
exchange.export_obj(surf, "cylindrical_surf.obj")

### 8.4 STL Format

Exporting to STL files works in the same way explained in OBJ Files section. To export a NURBS surface as a .stl file, you may use `export_stl()` function. This function saves in binary format by default but there is an option to change the save file format to plain text. Please see the documentation for details.

### 8.5 Object File Format (OFF)

Very similar to exporting as OBJ and STL formats, you may use `export_off()` function to export a NURBS surface as a .off file.

### 8.6 Custom Formats (libconfig, YAML, JSON)

NURBS-Python provides several custom formats, such as libconfig, YAML and JSON, for importing and exporting complete NURBS shapes (i.e. degrees, knot vectors and control points of single and multi curves/surfaces).

#### 8.6.1 libconfig

libconfig is a lightweight library for processing configuration files and it is often used on C/C++ projects. The library doesn’t define a format but it defines a syntax for the files it can process. NURBS-Python uses `export_cfg()` and `import_cfg()` functions to exporting and importing shape data which can be processed by libconfig-compatible libraries. Although exporting does not require any external libraries, importing functionality depends on libconfig module, which is a pure Python library for parsing libconfig-formatted files.

#### 8.6.2 YAML

YAML is a data serialization format and it is supported by the major programming languages. NURBS-Python uses ruamel.yaml package as an external dependency for its YAML support since the package is well-maintained and compatible with the latest YAML standards. NURBS-Python supports exporting and importing NURBS data to YAML format with the functions `export_yaml()` and `import_yaml()`, respectively.

#### 8.6.3 JSON

JSON is also a serialization and data interchange format and it is natively supported by Python via json module. NURBS-Python supports exporting and importing NURBS data to JSON format with the functions `export_json()` and `import_json()`, respectively.
8.6.4 Format Definition

Curve

The following example illustrates a 2-dimensional NURBS curve. 3-dimensional NURBS curves are also supported and they can be generated by updating the control points.

```plaintext
shape:
  type: curve  # type of the geometry
  count: 1  # number of curves in "data" list (optional)
  data:
    - rational: True  # rational or non-rational (optional)
      dimension: 2  # spatial dimension of the curve (optional)
      degree: 2
      knotvector: [0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1]
      control_points:
        points:  # cartesian coordinates of the control points
          - [0.0, -1.0]  # each control point is defined as a list
          - [-1.0, 0.0]
          - [-1.0, 1.0]
          - [0.0, 1.0]
          - [1.0, 0.0]
          - [1.0, -1.0]
          - [0.0, -1.0]
          weights:  # weights vector (required if rational)
            - 1.0
            - 0.707
            - 1.0
            - 0.707
            - 1.0
            - 0.707
            - 1.0
            - 0.707
            - 1.0
            delta: 0.01  # evaluation delta
```

- **Shape section**: This section contains the single or multi NURBS data. `type` and `data` sections are mandatory.
- **Type section**: This section defines the type of the NURBS shape. For NURBS curves, it should be set to `curve`.
- **Data section**: This section defines the NURBS data, i.e. degrees, knot vectors and `control_points`. `weights` and `delta` sections are optional.

Surface

The following example illustrates a NURBS surface:

```plaintext
shape:
  type: surface  # type of the geometry
  count: 1  # number of surfaces in "data" list (optional)
  data:
    - rational: True  # rational or non-rational (optional)
      dimension: 3  # spatial dimension of the surface (optional)
      degree_u: 1  # degree of the u-direction
      degree_v: 2  # degree of the v-direction
```

(continues on next page)
knotvector_u: [0.0, 0.0, 1.0, 1.0]

knotvector_v: [0.0, 0.0, 0.0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1.0, 1.0, 1.0]

size_u: 2 # number of control points on the u-direction

size_v: 9 # number of control points on the v-direction

control_points:

points: # cartesian coordinates (x, y, z) of the control points
- [1.0, 0.0, 0.0] # each control point is defined as a list
- [1.0, 1.0, 0.0]
- [0.0, 1.0, 0.0]
- [-1.0, 1.0, 0.0]
- [-1.0, 0.0, 0.0]
- [-1.0, -1.0, 0.0]
- [0.0, -1.0, 0.0]
- [1.0, -1.0, 0.0]
- [1.0, 0.0, 0.0]
- [1.0, 1.0, 0.0]
- [0.0, 1.0, 1.0]
- [-1.0, 1.0, 1.0]
- [-1.0, 0.0, 1.0]
- [-1.0, -1.0, 1.0]
- [0.0, -1.0, 1.0]
- [1.0, -1.0, 1.0]
- [1.0, 0.0, 1.0]
- [1.0, 0.0, 1.0]

weights: # weights vector (required if rational)
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0

delta:
- 0.05 # evaluation delta of the u-direction

- 0.05 # evaluation delta of the v-direction

delta:
- 0.05 # evaluation delta of the v-direction

trims: # define trim curves (optional)

count: 3 # number of trims in the "data" list (optional)

data:
- type: spline # type of the trim curve
  rational: False # rational or non-rational (optional)
  dimension: 2 # spatial dimension of the trim curve (optional)
  degree: 2 # degree of the 1st trim
  knotvector: [...] # knot vector of the 1st trim curve
  control_points:
  - points: # parametric coordinates of the 1st trim curve
    - [u1, v1] # expected to be 2-dimensional, corresponding to (u,v)
- \([u_2, v_2]\)
- ...
reversed: 0 # 0: trim inside, 1: trim outside (optional, default is 0)
- type: spline # type of the 2nd trim curve
rational: True # rational or non-rational (optional)
dimension: 2 # spatial dimension of the trim curve (optional)
degree: 1 # degree of the 2nd trim
knotvector: [ ... ] # knot vector of the 2nd trim curve
control_points:
  points: # parametric coordinates of the 2nd trim curve
  - [u_1, v_1] # expected to be 2-dimensional, corresponding to \((u,v)\)
  - [u_2, v_2]
  - ...
weights: # weights vector of the 2nd trim curve (required if rational)
  - 1.0
  - 1.0
  - ...
delta: 0.01 # evaluation delta (optional)
reversed: 1 # 0: trim inside, 1: trim outside (optional, default is 0)
- type: freeform # type of the 3rd trim curve
dimension: 2 # spatial dimension of the trim curve (optional)
points: # parametric coordinates of the 3rd trim curve
  - [u_1, v_1] # expected to be 2-dimensional, corresponding to \((u,v)\)
  - [u_2, v_2]
  - ...
name: "my freeform curve" # optional
reversed: 1 # 0: trim inside, 1: trim outside (optional, default is 0)
- type: container # type of the 4th trim curve
dimension: 2 # spatial dimension of the trim curve (optional)
data: # a list of freeform and/or spline geometries
  - ...
  - ...
name: "my trim curves" # optional
reversed: 1 # 0: trim inside, 1: trim outside (optional, default is 0)

- **Shape section**: This section contains the single or multi NURBS data. `type` and `data` sections are mandatory.
- **Type section**: This section defines the type of the NURBS shape. For NURBS curves, it should be set to `surface`.
- **Data section**: This section defines the NURBS data, i.e. degrees, knot vectors and `control_points`. `weights` and `delta` sections are optional.

Surfaces can also contain trim curves. These curves can be stored in 2 geometry types inside the surface:

- **spline** corresponds to a spline geometry, which is defined by a set of degrees, knot vectors and `control_points`
- **container** corresponds to a geometry container
- **freeform** corresponds to a freeform geometry; defined by a set of points

**Volume**

The following example illustrates a B-spline volume:

```python
shape:
  type: volume # type of the geometry
```
The file organization is very similar to the surface example. The main difference is the parametric 3rd dimension, w.

### 8.6.5 Example: Reading .cfg Files with libconf

The following example illustrates reading the exported .cfg file with libconf module as a reference for libconfig-based systems in different programming languages.

```python
# Assuming that you have already installed 'libconf'
import libconf

# Skipping export steps and assuming that we have already exported the data as 'my_nurbs.cfg'
with open("my_nurbs.cfg", "r") as fp:
    ns = libconf.load(fp)

# 'count' shows the number of shapes loaded from the file
print(ns['shape']['count'])

# Traverse through the loaded shapes
for n in ns['shape']['data']:
    # As an example, we get the control points
    ctrlpts = n['control_points']['points']
```

NURBS-Python exports data in the way that allows processing any number of curves or surfaces with a simple for loop. This approach simplifies implementation of file reading routines for different systems and programming languages.

### 8.7 Using Templates

NURBS-Python v5.x supports Jinja2 templates with the following functions:

- `import_txt()`
To import files formatted as Jinja2 templates, an additional `jinja2=True` keyword argument should be passed to the functions. For instance:

```python
from geomdl import exchange

# Importing a .yaml file formatted as a Jinja2 template
data = exchange.import_yaml("surface.yaml", jinja2=True)
```

NURBS-Python also provides some custom Jinja2 template functions for user convenience. These are:

- `knot_vector(d, np)`: generates a uniform knot vector. 
  - `d`: degree, 
  - `np`: number of control points
- `sqrt(x)`: square root of `x`
- `cubert(x)`: cube root of `x`
- `pow(x, y)`: `x` to the power of `y`

Please see `ex_cylinder_tmpl.py` and `ex_cylinder_tmpl.cptw` files in the *Examples repository* for details on using Jinja2 templates with control point text files.
Compatibility

Most of the time, users experience problems in converting data between different software packages. To aid this problem a little bit, NURBS-Python provides a compatibility module for converting control points sets into NURBS-Python compatible ones.

The following example illustrates the usage of compatibility module:

```python
from geomdl import NURBS
from geomdl import utilities as utils
from geomdl import compatibility as compat
from geomdl.visualization import VisMPL

# Surface exported from your CAD software

# Dimensions of the control points grid
p_size_u = 4
p_size_v = 3

# Control points in u-row order
p_ctrlpts = [[0, 0, 0], [1, 0, 6], [2, 0, 0], [3, 0, 0],
              [0, 1, 0], [1, 1, 0], [2, 1, 0], [3, 1, -3],
              [0, 2, -3], [1, 2, 0], [2, 2, 3], [3, 2, 0]]

# Weights vector
p_weights = [1, 1, 1, 1, 1, 1, 1, 1]

# Degrees
p_degree_u = 3
p_degree_v = 2

# Prepare data for import
```

(continues on next page)
# Combine weights vector with the control points list

```python
# Since NURBS-Python uses v-row order, we need to convert the exported ones
```

```python
n_ctrlptsw = compat.flip_ctrlpts_u(t_ctrlptsw, p_size_u, p_size_v)
```

```python
# Since we have no information on knot vectors, let's auto-generate them
n_knotvector_u = utils.generate_knot_vector(p_degree_u, p_size_u)
n_knotvector_v = utils.generate_knot_vector(p_degree_v, p_size_v)
```

```python
# Import surface to NURBS-Python
```

```python
# Create a NURBS surface instance
surf = NURBS.Surface()
```

```python
# Fill the surface object
surf.degree_u = p_degree_u
ti.knotvector_u = n_knotvector_u
surf.degree_v = p_degree_v
ti.knotvector_v = n_knotvector_v
surf_set_ctrlpts(n_ctrlptsw, p_size_u, p_size_v)
```

```python
# Set evaluation delta
surf.delta = 0.05
```

```python
# Set visualization component
vis_comp = VisMPL.VisSurface()
```

```python
surf.vis = vis_comp
```

```python
# Render the surface
surf.render()
```

Please see `Compatibility Module Documentation` for more details on manipulating and exporting control points. NURBS-Python has some other options for exporting and importing data. Please see `File Formats` page for details.
NURBS-Python comes with a simple surface generator which is designed to generate a control points grid to be used as a randomized input to \texttt{BSpline.Surface} and \texttt{NURBS.Surface}. It is capable of generating customized surfaces with arbitrary divisions and generating hills (or bumps) on the surface. It is also possible to export the surface as a text file in the format described under \texttt{File Formats} documentation.

The classes \texttt{CPGen.Grid} and \texttt{CPGen.GridWeighted} are responsible for generating the surfaces.

The following example illustrates a sample usage of the B-Spline surface generator:

```python
from geomdl import CPGen
from geomdl import BSpline
from geomdl import utilities
from geomdl.visualization import VisMPL
from matplotlib import cm

# Generate a plane with the dimensions 50x100
surfgrid = CPGen.Grid(50, 100)

# Generate a grid of 25x30
surfgrid.generate(50, 60)

# Generate bumps on the grid
surfgrid.bumps(num_bumps=5, bump_height=20, base_extent=8)

# Create a BSpline surface instance
surf = BSpline.Surface()

# Set degrees
surf.degree_u = 3
surf.degree_v = 3

# Get the control points from the generated grid
surf.ctrlpts2d = surfgrid.grid

# Set knot vectors
```

(continues on next page)
surf.knotvector_u = utilities.generate_knot_vector(surf.degree_u, surf.ctrlpts_size_u)
surf.knotvector_v = utilities.generate_knot_vector(surf.degree_v, surf.ctrlpts_size_v)

# Set sample size
surf.sample_size = 100

# Set visualization component
surf.vis = VisMPL.VisSurface(ctrlpts=False, legend=False)

# Plot the surface
surf.render(colormap=cm.terrain)

CPGen.Grid.bumps() method takes the following keyword arguments:

- **num_bumps**: Number of hills to be generated
- **bump_height**: Defines the peak height of the generated hills
- **base_extent**: Due to the structure of the grid, the hill base can be defined as a square with the edge length of \( a \). \( \text{base_extent} \) is defined by the value of \( a/2 \).
- **base_adjust**: Defines the padding of the area where the hills are generated. It accepts positive and negative values. A negative value means a padding to the inside of the grid and a positive value means padding to the outside of the grid.
Knot Refinement

New in version 5.1.

Knot refinement is simply the operation of *inserting multiple knots at the same time*. NURBS-Python (geomdl) supports knot refinement operation for the curves, surfaces and volumes via `operations.refine_knotvector()` function.

One of the interesting features of the `operations.refine_knotvector()` function is the controlling of **knot refinement density**. It can increase the number of knots to be inserted in a knot vector. Therefore, it increases the number of control points.

The following code snippet and the figure illustrate a 2-dimensional spline curve with knot refinement:

```python
from geomdl import BSpline
from geomdl import utilities
from geomdl import exchange
from geomdl.visualization import VisMPL

# Create a curve instance
curve = BSpline.Curve()

# Set degree
curve.degree = 4

# Set control points
curve.ctrlpts = [
    [5.0, 10.0], [15.0, 25.0], [30.0, 30.0], [45.0, 5.0], [55.0, 5.0],
    [70.0, 40.0], [60.0, 60.0], [35.0, 60.0], [20.0, 40.0]
]

# Set knot vector
curve.knotvector = [0.0, 0.0, 0.0, 0.0, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]

# Set visualization component
curve.vis = VisMPL.VisCurve2D()
```

(continues on next page)
# Refine knot vector
operations.refine_knotvector(curve, [1])

# Visualize
curve.render()

The default density value is 1 for the knot refinement operation. The following code snippet and the figure illustrate the result of the knot refinement operation if density is set to 2.

```python
from geomdl import BSpline
from geomdl import utilities
from geomdl import exchange
from geomdl.visualization import VisMPL

# Create a curve instance
curve = BSpline.Curve()

# Set degree
curve.degree = 4

# Set control points
curve.ctrlpts = [
    [5.0, 10.0], [15.0, 25.0], [30.0, 30.0], [45.0, 5.0], [55.0, 5.0],
    [70.0, 40.0], [60.0, 60.0], [35.0, 60.0], [20.0, 40.0]
]

# Set knot vector
curve.knotvector = [0.0, 0.0, 0.0, 0.0, 0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.0, 1.0, 1.0, 1.0]

# Set visualization component
curve.vis = VisMPL.VisCurve2D()

# Refine knot vector
operations.refine_knotvector(curve, [2])

# Visualize
curve.render()
```

The following code snippet and the figure illustrate the result of the knot refinement operation if density is set to 3.

```python
from geomdl import BSpline
from geomdl import utilities
from geomdl import exchange
from geomdl.visualization import VisMPL

# Create a curve instance
curve = BSpline.Curve()

# Set degree
curve.degree = 4

# Set control points
curve.ctrlpts = [
    [5.0, 10.0], [15.0, 25.0], [30.0, 30.0], [45.0, 5.0], [55.0, 5.0],
    [70.0, 40.0], [60.0, 60.0], [35.0, 60.0], [20.0, 40.0]
]

# Set visualization component
curve.vis = VisMPL.VisCurve2D()

# Refine knot vector
operations.refine_knotvector(curve, [3])

# Visualize
curve.render()
```
The following code snippet and the figure illustrate the knot refinement operation applied to a surface with density value of 3 for the u-direction. No refinement was applied for the v-direction.
Chapter 12

Curve & Surface Fitting

Geomdl includes 2 fitting methods for curves and surfaces: approximation and interpolation. Please refer to the Curve and Surface Fitting page for more details on the curve and surface fitting API.

The following sections explain 2-dimensional curve fitting using the included fitting methods. Geomdl also supports 3-dimensional curve and surface fitting (not shown here). Please refer to the Examples Repository for more examples on curve and surface fitting.

12.1 Interpolation

The following code snippet and the figure illustrate interpolation for a 2-dimensional curve:

```python
from geomdl import fitting
from geomdl.visualization import VisMPL as vis

# The NURBS Book Ex9.1
points = ((0, 0), (3, 4), (-1, 4), (-4, 0), (-4, -3))
degree = 3  # cubic curve

# Do global curve interpolation
curve = fitting.interpolate_curve(points, degree)

# Plot the interpolated curve
curve.delta = 0.01
curve.vis = vis.VisCurve2D()
curve.render()
```

The following figure displays the input data (sample) points in red and the evaluated curve after interpolation in blue:

12.2 Approximation

The following code snippet and the figure illustrate approximation method for a 2-dimensional curve:
from geomdl import fitting
from geomdl.visualization import VisMPL as vis

# The NURBS Book Ex9.1
points = ((0, 0), (3, 4), (-1, 4), (-4, 0), (-4, -3))
degree = 3  # cubic curve

# Do global curve approximation
curve = fitting.approximate_curve(points, degree)

# Plot the interpolated curve
curve.delta = 0.01
curve.vis = vis.VisCurve2D()
curve.render()

The following figure displays the input data (sample) points in red and the evaluated curve after approximation in blue:

Please note that a spline geometry with a constant set of evaluated points may be represented with an infinite set of control points. The number and positions of the control points depend on the application and the method used to generate the control points.
NURBS-Python comes with the following visualization modules for direct plotting evaluated curves and surfaces:

- VisMPL module for Matplotlib
- VisPlotly module for Plotly
- VisVTK module for VTK

Examples repository contains over 40 examples on how to use the visualization components in various ways. Please see Visualization Modules Documentation for more details.

13.1 Examples

The following figures illustrate some example NURBS and B-spline shapes that can be generated and directly visualized via NURBS-Python.

13.1.1 Curves

13.1.2 Surfaces

13.1.3 Volumes

13.1.4 Advanced Visualization Examples

The following example scripts can be found in Examples repository under the visualization directory.
mpl_curve2d_tangents.py

This example illustrates a more advanced visualization option for plotting the 2D curve tangents alongside with the control points grid and the evaluated curve.

![2D Curve Tangents](image)

mpl_curve3d_tangents.py

This example illustrates a more advanced visualization option for plotting the 3D curve tangents alongside with the control points grid and the evaluated curve.
mpl_curve3d_vectors.py

This example illustrates a visualization option for plotting the 3D curve tangent, normal and binormal vectors alongside with the control points grid and the evaluated curve.
The following figure illustrates tangent and normal vectors on `ex_surface02.py` example.
NURBS-Python is also capable of splitting the curves and the surfaces, as well as applying Bézier decomposition.

Splitting of curves can be achieved via `operations.split_curve()` method. For the surfaces, there are 2 different splitting methods, `operations.split_surface_u()` for splitting the surface on the u-direction and `operations.split_surface_v()` for splitting on the v-direction.

Bézier decomposition can be applied via `operations.decompose_curve()` and `operations.decompose_surface()` methods for curves and surfaces, respectively.

The following figures are generated from the examples provided in the Examples repository.

### 14.1 Splitting

The following 2D curve is split at $u = 0.6$ and applied translation by the tangent vector using `operations.translate()` method.
Splitting can also be applied to 3D curves (split at $u = 0.3$) without any translation.
Surface splitting is also possible. The following figure compares splitting at $u = 0.5$ and $v = 0.5$. Surfaces can also be translated too before or after splitting operation. The following figure illustrates translation after splitting the surface at $u = 0.5$. 
Multiple splitting is also possible for all curves and surfaces. The following figure describes multi splitting in surfaces. The initial surface is split at $u = 0.25$ and then, one of the resultant surfaces is split at $v = 0.75$, finally resulting in 3 surfaces.
14.2 Bézier Decomposition

The following figures illustrate Bézier decomposition capabilities of NURBS-Python. Let’s start with the most obvious one, a full circle with 9 control points. It also is possible to directly generate this shape via `geomdl.shapes` module.
The following is a circular curve generated with 7 control points as illustrated on page 301 of *The NURBS Book* (2nd Edition) by Piegl and Tiller. There is also an option to generate this shape via `geomdl.shapes` module.
The following figures illustrate the possibility of Bézier decomposition in B-Spline and NURBS surfaces.
The colors are randomly generated via `utilities.color_generator()` function.
Exporting Plots as Image Files

The `render()` method allows users to directly plot the curves and surfaces using predefined visualization classes. This method takes some keyword arguments to control plot properties at runtime. Please see the class documentation on description of these keywords. The `render()` method also allows users to save the plots directly as a file and to control the plot window visibility. The keyword arguments that control these features are `filename` and `plot`, respectively.

The following example script illustrates creating a 3-dimensional Bézier curve and saving the plot as `bezier-curve3d.pdf` without popping up the Matplotlib plot window. `filename` argument is a string value defining the name of the file to be saved and `plot` flag controls the visibility of the plot window.

```python
from geomdl import BSpline
from geomdl import utilities
from geomdl.visualization import VisMPL

# Create a 3D B-Spline curve instance (Beziers Curve)
curve = BSpline.Curve()

# Set up the Bezier curve
curve.degree = 3
curve.ctrlpts = [[10, 5, 10], [10, 20, -30], [40, 10, 25], [-10, 5, 0]]

# Auto-generate knot vector
curve.knotvector = utilities.generate_knot_vector(curve.degree, len(curve.ctrlpts))

# Set sample size
curve.sample_size = 40

# Evaluate curve
curve.evaluate()

# Plot the control point polygon and the evaluated curve
vis_comp = VisMPL.VisCurve3D()
curve.vis = vis_comp
```

(continues on next page)
# Don't pop up the plot window, instead save it as a PDF file

```python
curve.render(filename="bezier-curve3d.pdf", plot=False)
```

This functionality strongly depends on the plotting library used. Please see the documentation of the plotting library that you are using for more details on its figure exporting capabilities.
The following are the lists of modules included in NURBS-Python (geomdl) Core Library. They are split into separate groups to make the documentation more understandable.

16.1 User API

The User API is the main entrance point to the library. It provides geometry classes and containers, as well as the geometric operators and support modules.

The following is the list of the geometry classes included in the library:

16.1.1 B-Spline Geometry

BSpline module provides data storage and evaluation functions for non-rational spline geometries.

Inheritance Diagram
B-Spline Curve

class geomdl.BSpline.Curve(**kwargs)
    Bases: geomdl.abstract.Curve

Data storage and evaluation class for n-variate B-spline (non-rational) curves.

This class provides the following properties:

- type = spline
- id
- order
- degree
- knotvector
- ctrlpts
- delta
- sample_size
- bbox
- vis
- name
- dimension
- evaluator
- rational

The following code segment illustrates the usage of Curve class:

```python
from geomdl import BSpline

# Create a 3-dimensional B-spline Curve
curve = BSpline.Curve()

# Set degree
curve.degree = 3

# Set control points
curve.ctrlpts = [[10, 5, 10], [10, 20, -30], [40, 10, 25], [-10, 5, 0]]

# Set knot vector
curve.knotvector = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1]

# Set evaluation delta (controls the number of curve points)
curve.delta = 0.05

# Get curve points (the curve will be automatically evaluated)
curve_points = curve.evalpts
```

Keyword Arguments:

- precision: number of decimal places to round to. Default: 18
- normalize_kv: activates knot vector normalization. Default: True
• `find_span_func` sets knot span search implementation. Default: `helpers.find_span_linear()`

• `insert_knot_func` sets knot insertion implementation. Default: `operations.insert_knot()`

• `remove_knot_func` sets knot removal implementation. Default: `operations.remove_knot()`

Please refer to the `abstract.Curve()` documentation for more details.

`bbox`

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

**Getter** Gets the bounding box

**Type** tuple

`binormal(parpos, **kwargs)`

Evaluates the binormal vector of the curve at the given parametric position(s).

The `parpos` argument can be

• a float value for evaluation at a single parametric position

• a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

• `normalize`: normalizes the output vector. Default value is `True`.

**Parameters** `parpos` (float, list or tuple) – parametric position(s) where the evaluation will be executed

**Returns** binormal vector as a tuple of the origin point and the vector components

**Return type** tuple

`cpsize`

Number of control points in all parametric directions.

**Note:** This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

**Getter** Gets the number of control points

**Setter** Sets the number of control points

**Type** list

`ctrlpts`

Control points.

Please refer to the wiki for details on using this class member.

**Getter** Gets the control points

**Setter** Sets the control points
**Type** list

cntlpts_size
Total number of control points.

**Getter** Gets the total number of control points

**Type** int
data
Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.
degree
Degree.

Please refer to the wiki for details on using this class member.

**Getter** Gets the degree

**Setter** Sets the degree

**Type** int
delta
Evaluation delta.

Evaluation delta corresponds to the *step size* while *evaluate* function iterates on the knot vector to generate curve points. Decreasing step size results in generation of more curve points. Therefore; smaller the delta value, smoother the curve.

The following figure illustrates the working principles of the delta property:

\[
\left[u_{start}, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end}\right]
\]

Please refer to the wiki for details on using this class member.

**Getter** Gets the delta value

**Setter** Sets the delta value

**Type** float
derivatives \((u, \text{order=0, **kwargs})\)
Evaluates n-th order curve derivatives at the given parameter value.

The output of this method is list of n-th order derivatives. If \text{order} is 0, then it will only output the evaluated point. Similarly, if \text{order} is 2, then it will output the evaluated point, 1st derivative and the 2nd derivative. For instance:

```python
# Assuming a curve (crv) is defined on a parametric domain [0.0, 1.0]
# Let's take the curve derivative at the parametric position u = 0.35
ders = crv.derivatives(u=0.35, order=2)
ders[0]  # evaluated point, equal to crv.evaluate_single(0.35)
ders[1]  # 1st derivative at u = 0.35
ders[2]  # 2nd derivative at u = 0.35
```

**Parameters**

- **u** *(float)* – parameter value
- **order** *(int)* – derivative order

**Returns** a list containing up to {order}-th derivative of the curve
**Return type** list

**dimension**
Spatial dimension.
Spatial dimension will be automatically estimated from the first element of the control points array.
Please refer to the wiki for details on using this class member.

**Getter** Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

**domain**
Domain.
Domain is determined using the knot vector(s).

**Getter** Gets the domain

**evalpts**
Evaluated points.
Please refer to the wiki for details on using this class member.

**Getter** Gets the coordinates of the evaluated points

**Type** list

**evaluate** (**kwargs**)
Evaluates the curve.
The evaluated points are stored in `evalpts` property.

**Keyword arguments:**
- **start**: start parameter
- **stop**: stop parameter

The `start` and `stop` parameters allow evaluation of a curve segment in the range `[start, stop]`, i.e. the curve will also be evaluated at the `stop` parameter value.

The following examples illustrate the usage of the keyword arguments.

```python
# Start evaluating from u=0.2 to u=1.0
curve.evaluate(start=0.2)

# Start evaluating from u=0.0 to u=0.7
curve.evaluate(stop=0.7)

# Start evaluating from u=0.1 to u=0.5
curve.evaluate(start=0.1, stop=0.5)

# Get the evaluated points
curve_points = curve.evalpts
```

**evaluate_list**(param_list)
Evaluates the curve for an input range of parameters.

**Parameters**

- **param_list** (list, tuple) – list of parameters

**Returns**

- evaluated surface points at the input parameters

**Return type** list

---

16.1. User API
evaluate_single(param)
   Evaluates the curve at the input parameter.
   Parameters param (float) – parameter
   Returns evaluated surface point at the given parameter
   Return type list

evaluator
   Evaluator instance.
   Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.
   Please refer to the wiki for details on using this class member.
   Getter Gets the current Evaluator instance
   Setter Sets the Evaluator instance
   Type evaluators.AbstractEvaluator

id
   Object ID (as an integer).
   Please refer to the wiki for details on using this class member.
   Getter Gets the object ID
   Setter Sets the object ID
   Type int

insert_knot(param, **kwargs)
   Inserts the knot and updates the control points array and the knot vector.
   Keyword Arguments:
   • num: Number of knot insertions. Default: 1
   Parameters param (float) – knot to be inserted

knotvector
   Knot vector.
   The knot vector will be normalized to [0, 1] domain if the class is initialized with normalize_kv=True argument.
   Please refer to the wiki for details on using this class member.
   Getter Gets the knot vector
   Setter Sets the knot vector
   Type list

load(file_name)
   Loads the curve from a pickled file.
   Deprecated since version 5.2.4: Use exchange.import_json() instead.
   Parameters file_name (str) – name of the file to be loaded
name
Object name (as a string)

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name

**Type** str

normal(parpos, **kwargs)
Evaluates the normal to the tangent vector of the curve at the given parametric position(s).

The `parpos` argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- `normalize`: normalizes the output vector. Default value is `True`.

**Parameters**
- `parpos` (float, list or tuple) - parametric position(s) where the evaluation will be executed

**Returns** normal vector as a tuple of the origin point and the vector components

**Return type** tuple

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
geom.opt = ['face_id', 4] # creates "face_id" key and sets its value to an integer
geom.opt = ['contents', 'data values'] # creates "face_id" key and sets its value to a string
print(geom.opt) # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt # deletes the contents of the hash map
print(geom.opt) # will print: {}

geom.opt = ['body_id', 1] # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12] # changes the value of "body_id" to 12
print(geom.opt) # will print: {'body_id': 12}

geom.opt = ['body_id', None] # deletes "body_id"
print(geom.opt) # will print: {}
```

Please refer to the wiki for details on using this class member.

- **Getter** Gets the dict
- **Setter** Adds key and value pair to the dict
- **Deleter** Deletes the contents of the dict
opt_get(value)
   Safely query for the value from the opt property.
   
   Parameters value (str) – a key in the opt property
   
   Returns the corresponding value, if the key exists. None, otherwise.

order
   Order.
   
   Defined as order = degree + 1
   
   Please refer to the wiki for details on using this class member.
       
   Getter Gets the order
   
   Setter Sets the order
   
   Type int

dimension
   Parametric dimension.
   
   Please refer to the wiki for details on using this class member.
       
   Getter Gets the parametric dimension
   
   Type int

range
   Domain range.
       
   Getter Gets the range

rational
   Defines the rational and non-rational B-spline shapes.
   
   Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.
   
   Please refer to the wiki for details on using this class member.
       
   Getter Returns True is the B-spline object is rational (NURBS)
   
   Type bool

remove_knot(param, **kwargs)
   Removes the knot and updates the control points array and the knot vector.
   
   Keyword Arguments:
       
   • num: Number of knot removals. Default: 1
       
   Parameters param (float) – knot to be removed

render(**kwargs)
   Renders the curve using the visualization component.
   
   The visualization component must be set using vis property before calling this method.
   
   Keyword Arguments:
       
   • cpcolor: sets the color of the control points polygon
   • evalcolor: sets the color of the curve
• **bboxcolor**: sets the color of the bounding box

• **filename**: saves the plot with the input name

• **plot**: controls plot window visibility. *Default: True*

• **animate**: activates animation (if supported). *Default: False*

• **extras**: adds line plots to the figure. *Default: None*

*plot* argument is useful when you would like to work on the command line without any window context. If *plot* flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

*extras* argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
[  
    dict(  # line plot 1
        points=[[1, 2, 3], [4, 5, 6]],  # list of points
        name="My line Plot 1",  # name displayed on the legend
        color="red",  # color of the line plot
        size=6.5  # size of the line plot
    ),
    dict(  # line plot 2
        points=[[7, 8, 9], [10, 11, 12]],  # list of points
        name="My line Plot 2",  # name displayed on the legend
        color="navy",  # color of the line plot
        size=12.5  # size of the line plot
    )
]
```

*Returns*  the figure object

**reset** (**kwargs**)
Resets control points and/or evaluated points.

**Keyword Arguments:**

• **evalpts**: if True, then resets evaluated points

• **ctrlpts** if True, then resets control points

**reverse**()
Reverses the curve

**sample_size**
Sample size.
Sample size defines the number of evaluated points to generate. It also sets the *delta* property.

The following figure illustrates the working principles of sample size property:

\[
\begin{array}{c}
    \frac{u_{\text{start}}, \ldots, u_{\text{end}}}{\text{sample}}
\end{array}
\]

Please refer to the *wiki* for details on using this class member.

**Getter**  Gets sample size

**Setter**  Sets sample size
Type  int

**save** (*file_name*)
Saves the curve as a pickled file.

Deprecated since version 5.2.4: Use `exchange.export_json()` instead.

**Parameters**  
 *file_name* (*str*) – name of the file to be saved

**set_ctrlpts** (*ctrlpts, *args, **kwargs*)
Sets control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing (*x*, *y*, *z*) coordinates.

**Parameters**  
 *ctrlpts* (*list*) – input control points as a list of coordinates

**tangent** (*parpos, **kwargs*)
Evaluates the tangent vector of the curve at the given parametric position(s).

The *parpos* argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- normalize: normalizes the output vector. Default value is *True*.

**Parameters**  
 *parpos* (*float, list or tuple*) – parametric position(s) where the evaluation will be executed

**Returns**  
tangent vector as a tuple of the origin point and the vector components

**Return type**  
tuple

**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter**  
Gets the geometry type

**Type**  
str

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter**  
 Gets the visualization component

**Setter**  
Sets the visualization component

**Type**  
vis.VisAbstract

**weights**
Weights.
Note: Only available for rational spline geometries. Getter return None otherwise.

Please refer to the wiki for details on using this class member.

Getter  Gets the weights
Setter  Sets the weights

B-Spline Surface

class geomdl.BSpline.Surface(**kwargs)
    Bases: geomdl.abstract.Surface

Data storage and evaluation class for B-spline (non-rational) surfaces.

This class provides the following properties:

- type = spline
- id
- order_u
- order_v
- degree_u
- degree_v
- knotvector_u
- knotvector_v
- ctrlpts
- ctrlpts_size_u
- ctrlpts_size_v
- ctrlpts2d
- delta
- delta_u
- delta_v
- sample_size
- sample_size_u
- sample_size_v
- bbox
- name
- dimension
- vis
- evaluator
- tessellator
- rational
The following code segment illustrates the usage of Surface class:

```python
from geomdl import BSpline

# Create a BSpline surface instance (Bezier surface)
surf = BSpline.Surface()

# Set degrees
surf.degree_u = 3
surf.degree_v = 2

# Set control points
control_points = [[0, 0, 0], [0, 4, 0], [0, 8, -3],
                  [2, 0, 6], [2, 4, 0], [2, 8, 0],
                  [4, 0, 0], [4, 4, 0], [4, 8, 3],
                  [6, 0, 0], [6, 4, -3], [6, 8, 0]]
surf.set_ctrlpts(control_points, 4, 3)

# Set knot vectors
surf.knotvector_u = [0, 0, 0, 0, 1, 1, 1, 1]
surf.knotvector_v = [0, 0, 0, 1, 1, 1]

# Set evaluation delta (control the number of surface points)
surf.delta = 0.05

# Get surface points (the surface will be automatically evaluated)
surface_points = surf.evalpts
```

Keyword Arguments:

- `precision`: number of decimal places to round to. Default: 18
- `normalize_kv`: activates knot vector normalization. Default: True
- `find_span_func`: sets knot span search implementation. Default: helpers.find_span_linear()
- `insert_knot_func`: sets knot insertion implementation. Default: operations.insert_knot()
- `remove_knot_func`: sets knot removal implementation. Default: operations.remove_knot()

Please refer to the `abstract.Surface()` documentation for more details.

```python
add_trim(trim)
```

Adds a trim to the surface.

A trim is a 2-dimensional curve defined on the parametric domain of the surface. Therefore, x-coordinate of the trimming curve corresponds to u parametric direction of the surface and y-coordinate of the trimming curve corresponds to v parametric direction of the surface.

`trims` uses this method to add trims to the surface.

Parameters `trim` (abstract.Geometry) – surface trimming curve

```python
bbox
```

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.
**Getter**  Gets the bounding box
**Type**  tuple

**cpsize**
Number of control points in all parametric directions.

**Note:** This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the number of control points
**Setter**  Sets the number of control points
**Type**  list

**ctrlpts**
1-dimensional array of control points.

**Note:** The v index varies first. That is, a row of v control points for the first u value is found first. Then, the row of v control points for the next u value.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the control points
**Setter**  Sets the control points
**Type**  list

**ctrlpts2d**
2-dimensional array of control points.

The getter returns a tuple of 2D control points (weighted control points + weights if NURBS) in \([u]/[v]\) format. The rows of the returned tuple correspond to v-direction and the columns correspond to u-direction.

The following example can be used to traverse 2D control points:

```python
# Create a BSpline surface
surf_bs = BSpline.Surface()

# Do degree, control points and knot vector assignments here

# Each u includes a row of v values
for u in surf_bs.ctrlpts2d:
    # Each row contains the coordinates of the control points
    for v in u:
        print(str(v))  # will be something like (1.0, 2.0, 3.0)

# Create a NURBS surface
surf_nb = NURBS.Surface()

# Do degree, weighted control points and knot vector assignments here

# Each u includes a row of v values
for u in surf_nb.ctrlpts2d:
    # Each row contains the coordinates of the weighted control points
```
(continues on next page)
When using `NURBS.Surface` class, the output of `ctrlpts2d` property could be confusing since, `ctrlpts` always returns the unweighted control points, i.e. `ctrlpts` property returns 3D control points all divided by the weights and you can use `weights` property to access the weights vector, but `ctrlpts2d` returns the weighted ones plus weights as the last element. This difference is intentionally added for compatibility and interoperability purposes.

To explain this situation in a simple way:

- If you need the weighted control points directly, use `ctrlpts2d`
- If you need the control points and the weights separately, use `ctrlpts` and `weights`

**Note:** Please note that the setter doesn’t check for inconsistencies and using the setter is not recommended. Instead of the setter property, please use `set_ctrlpts()` function.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the control points as a 2-dimensional array in `[u][v]` format
- **Setter** Sets the control points as a 2-dimensional array in `[u][v]` format

**Type** list

`ctrlpts_size`

Total number of control points.

- **Getter** Gets the total number of control points
- **Type** int

`ctrlpts_size_u`

Number of control points for the u-direction.

- Please refer to the wiki for details on using this class member.
- **Getter** Gets number of control points for the u-direction
- **Setter** Sets number of control points for the u-direction

`ctrlpts_size_v`

Number of control points for the v-direction.

- Please refer to the wiki for details on using this class member.
- **Getter** Gets number of control points on the v-direction
- **Setter** Sets number of control points on the v-direction

`data`

Returns a dict which contains the geometry data.

- Please refer to the wiki for details on using this class member.

**degree**

 Degree for u- and v-directions

- **Getter** Gets the degree
- **Setter** Sets the degree
Type list

degree_u
Degree for the u-direction.
Please refer to the wiki for details on using this class member.

Getter Gets degree for the u-direction
Setter Sets degree for the u-direction
Type int

degree_v
Degree for the v-direction.
Please refer to the wiki for details on using this class member.

Getter Gets degree for the v-direction
Setter Sets degree for the v-direction
Type int

delta
Evaluation delta for both u- and v-directions.
Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta and sample_size properties correspond to the same variable with different descriptions. Therefore, setting delta will also set sample_size.
The following figure illustrates the working principles of the delta property:

\[ u_0, u_{\text{start}} + \delta, (u_{\text{start}} + \delta) + \delta, \ldots, u_{\text{end}} \]

Please refer to the wiki for details on using this class member.

Getter Gets evaluation delta as a tuple of values corresponding to u- and v-directions
Setter Sets evaluation delta for both u- and v-directions
Type float

delta_u
Evaluation delta for the u-direction.
Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_u and sample_size_u properties correspond to the same variable with different descriptions. Therefore, setting delta_u will also set sample_size_u.

Please refer to the wiki for details on using this class member.

Getter Gets evaluation delta for the u-direction
Setter Sets evaluation delta for the u-direction
Type float
**delta_v**

Evaluation delta for the v-direction.

Evaluation delta corresponds to the step size while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta_v` and `sample_size_v` properties correspond to the same variable with different descriptions. Therefore, setting `delta_v` will also set `sample_size_v`.

Please refer to the wiki for details on using this class member.

- **Getter** Gets evaluation delta for the v-direction
- **Setter** Sets evaluation delta for the v-direction
- **Type** float

**derivatives (u, v, order=0, **kwargs)**

Evaluates n-th order surface derivatives at the given (u, v) parameter pair.

- SKL[0][0] will be the surface point itself
- SKL[0][1] will be the 1st derivative w.r.t. v
- SKL[2][1] will be the 2nd derivative w.r.t. u and 1st derivative w.r.t. v

**Parameters**

- **u** (*float*) – parameter on the u-direction
- **v** (*float*) – parameter on the v-direction
- **order** (*integer*) – derivative order

**Returns** A list SKL, where SKL[k][l] is the derivative of the surface S(u,v) w.r.t. u k times and v l times

**Return type** list

**dimension**

Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the spatial dimension, e.g. 2D, 3D, etc.
- **Type** int

**domain**

Domain.

Domain is determined using the knot vector(s).

- **Getter** Gets the domain

**evalpts**

Evaluated points.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the coordinates of the evaluated points
- **Type** list
**evaluate** (**kwargs)
Evaluates the surface.

The evaluated points are stored in `evalpts` property.

**Keyword arguments:**
- `start_u`: start parameter on the u-direction
- `stop_u`: stop parameter on the u-direction
- `start_v`: start parameter on the v-direction
- `stop_v`: stop parameter on the v-direction

The `start_u`, `start_v`, and `stop_u` and `stop_v` parameters allow evaluation of a surface segment in the range `[start_u, stop_u][start_v, stop_v]` i.e. the surface will also be evaluated at the `stop_u` and `stop_v` parameter values.

The following examples illustrate the usage of the keyword arguments.

```python
# Start evaluating in range u=[0, 0.7] and v=[0.1, 1]
surf.evaluate(stop_u=0.7, start_v=0.1)

# Start evaluating in range u=[0, 1] and v=[0.1, 0.3]
surf.evaluate(start_v=0.1, stop_v=0.3)

# Get the evaluated points
surface_points = surf.evalpts
```

**evaluate_list** *(param_list)*
Evaluates the surface for a given list of (u, v) parameters.

**Parameters** `param_list` *(list, tuple)* – list of parameter pairs (u, v)

**Returns** evaluated surface point at the input parameter pairs

**Return type** tuple

**evaluate_single** *(param)*
Evaluates the surface at the input (u, v) parameter pair.

**Parameters** `param` *(list, tuple)* – parameter pair (u, v)

**Returns** evaluated surface point at the given parameter pair

**Return type** list

**evaluator**
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on `Evaluator` classes.

Please refer to the wiki for details on using this class member.

**Getter** Gets the current Evaluator instance

**Setter** Sets the Evaluator instance

**Type** `evaluators.AbstractEvaluator`

**faces**
Faces (triangles, quads, etc.) generated by the tessellation operation.

If the tessellation component is set to None, the result will be an empty list.
Getter  Gets the faces

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

  Getter  Gets the object ID
  Setter  Sets the object ID
  Type  int

**insert_knot** *(u=None, v=None, **kwargs)*
Inserts knot(s) on the u- or v-directions

**Keyword Arguments:**
- • **num_u**: Number of knot insertions on the u-direction. Default: 1
- • **num_v**: Number of knot insertions on the v-direction. Default: 1

**Parameters**
- • **u** *(float)* – knot to be inserted on the u-direction
- • **v** *(float)* – knot to be inserted on the v-direction

**knotvector**
Knot vector for u- and v-directions

  Getter  Gets the knot vector
  Setter  Sets the knot vector
  Type  list

**knotvector_u**
Knot vector for the u-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with normalize_kv=True argument.

Please refer to the wiki for details on using this class member.

  Getter  Gets knot vector for the u-direction
  Setter  Sets knot vector for the u-direction
  Type  list

**knotvector_v**
Knot vector for the v-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with normalize_kv=True argument.

Please refer to the wiki for details on using this class member.

  Getter  Gets knot vector for the v-direction
  Setter  Sets knot vector for the v-direction
  Type  list
load(file_name)

Loads the surface from a pickled file.

Deprecated since version 5.2.4: Use exchange.import_json() instead.

Parameters file_name (str) – name of the file to be loaded

name

Object name (as a string)

Please refer to the wiki for details on using this class member.

Getter Gets the object name

Setter Sets the object name

Type str

normal(parpos, **kwargs)

Evaluates the normal vector of the surface at the given parametric position(s).

The param argument can be

• a float value for evaluation at a single parametric position

• a list of float values for evaluation at the multiple parametric positions

The parametric positions should be a pair of (u,v) values. The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

• normalize: normalizes the output vector. Default value is True.

Parameters parpos (list or tuple) – parametric position(s) where the evaluation will be executed

Returns an array containing “point” and “vector” pairs

Return type tuple

opt

Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
gemPERT = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
gemPERT = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(gemPERT)  # will print: {'face_id': 4, 'contents': 'data values'}
del gemPERT  # deletes the contents of the hash map
print(gemPERT)  # will print: {}

gemPERT = ["body_id", 1]  # creates "body_id" key and sets its value to 1

gemPERT = ["body_id", 12]  # changes the value of "body_id" to 12
print(gemPERT)  # will print: {'body_id': 12}

gemPERT = ["body_id", None]  # deletes "body_id"
print(gemPERT)  # will print: {}```

Please refer to the wiki for details on using this class member.
Getter  Gets the dict
Setter  Adds key and value pair to the dict
Deleter  Deletes the contents of the dict

\texttt{opt\_get} (value)
Safely query for the value from the \texttt{opt} property.

\begin{description}
\item[Parameters] value (str) – a key in the \texttt{opt} property
\item[Returns] the corresponding value, if the key exists. \texttt{None}, otherwise.
\end{description}

\texttt{order\_u}
Order for the u-direction.
Defined as order = degree + 1
Please refer to the \texttt{wiki} for details on using this class member.

\begin{description}
\item[Getter] Gets order for the u-direction
\item[Setter] Sets order for the u-direction
\item[Type] int
\end{description}

\texttt{order\_v}
Order for the v-direction.
Defined as order = degree + 1
Please refer to the \texttt{wiki} for details on using this class member.

\begin{description}
\item[Getter] Gets surface order for the v-direction
\item[Setter] Sets surface order for the v-direction
\item[Type] int
\end{description}

\texttt{pdimension}
Parametric dimension.
Please refer to the \texttt{wiki} for details on using this class member.

\begin{description}
\item[Getter] Gets the parametric dimension
\item[Type] int
\end{description}

\texttt{range}
Domain range.

\begin{description}
\item[Getter] Gets the range
\end{description}

\texttt{rational}
Defines the rational and non-rational B-spline shapes.
Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.
Please refer to the \texttt{wiki} for details on using this class member.

\begin{description}
\item[Getter] Returns True is the B-spline object is rational (NURBS)
\item[Type] bool
\end{description}

\texttt{remove\_knot} (u=None, v=None, **kwargs)
Inserts knot(s) on the u- or v-directions
Keyword Arguments:

- `num_u`: Number of knot removals on the u-direction. Default: 1
- `num_v`: Number of knot removals on the v-direction. Default: 1

Parameters

- `u (float)`: knot to be removed on the u-direction
- `v (float)`: knot to be removed on the v-direction

`render (**kwargs)`

Renders the surface using the visualization component.

The visualization component must be set using `vis` property before calling this method.

Keyword Arguments:

- `cpcolor`: sets the color of the control points grid
- `evalcolor`: sets the color of the surface
- `trimcolor`: sets the color of the trim curves
- `filename`: saves the plot with the input name
- `plot`: controls plot window visibility. Default: True
- `animate`: activates animation (if supported). Default: False
- `extras`: adds line plots to the figure. Default: None
- `colormap`: sets the colormap of the surface

The `plot` argument is useful when you would like to work on the command line without any window context. If `plot` flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

`extras` argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
[
    dict(  # line plot 1
        points=[[1, 2, 3], [4, 5, 6]],  # list of points
        name="My line Plot 1",  # name displayed on the legend
        color="red",  # color of the line plot
        size=6.5  # size of the line plot
    ),
    dict(  # line plot 2
        points=[[7, 8, 9], [10, 11, 12]],  # list of points
        name="My line Plot 2",  # name displayed on the legend
        color="navy",  # color of the line plot
        size=12.5  # size of the line plot
    )
]
```

Please note that `colormap` argument can only work with visualization classes that support colormaps. As an example, please see `VisMPL.VisSurfTriangle()` class documentation. This method expects a single colormap input.

Returns the figure object
**reset(**kwargs**)

Resets control points and/or evaluated points.

**Keyword Arguments:**

- **evalpts**: if True, then resets evaluated points
- **ctrlpts**: if True, then resets control points

**sample_size**

Sample size for both u- and v-directions.

Sample size defines the number of surface points to generate. It also sets the `delta` property.

The following figure illustrates the working principles of sample size property:

\[
\begin{align*}
\{u_{start, \ldots, u_{end}}\} \\
\text{sample}
\end{align*}
\]

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size as a tuple of values corresponding to u- and v-directions

**Setter** Sets sample size for both u- and v-directions

**Type** int

**sample_size_u**

Sample size for the u-direction.

Sample size defines the number of surface points to generate. It also sets the `delta_u` property.

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the u-direction

**Setter** Sets sample size for the u-direction

**Type** int

**sample_size_v**

Sample size for the v-direction.

Sample size defines the number of surface points to generate. It also sets the `delta_v` property.

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the v-direction

**Setter** Sets sample size for the v-direction

**Type** int

**save**(file_name)

Saves the surface as a pickled file.

Deprecated since version 5.2.4: Use `exchange.export_json()` instead.

**Parameters** file_name (str) – name of the file to be saved

**set_ctrlpts**(ctrlpts, *args, **kwargs)

Sets the control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing (x, y, z) coordinates.
This method also generates 2D control points in \([u/v]\) format which can be accessed via `ctrlpts2d`.

**Note:** The \(v\) index varies first. That is, a row of \(v\) control points for the first \(u\) value is found first. Then, the row of \(v\) control points for the next \(u\) value.

**Parameters** `ctrlpts` (list) – input control points as a list of coordinates

**tangent** (`parpos`, **kwargs)

Evaluates the tangent vectors of the surface at the given parametric position(s).

The `param` argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The parametric positions should be a pair of (\(u,v\)) values. The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- `normalize`: normalizes the output vector. Default value is `True`.

**Parameters** `parpos` (list or tuple) – parametric position(s) where the evaluation will be executed

**Returns** an array containing “point” and “vector”s on \(u\)- and \(v\)-directions, respectively

**Return type** tuple

**tessellate** (**kwargs)

Tessellates the surface.

Keyword arguments are directly passed to the tessellation component.

**tessellator**

Tessellation component.

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets the tessellation component
- **Setter** Sets the tessellation component

**transpose** ()

Transposes the surface by swapping \(u\) and \(v\) parametric directions.

**trims**

Curves for trimming the surface.

Surface trims are 2-dimensional curves which are introduced on the parametric space of the surfaces. Trim curves can be a spline curve, an analytic curve or a 2-dimensional freeform shape. To visualize the trimmed surfaces, you need to use a tessellator that supports trimming. The following code snippet illustrates changing the default surface tessellator to the trimmed surface tessellator, `tessellate.TrimTessellate`.

```python
from geomdl import tessellate

# Assuming that "surf" variable stores the surface instance
surf.tessellator = tessellate.TrimTessellate()
```
In addition, using `trims` initialization argument of the visualization classes, trim curves can be visualized together with their underlying surfaces. Please refer to the visualization configuration class initialization arguments for more details.

Please refer to the wiki for details on using this class member.

**Getter** Gets the array of trim curves

**Setter** Sets the array of trim curves

**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter** Gets the geometry type

**Type** `str`

**vertices**
Vertices generated by the tessellation operation.

If the tessellation component is set to `None`, the result will be an empty list.

**Getter** Gets the vertices

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter** Gets the visualization component

**Setter** Sets the visualization component

**Type** `vis.VisAbstract`

**weights**
Weights.

**Note:** Only available for rational spline geometries. Getter return `None` otherwise.

Please refer to the wiki for details on using this class member.

**Getter** Gets the weights

**Setter** Sets the weights

**B-Spline Volume**

New in version 5.0.

```python
class geomdl.BSpline.Volume(**kwargs)
    Bases: geomdl.abstract.Volume

    Data storage and evaluation class for B-spline (non-rational) volumes.
```

This class provides the following properties:

- `type = spline`
- `id`
- `order_u`
• order_v
• order_w
• degree_u
• degree_v
• degree_w
• knotvector_u
• knotvector_v
• knotvector_w
• ctrlpts
• ctrlpts_size_u
• ctrlpts_size_v
• ctrlpts_size_w
• delta
• delta_u
• delta_v
• delta_w
• sample_size
• sample_size_u
• sample_size_v
• sample_size_w
• bbox
• name
• dimension
• vis
• evaluator
• rational

Keyword Arguments:

• precision: number of decimal places to round to. Default: 18
• normalize_kv: activates knot vector normalization. Default: True
• find_span_func: sets knot span search implementation. Default: helpers.find_span_linear()
• insert_knot_func: sets knot insertion implementation. Default: operations.insert_knot()
• remove_knot_func: sets knot removal implementation. Default: operations.remove_knot()

Please refer to the abstract.Volume() documentation for more details.
add_trim(trim)
    Adds a trim to the volume.

    trims uses this method to add trims to the volume.

    Parameters trim(abstract.Surface) – trimming surface

bbox
    Bounding box.

    Evaluates the bounding box and returns the minimum and maximum coordinates.

    Please refer to the wiki for details on using this class member.

    Getter Gets the bounding box
    Type  tuple

csize
    Number of control points in all parametric directions.

    Note: This is an expert property for getting and setting control point size(s) of the geometry.

    Please refer to the wiki for details on using this class member.

    Getter Gets the number of control points
    Setter Sets the number of control points
    Type  list

cpts
    1-dimensional array of control points.

    Please refer to the wiki for details on using this class member.

    Getter Gets the control points
    Setter Sets the control points
    Type  list

cpts_size
    Total number of control points.

    Getter Gets the total number of control points
    Type  int

cpts_size_u
    Number of control points for the u-direction.

    Please refer to the wiki for details on using this class member.

    Getter Gets number of control points for the u-direction
    Setter Sets number of control points for the u-direction

cpts_size_v
    Number of control points for the v-direction.

    Please refer to the wiki for details on using this class member.

    Getter Gets number of control points for the v-direction
    Setter Sets number of control points for the v-direction
**ctrlpts_size_w**
Number of control points for the w-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets number of control points for the w-direction
- **Setter** Sets number of control points for the w-direction

**data**
Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**degree**
Degree for u-, v- and w-directions

- **Getter** Gets the degree
- **Setter** Sets the degree
- **Type** list

**degree_u**
Degree for the u-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets degree for the u-direction
- **Setter** Sets degree for the u-direction
- **Type** int

**degree_v**
Degree for the v-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets degree for the v-direction
- **Setter** Sets degree for the v-direction
- **Type** int

**degree_w**
Degree for the w-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets degree for the w-direction
- **Setter** Sets degree for the w-direction
- **Type** int

**delta**
Evaluation delta for u-, v- and w-directions.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta and sample_size properties correspond to the same variable with different descriptions. Therefore, setting delta will also set sample_size.
The following figure illustrates the working principles of the delta property:

\[ [u_0, u_{\text{start}} + \delta, (u_{\text{start}} + \delta) + \delta, \ldots, u_{\text{end}}] \]

Please refer to the wiki for details on using this class member.

**Getter**  Gets evaluation delta as a tuple of values corresponding to u-, v- and w-directions

**Setter**  Sets evaluation delta for u-, v- and w-directions

**Type**  float

**delta_u**

Evaluation delta for the u-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_u and sample_size_u properties correspond to the same variable with different descriptions. Therefore, setting delta_u will also set sample_size_u.

Please refer to the wiki for details on using this class member.

**Getter**  Gets evaluation delta for the u-direction

**Setter**  Sets evaluation delta for the u-direction

**Type**  float

**delta_v**

Evaluation delta for the v-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_v and sample_size_v properties correspond to the same variable with different descriptions. Therefore, setting delta_v will also set sample_size_v.

Please refer to the wiki for details on using this class member.

**Getter**  Gets evaluation delta for the v-direction

**Setter**  Sets evaluation delta for the v-direction

**Type**  float

**delta_w**

Evaluation delta for the w-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_w and sample_size_w properties correspond to the same variable with different descriptions. Therefore, setting delta_w will also set sample_size_w.

Please refer to the wiki for details on using this class member.

**Getter**  Gets evaluation delta for the w-direction

**Setter**  Sets evaluation delta for the w-direction

**Type**  float
**dimension**
Spatial dimension.
Spatial dimension will be automatically estimated from the first element of the control points array.
Please refer to the wiki for details on using this class member.

**Getter** Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

**domain**
Domain.
Domain is determined using the knot vector(s).

**Getter** Gets the domain

**evalpts**
Evaluated points.
Please refer to the wiki for details on using this class member.

**Getter** Gets the coordinates of the evaluated points

**Type** list

**evaluate(**
**kwargs**
)**
Evaluates the volume.
The evaluated points are stored in `evalpts` property.

**Keyword arguments:**
- `start_u`: start parameter on the u-direction
- `stop_u`: stop parameter on the u-direction
- `start_v`: start parameter on the v-direction
- `stop_v`: stop parameter on the v-direction
- `start_w`: start parameter on the w-direction
- `stop_w`: stop parameter on the w-direction

**evaluate_list**(`param_list`)
Evaluates the volume for a given list of (u, v, w) parameters.

**Parameters** `param_list`(`list`, `tuple`) – list of parameters in format (u, v, w)

**Returns** evaluated surface point at the input parameter pairs

**Return type** tuple

**evaluate_single**(`param`)
Evaluates the volume at the input (u, v, w) parameter.

**Parameters** `param`(`list`, `tuple`) – parameter (u, v, w)

**Returns** evaluated surface point at the given parameter pair

**Return type** list

**evaluator**
Evaluator instance.
Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on `Evaluator` classes.

16.1. User API
Please refer to the wiki for details on using this class member.

**Getter**  
Gets the current Evaluator instance

**Setter**  
Sets the Evaluator instance

**Type**  
evaluators.AbstractEvaluator

### id

Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter**  
Gets the object ID

**Setter**  
Sets the object ID

**Type**  
int

**insert_knot** *(u=None, v=None, w=None, **kwargs)*  
Inserts knot(s) on the u-, v- and w-directions

**Keyword Arguments:**

- **num_u**: Number of knot insertions on the u-direction. *Default: 1*
- **num_v**: Number of knot insertions on the v-direction. *Default: 1*
- **num_w**: Number of knot insertions on the w-direction. *Default: 1*

**Parameters**

- **u** *(float)* – knot to be inserted on the u-direction
- **v** *(float)* – knot to be inserted on the v-direction
- **w** *(float)* – knot to be inserted on the w-direction

**knotvector**

Knot vector for u-, v- and w-directions

**Getter**  
Gets the knot vector

**Setter**  
Sets the knot vector

**Type**  
list

**knotvector_u**

Knot vector for the u-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

**Getter**  
Gets knot vector for the u-direction

**Setter**  
Sets knot vector for the u-direction

**Type**  
list

**knotvector_v**

Knot vector for the v-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.
### knotvector_w

Knot vector for the w-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

- **Getter** Gets knot vector for the w-direction
- **Setter** Sets knot vector for the w-direction

#### load

Loads the volume from a pickled file.

**Parameters**

- `file_name` (str) – name of the file to be loaded

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name

#### opt

Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {} 
```

Please refer to the wiki for details on using this class member.

- **Getter** Gets the dict
- **Setter** Adds key and value pair to the dict
Deleter Deletes the contents of the dict

**opt_get**(*value*)
Safely query for the value from the *opt* property.

**Parameters**

**value** (*str*) – a key in the *opt* property

**Returns**

the corresponding value, if the key exists. None, otherwise.

**order_u**
Order for the u-direction.

Defined as \( \text{order} = \text{degree} + 1 \)

Please refer to the wiki for details on using this class member.

**Getter**

Gets the surface order for u-direction

**Setter**

Sets the surface order for u-direction

**Type**

int

**order_v**
Order for the v-direction.

Defined as \( \text{order} = \text{degree} + 1 \)

Please refer to the wiki for details on using this class member.

**Getter**

Gets the surface order for v-direction

**Setter**

Sets the surface order for v-direction

**Type**

int

**order_w**
Order for the w-direction.

Defined as \( \text{order} = \text{degree} + 1 \)

Please refer to the wiki for details on using this class member.

**Getter**

Gets the surface order for v-direction

**Setter**

Sets the surface order for v-direction

**Type**

int

**pdimension**
Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the parametric dimension

**Type**

int

**range**
Domain range.

**Getter**

Gets the range

**rational**
Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.
Please refer to the wiki for details on using this class member.

**Getter**
Returns True if the B-spline object is rational (NURBS)

**Type**
`bool`

```python
remove_knot(u=None, v=None, w=None, **kwargs)
```

Inserts knot(s) on the u-, v- and w-directions

**Keyword Arguments:**
- `num_u`: Number of knot removals on the u-direction. Default: 1
- `num_v`: Number of knot removals on the v-direction. Default: 1
- `num_w`: Number of knot removals on the w-direction. Default: 1

**Parameters**
- `u` (`float`) – knot to be removed on the u-direction
- `v` (`float`) – knot to be removed on the v-direction
- `w` (`float`) – knot to be removed on the w-direction

```python
render(**kwargs)
```

Renders the volume using the visualization component.

The visualization component must be set using `vis` property before calling this method.

**Keyword Arguments:**
- `cpcolor`: sets the color of the control points
- `evalcolor`: sets the color of the volume
- `filename`: saves the plot with the input name
- `plot`: controls plot window visibility. Default: True
- `animate`: activates animation (if supported). Default: False
- `grid_size`: grid size for voxelization. Default: (8, 8, 8)
- `use_cubes`: use cube voxels instead of cuboid ones. Default: False
- `num_procs`: number of concurrent processes for voxelization. Default: 1

The `plot` argument is useful when you would like to work on the command line without any window context. If `plot` flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

`extras` argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
[
    dict(  # line plot 1
        points=[[1, 2, 3], [4, 5, 6]],  # list of points
        name="My line Plot 1",  # name displayed on the legend
        color="red",  # color of the line plot
        size=6.5  # size of the line plot
    ),
    dict(  # line plot 2
        points=[[7, 8, 9], [10, 11, 12]],  # list of points
        name="My line Plot 2",  # name displayed on the legend
        color="blue",  # color of the line plot
        size=4.5  # size of the line plot
    )
]
```

(continues on next page)
name="My line Plot 2",  # name displayed on the legend
color="navy",  # color of the line plot
size=12.5  # size of the line plot
}
]

Returns the figure object

reset(**kwargs)
Resets control points and/or evaluated points.

Keyword Arguments:

• evalpts: if True, then resets evaluated points
• ctrlpts if True, then resets control points

sample_size
Sample size for both u- and v-directions.
Sample size defines the number of surface points to generate. It also sets the delta property.
The following figure illustrates the working principles of sample size property:

\[ \underbrace{u_{start}, \ldots, u_{end}}_{n_{sample}} \]

Please refer to the wiki for details on using this class member.

  Getter  Gets sample size as a tuple of values corresponding to u-, v- and w-directions
  Setter  Sets sample size value for both u-, v- and w-directions
  Type    int

sample_size_u
Sample size for the u-direction.
Sample size defines the number of evaluated points to generate. It also sets the delta_u property.
Please refer to the wiki for details on using this class member.

  Getter  Gets sample size for the u-direction
  Setter  Sets sample size for the u-direction
  Type    int

sample_size_v
Sample size for the v-direction.
Sample size defines the number of evaluated points to generate. It also sets the delta_v property.
Please refer to the wiki for details on using this class member.

  Getter  Gets sample size for the v-direction
  Setter  Sets sample size for the v-direction
  Type    int
**sample_size_w**
Sample size for the w-direction.

Sample size defines the number of evaluated points to generate. It also sets the \texttt{delta_w} property. Please refer to the wiki for details on using this class member.

- **Getter** Get sample size for the w-direction
- **Setter** Sets sample size for the w-direction

**Type** \texttt{int}

**save** \texttt{(file_name)}
Saves the volume as a pickled file.

Deprecated since version 5.2.4: Use \texttt{exchange.export_json()} instead.

- **Parameters** \texttt{file_name} (\texttt{str}) – name of the file to be saved

**set_ctrlpts** \texttt{(ctrlpts, \*args, \**kwargs)}
Sets the control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing \((x, y, z)\) coordinates.

- **Parameters**
  - \texttt{ctrlpts} (\texttt{list}) – input control points as a list of coordinates
  - \texttt{args} (\texttt{tuple[int, int, int]}) – number of control points corresponding to each parametric dimension

**trims**
Trimming surfaces.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the array of trim surfaces
- **Setter** Sets the array of trim surfaces

**type**
Geometry type

Please refer to the wiki for details on using this class member.

- **Getter** Gets the geometry type
- **Type** \texttt{str}

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the visualization component
- **Setter** Sets the visualization component
- **Type** \texttt{vis.VisAbstract}

**weights**
Weights.
Note: Only available for rational spline geometries. Getter return None otherwise.

Please refer to the wiki for details on using this class member.

**Getter** Gets the weights

**Setter** Sets the weights

### 16.1.2 NURBS Geometry

NURBS module provides data storage and evaluation functions for rational spline geometries.

### Inheritance Diagram

![Inheritance Diagram](image)

### NURBS Curve

```python
class geomdl.NURBS.Curve(**kwargs)
    Bases: geomdl.BSpline.Curve
```

Data storage and evaluation class for n-variate NURBS (rational) curves.

The rational shapes have some minor differences between the non-rational ones. This class is designed to operate with weighted control points (Pw) as described in *The NURBS Book* by Piegl and Tiller. Therefore, it provides a different set of properties (i.e. getters and setters):

- `ctrlptsw`: 1-dimensional array of weighted control points
- `ctrlpts`: 1-dimensional array of control points
- `weights`: 1-dimensional array of weights

You may also use `set_ctrlpts()` function which is designed to work with all types of control points.

This class provides the following properties:

- `order`
- `degree`
- `knotvector`
- `ctrlptsw`
- `ctrlpts`
- `weights`
The following code segment illustrates the usage of Curve class:

```python
from geomdl import NURBS

# Create a 3-dimensional B-spline Curve
curve = NURBS.Curve()

# Set degree
curve.degree = 3

# Set control points (weights vector will be 1 by default)
# Use curve.ctrlpts_w is if you are using homogeneous points as Pw
curve.ctrlpts = [[10, 5, 10], [10, 20, -30], [40, 10, 25], [-10, 5, 0]]

# Set knot vector
curve.knotvector = [0, 0, 0, 0, 1, 1, 1, 1]

# Set evaluation delta (controls the number of curve points)
curve.delta = 0.05

# Get curve points (the curve will be automatically evaluated)
curve_points = curve.evalpts
```

**Keyword Arguments:**

- **precision**: number of decimal places to round to. *Default: 18*
- **normalize_kv**: activates knot vector normalization. *Default: True*
- **find_span_func**: sets knot span search implementation. *Default: helpers.find_span_linear()*
- **insert_knot_func**: sets knot insertion implementation. *Default: operations.insert_knot()*
- **remove_knot_func**: sets knot removal implementation. *Default: operations.remove_knot()*

Please refer to the `abstract.Curve()` documentation for more details.

**bbox**

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the bounding box
- **Type** tuple
binormal \( (parpos, **kwargs) \)
Evaluates the binormal vector of the curve at the given parametric position(s).

The \( parpos \) argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- \( \text{normalize} \): normalizes the output vector. Default value is \( True \).

Parameters \( \text{parpos} (\text{float, list or tuple}) \) – parametric position(s) where the evaluation will be executed

Returns binormal vector as a tuple of the origin point and the vector components

Return type tuple

cpsize
Number of control points in all parametric directions.

Note: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

- \( \text{Getter} \) Gets the number of control points
- \( \text{Setter} \) Sets the number of control points
- \( \text{Type} \) list
ctrlpts
Control points (P).

Please refer to the wiki for details on using this class member.

- \( \text{Getter} \) Gets unweighted control points. Use \( \text{weights} \) to get weights vector.
- \( \text{Setter} \) Sets unweighted control points
- \( \text{Type} \) list

ctrlpts_size
Total number of control points.

- \( \text{Getter} \) Gets the total number of control points
- \( \text{Type} \) int

ctrlptsw
Weighted control points (Pw).

Weighted control points are in \((x*w, y*w, z*w, w)\) format; where \(x,y,z\) are the coordinates and \(w\) is the weight.

Please refer to the wiki for details on using this class member.

- \( \text{Getter} \) Gets the weighted control points
- \( \text{Setter} \) Sets the weighted control points
**data**

Returns a dict which contains the geometry data.

Please refer to the [wiki](#) for details on using this class member.

**degree**

Degree.

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets the degree
- **Setter** Sets the degree
- **Type** int

**delta**

Evaluation delta.

Evaluation delta corresponds to the *step size* while `evaluate` function iterates on the knot vector to generate curve points. Decreasing step size results in generation of more curve points. Therefore; smaller the delta value, smoother the curve.

The following figure illustrates the working principles of the delta property:

\[ u_{\text{start}}, u_{\text{start}} + \delta, (u_{\text{start}} + \delta) + \delta, \ldots, u_{\text{end}} \]

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets the delta value
- **Setter** Sets the delta value
- **Type** float

**derivatives** (*u*, *order=0*, **kwargs)

Evaluates n-th order curve derivatives at the given parameter value.

The output of this method is list of n-th order derivatives. If *order* is 0, then it will only output the evaluated point. Similarly, if *order* is 2, then it will output the evaluated point, 1st derivative and the 2nd derivative. For instance;

```python
# Assuming a curve (crv) is defined on a parametric domain [0.0, 1.0]
# Let's take the curve derivative at the parametric position u = 0.35
ders = crv.derivatives(u=0.35, order=2)
ders[0] # evaluated point, equal to crv.evaluate_single(0.35)
ders[1] # 1st derivative at u = 0.35
ders[2] @ 2nd derivative at u = 0.35
```

- **Parameters**
  - *u* (*float*) – parameter value
  - *order* (*int*) – derivative order

- **Returns** a list containing up to *order*-th derivative of the curve

- **Return type** list

**dimension**

Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the [wiki](#) for details on using this class member.
Getter Gets the spatial dimension, e.g. 2D, 3D, etc.

Type int
domain
Domain.
Domain is determined using the knot vector(s).

Getter Gets the domain
evalpts
Evaluated points.
Please refer to the wiki for details on using this class member.

Getter Gets the coordinates of the evaluated points

Type list
evaluate(**kwargs)
Evaluates the curve.
The evaluated points are stored in evalpts property.

Keyword arguments:
• start: start parameter
• stop: stop parameter

The start and stop parameters allow evaluation of a curve segment in the range [start, stop], i.e. the curve will also be evaluated at the stop parameter value.

The following examples illustrate the usage of the keyword arguments.

```python
# Start evaluating from u=0.2 to u=1.0
curve.evaluate(start=0.2)

# Start evaluating from u=0.0 to u=0.7
curve.evaluate(stop=0.7)

# Start evaluating from u=0.1 to u=0.5
curve.evaluate(start=0.1, stop=0.5)

# Get the evaluated points
curve_points = curve.evalpts
```
evaluate_list(param_list)
Evaluates the curve for an input range of parameters.

Parameters param_list (list, tuple) – list of parameters

Returns evaluated surface points at the input parameters

Return type list
evaluate_single(param)
Evaluates the curve at the input parameter.

Parameters param (float) – parameter

Returns evaluated surface point at the given parameter

Return type list
**evaluator**
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the current Evaluator instance
- **Setter** Sets the Evaluator instance
- **Type** `evaluators.AbstractEvaluator`

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object ID
- **Setter** Sets the object ID
- **Type** `int`

**insert_knot** *(param, **kwargs)*
Inserts the knot and updates the control points array and the knot vector.

**Keyword Arguments:**

- **num**: Number of knot insertions. *Default*: 1

**Parameters**
- **param** *(float)* – knot to be inserted

**knotvector**
Knot vector.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the knot vector
- **Setter** Sets the knot vector
- **Type** `list`

**load** *(file_name)*
Loads the curve from a pickled file.

*Deprecated since version 5.2.4: Use `exchange.import_json()` instead.*

**Parameters**
- **file_name** *(str)* – name of the file to be loaded

**name**
Object name (as a string)

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name
- **Type** `str`

normal (parpos, *kwargs)
Evaluates the normal to the tangent vector of the curve at the given parametric position(s).

The parpos argument can be
• a float value for evaluation at a single parametric position
• a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:
• normalize: normalizes the output vector. Default value is True.

**Parameters**
parpos (float, list or tuple) – parametric position(s) where the evaluation will be executed

**Returns**
normal vector as a tuple of the origin point and the vector components

**Return type**
tuple

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in *key => value* format, where *key* is string, *value* is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
geom[opt] = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}
geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom[opt] = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom[opt])  # will print: {'body_id': 12}
geom[opt] = ['body_id', None]  # deletes "body_id"
print(geom[opt])  # will print: {}
```

Please refer to the [wiki](#) for details on using this class member.

**Getter**
 Gets the dict

**Setter**
 Adds key and value pair to the dict

**Deleter**
 Deletes the contents of the dict

opt_get (value)
Safely query for the value from the opt property.

**Parameters**
value (str) – a key in the opt property

**Returns**
the corresponding value, if the key exists. None, otherwise.

order
Order.

Defined as order = degree + 1
Please refer to the wiki for details on using this class member.

**Getter** Gets the order

**Setter** Sets the order

**Type** int

**pdimension**

Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter** Gets the parametric dimension

**Type** int

**range**

Domain range.

**Getter** Gets the range

**rational**

Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

**Getter** Returns True is the B-spline object is rational (NURBS)

**Type** bool

**remove_knot** *(param, **kwargs)*

Removes the knot and updates the control points array and the knot vector.

**Keyword Arguments:**

* num: Number of knot removals. Default: 1

**Parameters** param *(float)* – knot to be removed

**render** *(**kwargs)*

Renders the curve using the visualization component

The visualization component must be set using vis property before calling this method.

**Keyword Arguments:**

* cpcolor: sets the color of the control points polygon
* evalcolor: sets the color of the curve
* bboxcolor: sets the color of the bounding box
* filename: saves the plot with the input name
* plot: controls plot window visibility. Default: True
* animate: activates animation (if supported). Default: False
* extras: adds line plots to the figure. Default: None
plot argument is useful when you would like to work on the command line without any window context. If plot flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

extras argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
[  
    dict(  # line plot 1
        points=[[1, 2, 3], [4, 5, 6]],  # list of points
        name="My line Plot 1",  # name displayed on the legend
        color="red",  # color of the line plot
        size=6.5  # size of the line plot
    ),
    dict(  # line plot 2
        points=[[7, 8, 9], [10, 11, 12]],  # list of points
        name="My line Plot 2",  # name displayed on the legend
        color="navy",  # color of the line plot
        size=12.5  # size of the line plot
    )
]
```

Returns the figure object

reset (**kwargs)
Resets control points and/or evaluated points.

Keyword Arguments:
  • evalpts: if True, then resets evaluated points
  • ctrlpts if True, then resets control points

reverse()
Reverses the curve

sample_size
Sample size.

Sample size defines the number of evaluated points to generate. It also sets the delta property.

The following figure illustrates the working principles of sample size property:

\[
[u_{start}, \ldots, u_{end}] \quad \underbrace{\text{sample}}_{n_{sample}}
\]

Please refer to the wiki for details on using this class member.

  Getter Gets sample size
  Setter Sets sample size
  Type int

save (file_name)
Saves the curve as a pickled file.

Deprecated since version 5.2.4: Use exchange.export_json() instead.

  Parameters file_name (str) – name of the file to be saved
**set_ctrlpts** *(ctrlpts, *args, **kwargs)*

Sets control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing \((x, y, z)\) coordinates.

**Parameters**

- **ctrlpts** *(list)* – input control points as a list of coordinates

**tangent** *(parpos, **kwargs)*

Evaluates the tangent vector of the curve at the given parametric position(s).

The `parpos` argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- `normalize`: normalizes the output vector. Default value is `True`.

**Parameters**

- **parpos** *(float, list or tuple)* – parametric position(s) where the evaluation will be executed

**Returns**

tangent vector as a tuple of the origin point and the vector components

**Return type**

tuple

---

**type**

Geometry type

Please refer to the wiki for details on using this class member.

**Getter**

Gets the geometry type

**Type**

str

**vis**

Visualization component.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the visualization component

**Setter**

Sets the visualization component

**Type**

vis.VisAbstract

**weights**

Weights vector.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the weights vector

**Setter**

Sets the weights vector

**Type**

list
NURBS Surface

class geomdl.NURBS.Surface(**kwargs)
    Bases: geomdl.BSpline.Surface

Data storage and evaluation class for NURBS (rational) surfaces.

The rational shapes have some minor differences between the non-rational ones. This class is designed to operate with weighted control points (Pw) as described in *The NURBS Book* by Piegl and Tiller. Therefore, it provides a different set of properties (i.e. getters and setters):

- **ctrlptsw**: 1-dimensional array of weighted control points
- **ctrlpts2d**: 2-dimensional array of weighted control points
- **ctrlpts**: 1-dimensional array of control points
- **weights**: 1-dimensional array of weights

You may also use `set_ctrlpts()` function which is designed to work with all types of control points.

This class provides the following properties:

- **order_u**
- **order_v**
- **degree_u**
- **degree_v**
- **knotvector_u**
- **knotvector_v**
- **ctrlptsw**
- **ctrlpts**
- **weights**
- **ctrlpts_size_u**
- **ctrlpts_size_v**
- **ctrlpts2d**
- **delta**
- **delta_u**
- **delta_v**
- **sample_size**
- **sample_size_u**
- **sample_size_v**
- **bbox**
- **name**
- **dimension**
- **vis**
- **evaluator**
- **tessellator**
The following code segment illustrates the usage of Surface class:

```python
from geomdl import NURBS

# Create a NURBS surface instance
surf = NURBS.Surface()

# Set degrees
surf.degree_u = 3
surf.degree_v = 2

# Set control points (weights vector will be 1 by default)
control_points = [[0, 0, 0], [0, 4, 0], [0, 8, -3],
                  [2, 0, 6], [2, 4, 0], [2, 8, 0],
                  [4, 0, 0], [4, 4, 0], [4, 8, 3],
                  [6, 0, 0], [6, 4, -3], [6, 8, 0]]
surf.set_ctrlpts(control_points, 4, 3)

# Set knot vectors
surf.knotvector_u = [0, 0, 0, 0, 1, 1, 1, 1]
surf.knotvector_v = [0, 0, 0, 1, 1]

# Set evaluation delta (control the number of surface points)
surf.delta = 0.05

# Get surface points (the surface will be automatically evaluated)
surface_points = surf.evalpts
```

Keyword Arguments:

- **precision**: number of decimal places to round to. Default: 18
- **normalize_kv**: activates knot vector normalization. Default: True
- **find_span_func**: sets knot span search implementation. Default: helpers.find_span_linear()
- **insert_knot_func**: sets knot insertion implementation. Default: operations.insert_knot()
- **remove_knot_func**: sets knot removal implementation. Default: operations.remove_knot()

Please refer to the `abstract.Surface()` documentation for more details.

**add_trim**(trim)

Adds a trim to the surface.

A trim is a 2-dimensional curve defined on the parametric domain of the surface. Therefore, x-coordinate of the trimming curve corresponds to u parametric direction of the surface and y-coordinate of the trimming curve corresponds to v parametric direction of the surface.

`trims` uses this method to add trims to the surface.

Parameters **trim**(abstract.Geometry) – surface trimming curve

**bbox**

Bounding box.
Evaluates the bounding box and returns the minimum and maximum coordinates. Please refer to the wiki for details on using this class member.

**Getter**  Gets the bounding box

**Type**  tuple

cpsize
Number of control points in all parametric directions.

**Note:** This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the number of control points

**Setter**  Sets the number of control points

**Type**  list

ctrlpts
1-dimensional array of control points (P).

This property sets and gets the control points in 1-D.

**Getter**  Gets unweighted control points. Use `weights` to get weights vector.

**Setter**  Sets unweighted control points.

**Type**  list

ctrlpts2d
2-dimensional array of control points.

The getter returns a tuple of 2D control points (weighted control points + weights if NURBS) in \([u/v]\) format. The rows of the returned tuple correspond to v-direction and the columns correspond to u-direction.

The following example can be used to traverse 2D control points:

```python
# Create a BSpline surface
surf_bs = BSpline.Surface()

# Do degree, control points and knot vector assignments here

# Each u includes a row of v values
for u in surf_bs.ctrlpts2d:
    # Each row contains the coordinates of the control points
    for v in u:
        print(str(v))  # will be something like (1.0, 2.0, 3.0)

# Create a NURBS surface
surf_nb = NURBS.Surface()

# Do degree, weighted control points and knot vector assignments here

# Each u includes a row of v values
for u in surf_nb.ctrlpts2d:
    # Each row contains the coordinates of the weighted control points
    for v in u:
        print(str(v))  # will be something like (0.5, 1.0, 1.5, 0.5)
```
When using NURBS.Surface class, the output of ctrlpts2d property could be confusing since, ctrlpts always returns the unweighted control points, i.e. ctrlpts property returns 3D control points all divided by the weights and you can use weights property to access the weights vector, but ctrlpts2d returns the weighted ones plus weights as the last element. This difference is intentionally added for compatibility and interoperability purposes.

To explain this situation in a simple way:

- If you need the weighted control points directly, use ctrlpts2d
- If you need the control points and the weights separately, use ctrlpts and weights

Note: Please note that the setter doesn’t check for inconsistencies and using the setter is not recommended. Instead of the setter property, please use set_ctrlpts() function.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the control points as a 2-dimensional array in [u][v] format

**Setter**  Sets the control points as a 2-dimensional array in [u][v] format

**Type**  list

**ctrlpts_size**
Total number of control points.

**Getter**  Gets the total number of control points

**Type**  int

**ctrlpts_size_u**
Number of control points for the u-direction.

Please refer to the wiki for details on using this class member.

**Getter**  Gets number of control points for the u-direction

**Setter**  Sets number of control points for the u-direction

**ctrlpts_size_v**
Number of control points for the v-direction.

Please refer to the wiki for details on using this class member.

**Getter**  Gets number of control points on the v-direction

**Setter**  Sets number of control points on the v-direction

**ctrlptsw**
1-dimensional array of weighted control points (Pw).

Weighted control points are in (x*w, y*w, z*w, w) format; where x,y,z are the coordinates and w is the weight.

This property sets and gets the control points in 1-D.

**Getter**  Gets weighted control points

**Setter**  Sets weighted control points

**data**
Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.
**degree**
Degree for u- and v-directions

- **Getter** Gets the degree
- **Setter** Sets the degree
- **Type** list

**degree_u**
Degree for the u-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets degree for the u-direction
- **Setter** Sets degree for the u-direction
- **Type** int

**degree_v**
Degree for the v-direction.

Please refer to the wiki for details on using this class member.

- **Getter** Gets degree for the v-direction
- **Setter** Sets degree for the v-direction
- **Type** int

**delta**
Evaluation delta for both u- and v-directions.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta and sample_size properties correspond to the same variable with different descriptions. Therefore, setting delta will also set sample_size.

The following figure illustrates the working principles of the delta property:

\[
[u_0, u_{\text{start}} + \delta, (u_{\text{start}} + \delta) + \delta, \ldots, u_{\text{end}}]
\]

Please refer to the wiki for details on using this class member.

- **Getter** Gets evaluation delta as a tuple of values corresponding to u- and v-directions
- **Setter** Sets evaluation delta for both u- and v-directions
- **Type** float

**delta_u**
Evaluation delta for the u-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_u and sample_size_u properties correspond to the same variable with different descriptions. Therefore, setting delta_u will also set sample_size_u.

Please refer to the wiki for details on using this class member.

- **Getter** Gets evaluation delta for the u-direction
Setter  Sets evaluation delta for the u-direction

Type  float

delta_v
Evaluation delta for the v-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_v and sample_size_v properties correspond to the same variable with different descriptions. Therefore, setting delta_v will also set sample_size_v.

Please refer to the wiki for details on using this class member.

Getter  Gets evaluation delta for the v-direction

Setter  Sets evaluation delta for the v-direction

Type  float

derivatives(u, v, order=0, **kwargs)
Evaluates n-th order surface derivatives at the given (u, v) parameter pair.

• SKL[0][0] will be the surface point itself
• SKL[0][1] will be the 1st derivative w.r.t. v
• SKL[2][1] will be the 2nd derivative w.r.t. u and 1st derivative w.r.t. v

Parameters
• u (float) – parameter on the u-direction
• v (float) – parameter on the v-direction
• order (integer) – derivative order

Returns  A list SKL, where SKL[k][l] is the derivative of the surface S(u,v) w.r.t. u k times and v l times

Return type  list

dimension
Spatial dimension.
Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

Getter  Gets the spatial dimension, e.g. 2D, 3D, etc.

Type  int
domain
Domain.
Domain is determined using the knot vector(s).

Getter  Gets the domain
evalpts
Evaluated points.

Please refer to the wiki for details on using this class member.
**Getter**  Gets the coordinates of the evaluated points

**Type**  list

**evaluate(**kwargs)**  
Evaluates the surface. 

The evaluated points are stored in `evalpts` property.

**Keyword arguments:**

- `start_u`: start parameter on the u-direction
- `stop_u`: stop parameter on the u-direction
- `start_v`: start parameter on the v-direction
- `stop_v`: stop parameter on the v-direction

The `start_u`, `start_v`, `stop_u`, and `stop_v` parameters allow evaluation of a surface segment in the range `[start_u, stop_u][start_v, stop_v]` i.e. the surface will also be evaluated at the `stop_u` and `stop_v` parameter values.

The following examples illustrate the usage of the keyword arguments.

```python
# Start evaluating in range u=[0, 0.7] and v=[0.1, 1]
surf.evaluate(stop_u=0.7, start_v=0.1)

# Start evaluating in range u=[0, 1] and v=[0.1, 0.3]
surf.evaluate(start_v=0.1, stop_v=0.3)

# Get the evaluated points
surface_points = surf.evalpts
```

**evaluate_list**(param_list)**  
Evaluates the surface for a given list of (u, v) parameters.

**Parameters**  param_list(list, tuple) – list of parameter pairs (u, v)

**Returns**  evaluated surface point at the input parameter pairs

**Return type**  tuple

**evaluate_single**(param)**  
Evaluates the surface at the input (u, v) parameter pair.

**Parameters**  param(list, tuple) – parameter pair (u, v)

**Returns**  evaluated surface point at the given parameter pair

**Return type**  list

**evaluator**  
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the current Evaluator instance

**Setter**  Sets the Evaluator instance

**Type**  evaluators.AbstractEvaluator
faces
Faces (triangles, quads, etc.) generated by the tessellation operation.
If the tessellation component is set to None, the result will be an empty list.

Getter Gets the faces

id
Object ID (as an integer).
Please refer to the wiki for details on using this class member.

Getter Gets the object ID
Setter Sets the object ID
Type int

insert_knot \((u=None, v=None, **kwargs)\)
Inserts knot(s) on the u- or v-directions

Keyword Arguments:
• num_u: Number of knot insertions on the u-direction. Default: 1
• num_v: Number of knot insertions on the v-direction. Default: 1

Parameters
• u (float) – knot to be inserted on the u-direction
• v (float) – knot to be inserted on the v-direction

knotvector
Knot vector for u- and v-directions

Getter Gets the knot vector
Setter Sets the knot vector
Type list

knotvector_u
Knot vector for the u-direction.
The knot vector will be normalized to \([0, 1]\) domain if the class is initialized with normalize_kv=True argument.
Please refer to the wiki for details on using this class member.

Getter Gets knot vector for the u-direction
Setter Sets knot vector for the u-direction
Type list

knotvector_v
Knot vector for the v-direction.
The knot vector will be normalized to \([0, 1]\) domain if the class is initialized with normalize_kv=True argument.
Please refer to the wiki for details on using this class member.

Getter Gets knot vector for the v-direction
Setter Sets knot vector for the v-direction
Type list

**load**(*file_name*)

Loads the surface from a pickled file.

Deprecated since version 5.2.4: Use exchange.import_json() instead.

**Parameters**

- **file_name** (str) – name of the file to be loaded

**name**

Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter**

Gets the object name

**Setter**

Sets the object name

**Type** str

**normal**(*parpos*, **kwargs*)

Evaluates the normal vector of the surface at the given parametric position(s).

The param argument can be

- a float value for evaluation at a single parametric position
- a list of float values for evaluation at the multiple parametric positions

The parametric positions should be a pair of (u,v) values. The return value will be in the order of the input parametric position list.

This method accepts the following keyword arguments:

- normalize: normalizes the output vector. Default value is *True*.

**Parameters**

- **parpos** (list or tuple) – parametric position(s) where the evaluation will be executed

**Returns**

an array containing “point” and “vector” pairs

**Return type**

tuple

**opt**

Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4] # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"] # creates "face_id" key and sets its value to a string
print(geom.opt) # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt # deletes the contents of the hash map
print(geom.opt) # will print: {}

geom.opt = ["body_id", 1] # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12] # changes the value of "body_id" to 12
print(geom.opt) # will print: {'body_id': 12}

geom.opt = ["body_id", None] # deletes "body_id"
print(geom.opt) # will print: {}
```
Please refer to the wiki for details on using this class member.

**Getter**  Gets the dict

**Setter**  Adds key and value pair to the dict

**Deleter**  Deletes the contents of the dict

```python
opt_get(value)
```
Safely query for the value from the opt property.

**Parameters**  
- **value**  (str) – a key in the opt property

**Returns**  
- the corresponding value, if the key exists. None, otherwise.

**order_u**
Order for the u-direction.
Defined as order = degree + 1

Please refer to the wiki for details on using this class member.

**Getter**  Gets order for the u-direction

**Setter**  Sets order for the u-direction

Type  int

**order_v**
Order for the v-direction.
Defined as order = degree + 1

Please refer to the wiki for details on using this class member.

**Getter**  Gets surface order for the v-direction

**Setter**  Sets surface order for the v-direction

Type  int

**pdimension**
Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the parametric dimension

Type  int

**range**
Domain range.

**Getter**  Gets the range

**rational**
Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

**Getter**  Returns True is the B-spline object is rational (NURBS)

Type  bool
**remove_knot** *(u=None, v=None, **kwargs)*

Inserts knot(s) on the u- or v-directions

**Keyword Arguments:**

- **num_u**: Number of knot removals on the u-direction. Default: 1
- **num_v**: Number of knot removals on the v-direction. Default: 1

**Parameters**

- **u (float)** – knot to be removed on the u-direction
- **v (float)** – knot to be removed on the v-direction

**render** *(**kwargs)*

Renders the surface using the visualization component.

The visualization component must be set using `vis` property before calling this method.

**Keyword Arguments:**

- **cpcolor**: sets the color of the control points grid
- **evalcolor**: sets the color of the surface
- **trimcolor**: sets the color of the trim curves
- **filename**: saves the plot with the input name
- **plot**: controls plot window visibility. Default: True
- **animate**: activates animation (if supported). Default: False
- **extras**: adds line plots to the figure. Default: None
- **colormap**: sets the colormap of the surface

The `plot` argument is useful when you would like to work on the command line without any window context. If `plot` flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

`extras` argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
[dict(  # line plot 1
    points=[[1, 2, 3], [4, 5, 6]],  # list of points
    name="My line Plot 1",  # name displayed on the legend
    color="red",  # color of the line plot
    size=6.5  # size of the line plot
),
 dict(  # line plot 2
    points=[[7, 8, 9], [10, 11, 12]],  # list of points
    name="My line Plot 2",  # name displayed on the legend
    color="navy",  # color of the line plot
    size=12.5  # size of the line plot
 )]
```

Please note that `colormap` argument can only work with visualization classes that support colormaps. As an example, please see `VisMPL.VisSurfTriangle()` class documentation. This method expects a single colormap input.
Returns the figure object

**reset**(**kwargs**)

Resets control points and/or evaluated points.

Keyword Arguments:

• `evalpts`: if True, then resets evaluated points
• `ctrlpts`: if True, then resets control points

**sample size**

Sample size for both u- and v-directions.

Sample size defines the number of surface points to generate. It also sets the `delta` property.

The following figure illustrates the working principles of sample size property:

![Sample Size Illustration](image)

Please refer to the [wiki](#) for details on using this class member.

**Getter** Gets sample size as a tuple of values corresponding to u- and v-directions

**Setter** Sets sample size for both u- and v-directions

**Type** int

**sample_size_u**

Sample size for the u-direction.

Sample size defines the number of surface points to generate. It also sets the `delta_u` property.

Please refer to the [wiki](#) for details on using this class member.

**Getter** Gets sample size for the u-direction

**Setter** Sets sample size for the u-direction

**Type** int

**sample_size_v**

Sample size for the v-direction.

Sample size defines the number of surface points to generate. It also sets the `delta_v` property.

Please refer to the [wiki](#) for details on using this class member.

**Getter** Gets sample size for the v-direction

**Setter** Sets sample size for the v-direction

**Type** int

**save**(file_name)

Saves the surface as a pickled file.

Deprecated since version 5.2.4: Use `exchange.export_json()` instead.

**Parameters**

`file_name` (str) – name of the file to be saved

**set_ctrlpts**(ctrlpts, *args, **kwargs)

Sets the control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input
will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing \((x, y, z)\) coordinates.

This method also generates 2D control points in \([u][v]\) format which can be accessed via \texttt{ctrlpts2d}.

\textbf{Note:} The \(v\) index varies first. That is, a row of \(v\) control points for the first \(u\) value is found first. Then, the row of \(v\) control points for the next \(u\) value.

\begin{itemize}
  \item \textbf{Parameters} \texttt{ctrlpts (list)} – input control points as a list of coordinates
  \item \textbf{tangent} (\texttt{parpos, **kwargs})
    Evaluates the tangent vectors of the surface at the given parametric position(s).
    The \texttt{param} argument can be
    \begin{itemize}
      \item a float value for evaluation at a single parametric position
      \item a list of float values for evaluation at the multiple parametric positions
    \end{itemize}
    The parametric positions should be a pair of \((u,v)\) values. The return value will be in the order of the input parametric position list.
    This method accepts the following keyword arguments:
    \begin{itemize}
      \item \texttt{normalize}: normalizes the output vector. Default value is \texttt{True}.
    \end{itemize}
    \item \textbf{Parameters} \texttt{parpos (list or tuple)} – parametric position(s) where the evaluation will be executed
    \item \textbf{Returns} an array containing “point” and “vector”\’s on u- and v-directions, respectively
    \item \textbf{Return type} tuple
  \end{itemize}

\textbf{tessellate} (**kwargs)
Tessellates the surface.
Keyword arguments are directly passed to the tessellation component.

\textbf{tessellator}
Tessellation component.
Please refer to the \texttt{wiki} for details on using this class member.

\textbf{Getter} Gets the tessellation component
\textbf{Setter} Sets the tessellation component

\textbf{transpose}()
Transposes the surface by swapping u and v parametric directions.

\textbf{trims}
Curves for trimming the surface.
Surface trims are 2-dimensional curves which are introduced on the parametric space of the surfaces. Trim curves can be a spline curve, an analytic curve or a 2-dimensional freeform shape. To visualize the trimmed surfaces, you need to use a tessellator that supports trimming. The following code snippet illustrates changing the default surface tessellator to the trimmed surface tessellator, \texttt{tessellate.TrimTessellate}.\n
120 Chapter 16. Core Modules
from geomdl import tessellate

# Assuming that "surf" variable stores the surface instance
surf.tessellator = tessellate.TrimTessellate()

In addition, using *trims* initialization argument of the visualization classes, trim curves can be visualized together with their underlying surfaces. Please refer to the visualization configuration class initialization arguments for more details.

Please refer to the wiki for details on using this class member.

**Getter** Gets the array of trim curves

**Setter** Sets the array of trim curves

**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter** Gets the geometry type

**Type** str

**vertices**
Vertices generated by the tessellation operation.

If the tessellation component is set to None, the result will be an empty list.

**Getter** Gets the vertices

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter** Gets the visualization component

**Setter** Sets the visualization component

**Type** vis.VisAbstract

**weights**
Weights vector.

**Getter** Gets the weights vector

**Setter** Sets the weights vector

**Type** list

**NURBS Volume**

New in version 5.0.

**class** geomdl.NURBS.Volume(**kwargs)

**Bases:** geomdl.BSpline.Volume

Data storage and evaluation class for NURBS (rational) volumes.

The rational shapes have some minor differences between the non-rational ones. This class is designed to operate with weighted control points (Pw) as described in *The NURBS Book* by Piegl and Tiller. Therefore, it provides a different set of properties (i.e. getters and setters):
• ctrlptsw: 1-dimensional array of weighted control points
• ctrlpts: 1-dimensional array of control points
• weights: 1-dimensional array of weights

This class provides the following properties:
• order_u
• order_v
• order_w
• degree_u
• degree_v
• degree_w
• knotvector_u
• knotvector_v
• knotvector_w
• ctrlptsw
• ctrlpts
• weights
• ctrlpts_size_u
• ctrlpts_size_v
• ctrlpts_size_w
• delta
• delta_u
• delta_v
• delta_w
• sample_size
• sample_size_u
• sample_size_v
• sample_size_w
• bbox
• name
• dimension
• vis
• evaluator
• rational

Keyword Arguments:
• precision: number of decimal places to round to. Default: 18
• normalize_kv: activates knot vector normalization. Default: True
- **find_span_func**: sets knot span search implementation.  
  Default: `helpers.find_span_linear()`

- **insert_knot_func**: sets knot insertion implementation.  
  Default: `operations.insert_knot()`

- **remove_knot_func**: sets knot removal implementation.  
  Default: `operations.remove_knot()`

Please refer to the `abstract.Volume()` documentation for more details.

### `add_trim(trim)`

Adds a trim to the volume.

* `trims` uses this method to add trims to the volume.

#### Parameters

- **trim**: abstract.Surface – trimming surface

### `bbox` (Bounding box)

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the bounding box
- **Type**: tuple

### `cpsize` (Number of control points in all parametric directions.)

- **Note**: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the number of control points
- **Setter**: Sets the number of control points
- **Type**: list

### `ctrlpts` (1-dimensional array of control points (P))

This property sets and gets the control points in 1-D.

- **Getter**: Gets unweighted control points. Use `weights` to get weights vector.
- **Setter**: Sets unweighted control points.
- **Type**: list

### `ctrlpts_size` (Total number of control points)

- **Getter**: Gets the total number of control points
- **Type**: int

### `ctrlpts_size_u` (Number of control points for the u-direction)

- Please refer to the wiki for details on using this class member.
- **Getter**: Gets number of control points for the u-direction
Setter  Sets number of control points for the u-direction

**ctrlpts_size_v**
Number of control points for the v-direction.

Please refer to the wiki for details on using this class member.

Getter  Gets number of control points for the v-direction

Setter  Sets number of control points for the v-direction

**ctrlpts_size_w**
Number of control points for the w-direction.

Please refer to the wiki for details on using this class member.

Getter  Gets number of control points for the w-direction

Setter  Sets number of control points for the w-direction

**ctrlptsw**
1-dimensional array of weighted control points (Pw).

Weighted control points are in \((x*w, y*w, z*w, w)\) format; where \(x,y,z\) are the coordinates and \(w\) is the weight.

This property sets and gets the control points in 1-D.

Getter  Gets weighted control points

Setter  Sets weighted control points

**data**
Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**degree**
Degree for u-, v- and w-directions

Getter  Gets the degree

Setter  Sets the degree

Type  list

**degree_u**
Degree for the u-direction.

Please refer to the wiki for details on using this class member.

Getter  Gets degree for the u-direction

Setter  Sets degree for the u-direction

Type  int

**degree_v**
Degree for the v-direction.

Please refer to the wiki for details on using this class member.

Getter  Gets degree for the v-direction

Setter  Sets degree for the v-direction

Type  int
degree_w
Degree for the w-direction.

Please refer to the wiki for details on using this class member.

**Getter** Gets degree for the w-direction

**Setter** Sets degree for the w-direction

**Type** int

delta
Evaluation delta for u-, v- and w-directions.

Evaluation delta corresponds to the *step size* while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta` and `sample_size` properties correspond to the same variable with different descriptions. Therefore, setting `delta` will also set `sample_size`.

The following figure illustrates the working principles of the delta property:

\[
[\begin{array}{cccc}
  u_0, & u_{\text{start}} + \delta, & (u_{\text{start}} + \delta) + \delta, & \ldots, & u_{\text{end}}
\end{array}]
\]

Please refer to the wiki for details on using this class member.

**Getter** Gets evaluation delta as a tuple of values corresponding to u-, v- and w-directions

**Setter** Sets evaluation delta for u-, v- and w-directions

**Type** float

delta_u
Evaluation delta for the u-direction.

Evaluation delta corresponds to the *step size* while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta_u` and `sample_size_u` properties correspond to the same variable with different descriptions. Therefore, setting `delta_u` will also set `sample_size_u`.

Please refer to the wiki for details on using this class member.

**Getter** Gets evaluation delta for the u-direction

**Setter** Sets evaluation delta for the u-direction

**Type** float

delta_v
Evaluation delta for the v-direction.

Evaluation delta corresponds to the *step size* while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta_v` and `sample_size_v` properties correspond to the same variable with different descriptions. Therefore, setting `delta_v` will also set `sample_size_v`.

Please refer to the wiki for details on using this class member.

**Getter** Gets evaluation delta for the v-direction

**Setter** Sets evaluation delta for the v-direction
Type float

delta_w
Evaluation delta for the w-direction.

Evaluation delta corresponds to the step size while `evaluate()` function iterates on the knot vector to
generate surface points. Decreasing step size results in generation of more surface points. Therefore;
smaller the delta value, smoother the surface.

Please note that `delta_w` and `sample_size_w` properties correspond to the same variable with differ-
etent descriptions. Therefore, setting `delta_w` will also set `sample_size_w`.

Please refer to the wiki for details on using this class member.

Getter Gets evaluation delta for the w-direction

Setter Sets evaluation delta for the w-direction

Type float
dimension
Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

Getter Gets the spatial dimension, e.g. 2D, 3D, etc.

Type int
domain
Domain.

Domain is determined using the knot vector(s).

Getter Gets the domain
evalpts
Evaluated points.

Please refer to the wiki for details on using this class member.

Getter Gets the coordinates of the evaluated points

Type list
evaluate(**kwars)
Evaluates the volume.

The evaluated points are stored in `evalpts` property.

Keyword arguments:

- `start_u`: start parameter on the u-direction
- `stop_u`: stop parameter on the u-direction
- `start_v`: start parameter on the v-direction
- `stop_v`: stop parameter on the v-direction
- `start_w`: start parameter on the w-direction
- `stop_w`: stop parameter on the w-direction

evaluate_list(param_list)
Evaluates the volume for a given list of (u, v, w) parameters.
Parameters `param_list` *(list, tuple)* – list of parameters in format *(u, v, w)*

Returns evaluated surface point at the input parameter pairs

Return type tuple

**evaluate_single** *(param)*
Evaluates the volume at the input *(u, v, w)* parameter.

Parameters `param` *(list, tuple)* – parameter *(u, v, w)*

Returns evaluated surface point at the given parameter pair

Return type list

**evaluator**
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

**Getter** Gets the current Evaluator instance

**Setter** Sets the Evaluator instance

**Type** `evaluators.AbstractEvaluator`

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter** Gets the object ID

**Setter** Sets the object ID

**Type** int

**insert_knot** *(u=None, v=None, w=None, **kwargs)*
Inserts knot(s) on the u-, v- and w-directions

**Keyword Arguments:**
- **num_u**: Number of knot insertions on the u-direction. Default: 1
- **num_v**: Number of knot insertions on the v-direction. Default: 1
- **num_w**: Number of knot insertions on the w-direction. Default: 1

Parameters
- **u** *(float)* – knot to be inserted on the u-direction
- **v** *(float)* – knot to be inserted on the v-direction
- **w** *(float)* – knot to be inserted on the w-direction

**knotvector**
Knot vector for u-, v- and w-directions

**Getter** Gets the knot vector

**Setter** Sets the knot vector

**Type** list
knotvector_u
Knot vector for the u-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

- **Getter** Gets knot vector for the u-direction
- **Setter** Sets knot vector for the u-direction
- **Type** list

knotvector_v
Knot vector for the v-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

- **Getter** Gets knot vector for the v-direction
- **Setter** Sets knot vector for the v-direction
- **Type** list

knotvector_w
Knot vector for the w-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

- **Getter** Gets knot vector for the w-direction
- **Setter** Sets knot vector for the w-direction
- **Type** list

load(file_name)
Loads the volume from a pickled file.

Deprecated since version 5.2.4: Use `exchange.import_json()` instead.

- **Parameters** file_name (str) – name of the file to be loaded

name
Object name (as a string)

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name
- **Type** str

opt
Dictionary for storing custom data in the current geometry object.

`opt` is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}

Please refer to the wiki for details on using this class member.

**Getter** Gets the dict

**Setter** Adds key and value pair to the dict

**Deleter** Deletes the contents of the dict

**opt_get**(value)
Safely query for the value from the opt property.

**Parameters**

value (str) – a key in the opt property

**Returns** the corresponding value, if the key exists. None, otherwise.

**order_u**
Order for the u-direction.

Defined as order = degree + 1

Please refer to the wiki for details on using this class member.

**Getter** Gets the surface order for u-direction

**Setter** Sets the surface order for u-direction

**Type** int

**order_v**
Order for the v-direction.

Defined as order = degree + 1

Please refer to the wiki for details on using this class member.

**Getter** Gets the surface order for v-direction

**Setter** Sets the surface order for v-direction

**Type** int

**order_w**
Order for the w-direction.

Defined as order = degree + 1

Please refer to the wiki for details on using this class member.

**Getter** Gets the surface order for v-direction
**Setter**  Sets the surface order for v-direction

**Type**  int

**pdimension**

Sets the parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the parametric dimension

**Type**  int

**range**

Sets the domain range.

**Getter**  Gets the range

**rational**

Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

**Getter**  Returns True is the B-spline object is rational (NURBS)

**Type**  bool

**remove_knot**(u=None, v=None, w=None, **kwargs)

Inserts knot(s) on the u-, v- and w-directions

**Keyword Arguments:**

- **num_u**:  Number of knot removals on the u-direction.  
  Default: 1

- **num_v**:  Number of knot removals on the v-direction.  
  Default: 1

- **num_w**:  Number of knot removals on the w-direction.  
  Default: 1

**Parameters**

- **u**(float) – knot to be removed on the u-direction

- **v**(float) – knot to be removed on the v-direction

- **w**(float) – knot to be removed on the w-direction

**render**(**kwargs)**

Renders the volume using the visualization component.

The visualization component must be set using `vis` property before calling this method.

**Keyword Arguments:**

- **cpcolor**: sets the color of the control points

- **evalcolor**: sets the color of the volume

- **filename**: saves the plot with the input name

- **plot**: controls plot window visibility.  
  Default: True

- **animate**: activates animation (if supported).  
  Default: False

- **grid_size**: grid size for voxelization.  
  Default: (8, 8, 8)
• use_cubes: use cube voxels instead of cuboid ones. Default: False
• num_procs: number of concurrent processes for voxelization. Default: 1

The plot argument is useful when you would like to work on the command line without any window context. If plot flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

extras argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```
[  
  dict(  # line plot 1
      points=[[1, 2, 3], [4, 5, 6]],  # list of points
      name="My line Plot 1",  # name displayed on the legend
      color="red",  # color of the line plot
      size=6.5  # size of the line plot
    ),
  dict(  # line plot 2
      points=[[7, 8, 9], [10, 11, 12]],  # list of points
      name="My line Plot 2",  # name displayed on the legend
      color="navy",  # color of the line plot
      size=12.5  # size of the line plot
    )
]
```

Returns the figure object

reset(**kwargs)
Resets control points and/or evaluated points.

Keyword Arguments:
• evalpts: if True, then resets the evaluated points
• ctrlpts if True, then resets the control points

sample_size
Sample size for both u- and v-directions.

Sample size defines the number of surface points to generate. It also sets the delta property.

The following figure illustrates the working principles of sample size property:

\[
\begin{align*}
\begin{bmatrix}
u_{\text{start}}, \ldots, u_{\text{end}}
\end{bmatrix} \\
n_{\text{sample}}
\end{align*}
\]

Please refer to the wiki for details on using this class member.

Getter Gets sample size as a tuple of values corresponding to u-, v- and w-directions
 Setter Sets sample size value for both u-, v- and w-directions
 Type int

sample_size_u
Sample size for the u-direction.

Sample size defines the number of evaluated points to generate. It also sets the delta_u property.

Please refer to the wiki for details on using this class member.
NURBS-Python Documentation, Release 5.2.9

**Getter**  Gets sample size for the u-direction

**Setter**  Sets sample size for the u-direction

**Type**  int

**sample_size_v**

Sample size for the v-direction.

Sample size defines the number of evaluated points to generate. It also sets the \( \text{delta}_v \) property.

Please refer to the wiki for details on using this class member.

**Getter**  Gets sample size for the v-direction

**Setter**  Sets sample size for the v-direction

**Type**  int

**sample_size_w**

Sample size for the w-direction.

Sample size defines the number of evaluated points to generate. It also sets the \( \text{delta}_w \) property.

Please refer to the wiki for details on using this class member.

**Getter**  Gets sample size for the w-direction

**Setter**  Sets sample size for the w-direction

**Type**  int

**save** (\textit{file_name})

Saves the volume as a pickled file.

Deprecated since version 5.2.4: Use \textit{exchange.export_json()} instead.

**Parameters**

\textit{file_name} \texttt{(str)} – name of the file to be saved

**set_ctrlpts** (\textit{ctrlpts}, *\textit{args}, **\textit{kwargs})

Sets the control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing \((x, y, z)\) coordinates.

**Parameters**

- \textit{ctrlpts} \texttt{(list)} – input control points as a list of coordinates
- \textit{args} \texttt{(tuple[int, int, int])} – number of control points corresponding to each parametric dimension

**trims**

Trimming surfaces.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the array of trim surfaces

**Setter**  Sets the array of trim surfaces

**type**

Geometry type

Please refer to the wiki for details on using this class member.
**Getter**  Gets the geometry type

**Type**  str

**vis**

Visualization component.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the visualization component

**Setter**  Sets the visualization component

**Type**  vis.VisAbstract

**weights**

Weights vector.

**Getter**  Gets the weights vector

**Setter**  Sets the weights vector

**Type**  list

### 16.1.3 Freeform Geometry

New in version 5.2.

The `freeform` module provides classes for representing freeform geometry objects.

The `Freeform` class provides a basis for storing freeform geometries. The points of the geometry can be set via the `evaluate()` method using a keyword argument.

#### Inheritance Diagram

```
geomdl.abstract.Geometry -> geomdl.freeform.Freeform
```

#### Class Reference

```python
class geomdl.freeform.Freeform(**kwargs)
```

**Bases:**  `geomdl.abstract.Geometry`

n-dimensional freeform geometry

**data**

Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**dimension**

Spatial dimension.

Please refer to the wiki for details on using this class member.
Getter  Gets the spatial dimension, e.g. 2D, 3D, etc.
Type    int

evalpts
Evaluated points.
Please refer to the wiki for details on using this class member.

Getter  Gets the coordinates of the evaluated points
Type    list

evaluate(**kwargs)
Sets points that form the geometry.

Keyword Arguments:
  • points: sets the points

id
Object ID (as an integer).
Please refer to the wiki for details on using this class member.

Getter  Gets the object ID
Setter  Sets the object ID
Type    int

name
Object name (as a string)
Please refer to the wiki for details on using this class member.

Getter  Gets the object name
Setter  Sets the object name
Type    str

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}
del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}
geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}
geom.opt = ['body_id', None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.
Getter  Gets the dict
Setter  Adds key and value pair to the dict
Deleter  Deletes the contents of the dict

opt_get (value)
Safely query for the value from the opt property.

Parameters value (str) – a key in the opt property

Returns  the corresponding value, if the key exists. None, otherwise.

type
Geometry type

Please refer to the wiki for details on using this class member.

Getter  Gets the geometry type
Type  str

16.1.4 Geometry Containers

The multi module provides specialized geometry containers. A container is a holder object that stores a collection of other objects, i.e. its elements. In NURBS-Python, containers can be generated as a result of

• A geometric operation, such as splitting
• File import, e.g. reading a file or a set of files containing multiple surfaces

The multi module contains the following classes:

• AbstractContainer abstract base class for containers
• CurveContainer for storing multiple curves
• SurfaceContainer for storing multiple surfaces
• VolumeContainer for storing multiple volumes

How to Use

These containers can be used for many purposes, such as visualization of a multi-component geometry or file export. For instance, the following figure shows a heart valve with 3 leaflets:
Each leaflet is a NURBS surface added to a `SurfaceContainer` and rendered via Matplotlib visualization module. It is possible to input a list of colors to the `render` method, otherwise it will automatically pick an arbitrary color.

**Inheritance Diagram**

```plaintext
geomdl.multi.CurveContainer
geomdl.multi.VolumeContainer
```

**Abstract Container**

```python
class geomdl.multi.AbstractContainer(*args, **kwargs)
Bases: geomdl.abstract.GeomdlBase
```
Abstract class for geometry containers.

This class implements Python Iterator Protocol and therefore any instance of this class can be directly used in a for loop.

This class provides the following properties:

- `type = container`
- `id`
- `name`
- `dimension`
- `opt`
- `pdimension`
- `evalpts`
- `bbox`
- `vis`
- `delta`
- `sample_size`

**add**(element)

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

Parameters **element** – geometry object

**append**(element)

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

Parameters **element** – geometry object

**bbox**

Bounding box.

Please refer to the wiki for details on using this class member.

Getter Gets the bounding box of all contained geometries

**data**

Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**delta**

Evaluation delta (for all parametric directions).

Evaluation delta corresponds to the step size. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta value, smoother the shape.

The following figure illustrates the working principles of the delta property:

\[
[u_{start}, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end}]
\]

Please refer to the wiki for details on using this class member.

Getter Gets the delta value
**Setter** Sets the delta value

**dimension**
Spatial dimension.

Please refer to the wiki for details on using this class member.

**Getter** Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

**evalpts**
Evaluated points.

Since there are multiple geometry objects contained in the multi objects, the evaluated points will be returned in the format of list of individual evaluated points which is also a list of Cartesian coordinates.

The following code example illustrates these details:

```python
multi_obj = multi.SurfaceContainer()  # it can also be multi.CurveContainer()
# Add geometries to multi_obj via multi_obj.add() method
# Then, the following loop will print all the evaluated points of the Multi object
for idx, mpt in enumerate(multi_obj.evalpts):
    print("Shape", idx+1, "contains", len(mpt), "points. These points are:")
    for pt in mpt:
        line = ", ".join([str(p) for p in pt])
        print(line)
```

Please refer to the wiki for details on using this class member.

**Getter** Gets the evaluated points of all contained geometries

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter** Gets the object ID

**Setter** Sets the object ID

**Type** int

**name**
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter** Gets the object name

**Setter** Sets the object name

**Type** str

**opt**
Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use **opt** property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
```

(continues on next page)
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ['body_id', None]  # deletes "body_id"
print(geom.opt)  # will print: {}

Please refer to the wiki for details on using this class member.

**Getter**  Gets the dict

**Setter**  Adds key and value pair to the dict

**Deleter**  Deletes the contents of the dict

```python
opt_get(value)
```
Safely query for the value from the `opt` property.

**Parameters**  `value` (`str`) – a key in the `opt` property

**Returns**  the corresponding value, if the key exists. `None`, otherwise.

**pdimension**
Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the parametric dimension

**Type**  `int`

```python
render(**kwargs)
```
Renders plots using the visualization component.

**Note:** This is an abstract method and it must be implemented in the subclass.

```python
reset()
```
Resets the cache.

**sample_size**
Sample size (for all parametric directions).

Sample size defines the number of points to evaluate. It also sets the `delta` property.

The following figure illustrates the working principles of sample size property:

\[
\begin{array}{c}
[\mathcal{u}_{\text{start}}, \ldots, \mathcal{u}_{\text{end}}] \\
\mathcal{n}_{\text{sample}}
\end{array}
\]

Please refer to the wiki for details on using this class member.

**Getter**  Gets sample size

**Setter**  Sets sample size
**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter** Gets the geometry type

**Type** str

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter** Gets the visualization component

**Setter** Sets the visualization component

---

**Curve Container**

```python
class geomdl.multi.CurveContainer(*args, **kwargs)
```

Bases: `geomdl.multi.AbstractContainer`

Container class for storing multiple curves.

This class implements Python Iterator Protocol and therefore any instance of this class can be directly used in a for loop.

This class provides the following properties:

- `type` = container
- `id`
- `name`
- `dimension`
- `opt`
- `pdimension`
- `evalpts`
- `bbox`
- `vis`
- `delta`
- `sample_size`

The following code example illustrates the usage of the Python properties:

```python
# Create a multi-curve container instance
mcrv = multi.CurveContainer()

# Add single or multi curves to the multi container using mcrv.add() command
# Addition operator, e.g. mcrv1 + mcrv2, also works

# Set the evaluation delta of the multi-curve
mcrv.delta = 0.05

# Get the evaluated points
curve_points = mcrv.evalpts
```
**add** *(element)*

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

**Parameters**

*element* – geometry object

**append** *(element)*

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

**Parameters**

*element* – geometry object

**bbox**

Bounding box.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the bounding box of all contained geometries

**data**

Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**delta**

Evaluation delta (for all parametric directions).

Evaluation delta corresponds to the *step size*. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta value, smoother the shape.

The following figure illustrates the working principles of the delta property:

\[ u_{start}, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end} \]

Please refer to the wiki for details on using this class member.

**Getter**

Gets the delta value

**Setter**

Sets the delta value

**dimension**

Spatial dimension.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type**

int

**evalpts**

Evaluated points.

Since there are multiple geometry objects contained in the multi objects, the evaluated points will be returned in the format of list of individual evaluated points which is also a list of Cartesian coordinates.

The following code example illustrates these details:

```python
multi_obj = multi.SurfaceContainer()  # it can also be multi.CurveContainer()
# Add geometries to multi_obj via multi_obj.add() method
# Then, the following loop will print all the evaluated points of the Multi
→object
for idx, mpt in enumerate(multi_obj.evalpts):
    print("Shape", idx+1, "contains", len(mpt), "points. These points are:")
```

(continues on next page)
for pt in mpt:
    line = ', '.join([str(p) for p in pt])
    print(line)

Please refer to the wiki for details on using this class member.

**Getter** Gets the evaluated points of all contained geometries

### id
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter** Gets the object ID

**Setter** Sets the object ID

**Type** int

### name
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter** Gets the object name

**Setter** Sets the object name

**Type** str

### opt
Dictionary for storing custom data in the current geometry object.

*opt* is a wrapper to a dict in *key => value* format, where *key* is string, *value* is any Python object. You can use *opt* property to store custom data inside the geometry object. For instance:

```python
geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ['body_id', None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.

**Getter** Gets the dict

**Setter** Adds key and value pair to the dict

**Deleter** Deletes the contents of the dict

**opt_get**(value)
Safely query for the value from the *opt* property.
Parameters

- **value** *(str)* – a key in the *opt* property

Returns

the corresponding value, if the key exists. *None*, otherwise.

**pdimension**

Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**

Gets the parametric dimension

**Type**

*int*

**render** (**kwargs**)

Renders the curves.

The visualization component must be set using *vis* property before calling this method.

Keyword Arguments:

- **cpcolor**
  sets the color of the control points grid
- **evalcolor**
  sets the color of the surface
- **filename**
  saves the plot with the input name
- **plot**
  controls plot window visibility. *Default*: *True*
- **animate**
  activates animation (if supported). *Default*: *False*
- **delta**
  if True, the evaluation delta of the container object will be used. *Default*: *True*
- **reset_names**
  resets the name of the curves inside the container. *Default*: *False*

The *cpcolor* and *evalcolor* arguments can be a string or a list of strings corresponding to the color values. Both arguments are processed separately, e.g. *cpcolor* can be a string whereas *evalcolor* can be a list or a tuple, or vice versa. A single string value sets the color to the same value. List input allows customization over the color values. If none provided, a random color will be selected.

The *plot* argument is useful when you would like to work on the command line without any window context. If *plot* flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

**reset** ()

Resets the cache.

**sample_size**

Sample size (for all parametric directions).

Sample size defines the number of points to evaluate. It also sets the *delta* property.

The following figure illustrates the working principles of sample size property:

\[
\left[u_{\text{start}}, \ldots, u_{\text{end}}\right]_{n_{\text{sample}}}
\]

Please refer to the wiki for details on using this class member.

**Getter**

Gets sample size

**Setter**

Sets sample size

**type**

Geometry type

Please refer to the wiki for details on using this class member.
**Getter**  Gets the geometry type

**Type**  str

**vis**

Visualization component.

Please refer to the [wiki](#) for details on using this class member.

**Getter**  Gets the visualization component

**Setter**  Sets the visualization component

---

**Surface Container**

```python
class geomdl.multi.SurfaceContainer(*args, **kwargs)
Bases: geomdl.multi.AbstractContainer
```

Container class for storing multiple surfaces.

This class implements Python Iterator Protocol and therefore any instance of this class can be directly used in a for loop.

This class provides the following properties:

- `type` = container
- `id`
- `name`
- `dimension`
- `opt`
- `pdimension`
- `evalpts`
- `bbox`
- `vis`
- `delta`
- `delta_u`
- `delta_v`
- `sample_size`
- `sample_size_u`
- `sample_size_v`
- `tessellator`
- `vertices`
- `faces`

The following code example illustrates the usage of these Python properties:
# Create a multi-surface container instance
msurf = multi.SurfaceContainer()

# Add single or multi surfaces to the multi container using msurf.add() command
# Addition operator, e.g. msurf1 + msurf2, also works

# Set the evaluation delta of the multi-surface
msurf.delta = 0.05

# Get the evaluated points
surface_points = msurf.evalpts

def add(element):
    """Add geometry objects to the container.
    """    The input can be a single geometry, a list of geometry objects or a geometry container object.

    Parameters:
    element -- geometry object

def append(element):
    """Add geometry objects to the container.
    """    The input can be a single geometry, a list of geometry objects or a geometry container object.

    Parameters:
    element -- geometry object

bbox
    Bounding box.

    Please refer to the wiki for details on using this class member.

    Getter
    Gets the bounding box of all contained geometries

data
    Returns a dict which contains the geometry data.

    Please refer to the wiki for details on using this class member.

def delta:
    """Evaluation delta (for all parametric directions).
    """
    Evaluation delta corresponds to the step size. Decreasing the step size results in evaluation of more points.
    Therefore; smaller the delta value, smoother the shape.

    The following figure illustrates the working principles of the delta property:

    \[ u_{start}, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end} \]

    Please refer to the wiki for details on using this class member.

    Getter
    Gets the delta value

    Setter
    Sets the delta value

def delta_u:
    """Evaluation delta for the u-direction.
    """
    Evaluation delta corresponds to the step size. Decreasing the step size results in evaluation of more points.
    Therefore; smaller the delta, smoother the shape.

    Please note that delta_u and sample_size_u properties correspond to the same variable with different descriptions. Therefore, setting delta_u will also set sample_size_u.

    Please refer to the wiki for details on using this class member.
**delta_v**

Evaluation delta for the v-direction.

Evaluation delta corresponds to the step size. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta, smoother the shape.

Please note that `delta_v` and `sample_size_v` properties correspond to the same variable with different descriptions. Therefore, setting `delta_v` will also set `sample_size_v`.

Please refer to the wiki for details on using this class member.

**dimension**

Spatial dimension.

Please refer to the wiki for details on using this class member.

**evalpts**

Evaluated points.

Since there are multiple geometry objects contained in the multi objects, the evaluated points will be returned in the format of list of individual evaluated points which is also a list of Cartesian coordinates.

The following code example illustrates these details:

```python
multi_obj = multi.SurfaceContainer()  # it can also be multi.CurveContainer()
# Add geometries to multi_obj via multi_obj.add() method
# Then, the following loop will print all the evaluated points of the Multi
for idx, mpt in enumerate(multi_obj.evalpts):
    print("Shape", idx+1, "contains", len(mpt), "points. These points are:")
    for pt in mpt:
        line = ", ".join([str(p) for p in pt])
        print(line)
```

Please refer to the wiki for details on using this class member.

**faces**

Faces (triangles, quads, etc.) generated by the tessellation operation.

If the tessellation component is set to None, the result will be an empty list.

**id**

Object ID (as an integer).

Please refer to the wiki for details on using this class member.
Getter  Gets the object ID

Setter  Sets the object ID

Type  int

name

Object name (as a string)

Please refer to the wiki for details on using this class member.

Getter  Gets the object name

Setter  Sets the object name

Type  str

opt

Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in \texttt{key => value} format, where \texttt{key} is string, \texttt{value} is any Python object. You can use \texttt{opt} property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.

Getter  Gets the dict

Setter  Adds key and value pair to the dict

Deleter  Deletes the contents of the dict

\texttt{opt_get(value)}

Safely query for the value from the \texttt{opt} property.

Parameters  \texttt{value (str)} – a key in the \texttt{opt} property

Returns  the corresponding value, if the key exists. None, otherwise.

pdimension

Parametric dimension.

Please refer to the wiki for details on using this class member.

Getter  Gets the parametric dimension

Type  int

render(**kwargs)

Renders the surfaces.
The visualization component must be set using \textit{vis} property before calling this method.

**Keyword Arguments:**

- \texttt{cpcolor}: sets the color of the control points grids
- \texttt{evalcolor}: sets the color of the surface
- \texttt{filename}: saves the plot with the input name
- \texttt{plot}: controls plot window visibility. \textit{Default: True}
- \texttt{animate}: activates animation (if supported). \textit{Default: False}
- \texttt{colormap}: sets the colormap of the surfaces
- \texttt{delta}: if True, the evaluation delta of the container object will be used. \textit{Default: True}
- \texttt{reset_names}: resets the name of the surfaces inside the container. \textit{Default: False}
- \texttt{num_procs}: number of concurrent processes for rendering the surfaces. \textit{Default: 1}

The \texttt{cpcolor} and \texttt{evalcolor} arguments can be a string or a list of strings corresponding to the color values. Both arguments are processed separately, e.g. \texttt{cpcolor} can be a string whereas \texttt{evalcolor} can be a list or a tuple, or vice versa. A single string value sets the color to the same value. List input allows customization over the color values. If none provided, a random color will be selected.

The \texttt{plot} argument is useful when you would like to work on the command line without any window context. If \texttt{plot} flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

Please note that \texttt{colormap} argument can only work with visualization classes that support colormaps. As an example, please see \texttt{VisMPL.VisSurfTriangle()} class documentation. This method expects multiple colormap inputs as a list or tuple, preferable the input list size is the same as the number of surfaces contained in the class. In the case of number of surfaces is bigger than number of input colormaps, this method will automatically assign a random color for the remaining surfaces.

\texttt{reset()} Reset the cache.

\texttt{sample_size} Sample size (for all parametric directions).

Sample size defines the number of points to evaluate. It also sets the \texttt{delta} property.

The following figure illustrates the working principles of sample size property:

\[
\begin{align*}
\left[ u_{\text{start}}, \ldots, u_{\text{end}} \right]
\end{align*}
\]

\[ n_{\text{sample}} \]

Please refer to the \texttt{wiki} for details on using this class member.

- \textbf{Getter} Gets sample size
- \textbf{Setter} Sets sample size

\texttt{sample_size_u} Sample size for the u-direction.

Sample size defines the number of points to evaluate. It also sets the \texttt{delta_u} property.

Please refer to the \texttt{wiki} for details on using this class member.

- \textbf{Getter} Gets sample size for the u-direction
**Setter** Sets sample size for the u-direction

**Type** int

**sample_size_v**
Sample size for the v-direction.

Sample size defines the number of points to evaluate. It also sets the `delta_v` property.

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the v-direction

**Setter** Sets sample size for the v-direction

**Type** int

**tessellate(**kwargs)**
Tessellates the surfaces inside the container.

Keyword arguments are directly passed to the tessellation component.

The following code snippet illustrates getting the vertices and faces of the surfaces inside the container:

```python
# Tessellate the surfaces inside the container
surf_container.tessellate()

# Vertices and faces are stored inside the tessellator component
ts = surf_container.tessellator

# Loop through all tessellator components
for t in ts:
    # Get the vertices
    vertices = t.tessellator.vertices
    # Get the faces (triangles, quads, etc.)
    faces = t.tessellator.faces
```

**Keyword Arguments:**
- `num_procs`: number of concurrent processes for tessellating the surfaces. **Default:** 1
- `delta`: if True, the evaluation delta of the container object will be used. **Default:** True
- `force`: flag to force tessellation. **Default:** False

**tessellator**
Tessellation component of the surfaces inside the container.

Please refer to `Tessellation` documentation for details.

```python
from geomdl import multi
from geomdl import tessellate

# Create the surface container
surf_container = multi.SurfaceContainer(surf_list)

# Set tessellator component
surf_container.tessellator = tessellate.TrimTessellate()
```

**Getter** gets the tessellation component

**Setter** sets the tessellation component
**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter** Gets the geometry type

**Type** str

**vertices**
Vertices generated by the tessellation operation.

If the tessellation component is set to None, the result will be an empty list.

**Getter** Gets the vertices

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter** Gets the visualization component

**Setter** Sets the visualization component

---

**Volume Container**

class `geomdl.multi.VolumeContainer(*args,**kwargs)`

**Bases:** `geomdl.multi.AbstractContainer`

Container class for storing multiple volumes.

This class implements Python Iterator Protocol and therefore any instance of this class can be directly used in a for loop.

This class provides the following properties:

- `type`
- `id`
- `name`
- `dimension`
- `opt`
- `pdimension`
- `evalpts`
- `bbox`
- `vis`
- `delta`
- `delta_u`
- `delta_v`
- `delta_w`
- `sample_size`
- `sample_size_u`
- `sample_size_v`
The following code example illustrates the usage of these Python properties:

```python
# Create a multi-volume container instance
mvol = multi.VolumeContainer()

# Add single or multi volumes to the multi container using mvol.add() command
# Addition operator, e.g. mvol1 + mvol2, also works

# Set the evaluation delta of the multi-volume
mvol.delta = 0.05

# Get the evaluated points
volume_points = mvol.evalpts
```

### add(element)

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

**Parameters**

- **element** -- geometry object

### append(element)

Adds geometry objects to the container.

The input can be a single geometry, a list of geometry objects or a geometry container object.

**Parameters**

- **element** -- geometry object

### bbox

Bounding box.

Please refer to the [wiki](#) for details on using this class member.

**Getter**

Gets the bounding box of all contained geometries

### data

Returns a dict which contains the geometry data.

Please refer to the [wiki](#) for details on using this class member.

### delta

Evaluation delta (for all parametric directions).

Evaluation delta corresponds to the *step size*. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta value, smoother the shape.

The following figure illustrates the working principles of the delta property:

\[
[u_{\text{start}}, u_{\text{start}} + \delta, (u_{\text{start}} + \delta) + \delta, \ldots, u_{\text{end}}]
\]

Please refer to the [wiki](#) for details on using this class member.

**Getter**

Gets the delta value

**Setter**

Sets the delta value

### delta_u

Evaluation delta for the u-direction.

Evaluation delta corresponds to the *step size*. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta, smoother the shape.
Please note that `delta_u` and `sample_size_u` properties correspond to the same variable with different descriptions. Therefore, setting `delta_u` will also set `sample_size_u`.

Please refer to the [wiki](#) for details on using this class member.

**Getter**  Gets the delta value for the u-direction

**Setter**  Sets the delta value for the u-direction

**Type**  float

### delta_v
Evaluation delta for the v-direction.

Evaluation delta corresponds to the *step size*. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta, smoother the shape.

Please note that `delta_v` and `sample_size_v` properties correspond to the same variable with different descriptions. Therefore, setting `delta_v` will also set `sample_size_v`.

Please refer to the [wiki](#) for details on using this class member.

**Getter**  Gets the delta value for the v-direction

**Setter**  Sets the delta value for the v-direction

**Type**  float

### delta_w
Evaluation delta for the w-direction.

Evaluation delta corresponds to the *step size*. Decreasing the step size results in evaluation of more points. Therefore; smaller the delta, smoother the shape.

Please note that `delta_w` and `sample_size_w` properties correspond to the same variable with different descriptions. Therefore, setting `delta_w` will also set `sample_size_w`.

Please refer to the [wiki](#) for details on using this class member.

**Getter**  Gets the delta value for the w-direction

**Setter**  Sets the delta value for the w-direction

**Type**  float

### dimension
Spatial dimension.

Please refer to the [wiki](#) for details on using this class member.

**Getter**  Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type**  int

### evalpts
Evaluated points.

Since there are multiple geometry objects contained in the multi objects, the evaluated points will be returned in the format of list of individual evaluated points which is also a list of Cartesian coordinates.

The following code example illustrates these details:

```python
multi_obj = multi.SurfaceContainer()  # it can also be multi.CurveContainer()
# Add geometries to multi_obj via multi_obj.add() method
# Then, the following loop will print all the evaluated points of the MultiObject
```
for idx, mpt in enumerate(multi_obj.evalpts):
    print("Shape", idx+1, "contains", len(mpt), "points. These points are:")
    for pt in mpt:
        line = ", ".join([str(p) for p in pt])
    print(line)

Please refer to the wiki for details on using this class member.

**Getter** Gets the evaluated points of all contained geometries

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter** Gets the object ID

**Setter** Sets the object ID

**Type** int

**name**
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter** Gets the object name

**Setter** Sets the object name

**Type** str

**opt**
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
gemopt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
gramopt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(gramopt)  # will print: ['face_id': 4, 'contents': 'data values']
del gemopt  # deletes the contents of the hash map
print(gemopt)  # will print: {}  
gemopt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
gramopt = ['body_id', 12]  # changes the value of "body_id" to 12
print(gramopt)  # will print: {'body_id': 12}
gemopt = ['body_id', None]  # deletes "body_id"
print(gramopt)  # will print: {}  
```

Please refer to the wiki for details on using this class member.

**Getter** Gets the dict

**Setter** Adds key and value pair to the dict

**Deleter** Deletes the contents of the dict
opt_get(value)

Safely query for the value from the opt property.

Parameters value (str) – a key in the opt property

Returns the corresponding value, if the key exists. None, otherwise.

pdimension

Parametric dimension.

Please refer to the wiki for details on using this class member.

Getter Gets the parametric dimension

Type int

render(**kwargs)

Renders the volumes.

The visualization component must be set using vis property before calling this method.

Keyword Arguments:

- cpcolor: sets the color of the control points plot
- evalcolor: sets the color of the volume
- filename: saves the plot with the input name
- plot: controls plot window visibility. Default: True
- animate: activates animation (if supported). Default: False
- delta: if True, the evaluation delta of the container object will be used. Default: True
- reset_names: resets the name of the volumes inside the container. Default: False
- grid_size: grid size for voxelization. Default: (16, 16, 16)
- num_procs: number of concurrent processes for voxelization. Default: 1

The cpcolor and evalcolor arguments can be a string or a list of strings corresponding to the color values. Both arguments are processed separately, e.g. cpcolor can be a string whereas evalcolor can be a list or a tuple, or vice versa. A single string value sets the color to the same value. List input allows customization over the color values. If none provided, a random color will be selected.

The plot argument is useful when you would like to work on the command line without any window context. If plot flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

reset()

Resets the cache.

sample_size

Sample size (for all parametric directions).

Sample size defines the number of points to evaluate. It also sets the delta property.

The following figure illustrates the working principles of sample size property:

\[
\left[ u_{\text{start}}, \ldots, u_{\text{end}} \right]_{n_{\text{sample}}}
\]

Please refer to the wiki for details on using this class member.

Getter Gets sample size
Setter  Sets sample size  

sample_size_u  
Sample size for the u-direction.  
Sample size defines the number of points to evaluate. It also sets the delta_u property.  
Please refer to the wiki for details on using this class member.  
  Getter  Gets sample size for the u-direction  
  Setter  Sets sample size for the u-direction  
  Type  int  

sample_size_v  
Sample size for the v-direction.  
Sample size defines the number of points to evaluate. It also sets the delta_v property.  
Please refer to the wiki for details on using this class member.  
  Getter  Gets sample size for the v-direction  
  Setter  Sets sample size for the v-direction  
  Type  int  

sample_size_w  
Sample size for the w-direction.  
Sample size defines the number of points to evaluate. It also sets the delta_w property.  
Please refer to the wiki for details on using this class member.  
  Getter  Gets sample size for the w-direction  
  Setter  Sets sample size for the w-direction  
  Type  int  

type  
Geometry type  
Please refer to the wiki for details on using this class member.  
  Getter  Gets the geometry type  
  Type  str  

vis  
Visualization component.  
Please refer to the wiki for details on using this class member.  
  Getter  Gets the visualization component  
  Setter  Sets the visualization component  
The following is the list of the features and geometric operations included in the library:

### 16.1.5 Geometric Operations

This module provides common geometric operations for curves and surfaces. It includes the following operations:

- Knot insertion, removal and refinement
- Curve and surface splitting / Bézier decomposition
• Tangent, normal and binormal evaluations
• Hodograph curve and surface computations
• Translation, rotation and scaling

Function Reference

geomdl.operations.insert_knot(obj, param, num, **kwargs)

Inserts knots n-times to a spline geometry.

The following code snippet illustrates the usage of this function:

```python
# Insert knot u=0.5 to a curve 2 times
operations.insert_knot(curve, [0.5], [2])

# Insert knot v=0.25 to a surface 1 time
operations.insert_knot(surface, [None, 0.25], [0, 1])

# Insert knots u=0.75, v=0.25 to a surface 2 and 1 times, respectively
operations.insert_knot(surface, [0.75, 0.25], [2, 1])

# Insert knot w=0.5 to a volume 1 time
operations.insert_knot(volume, [None, None, 0.5], [0, 0, 1])
```

Please note that input spline geometry object will always be updated if the knot insertion operation is successful.

Keyword Arguments:
• check_num: enables/disables operation validity checks. Default: True

Parameters
• obj (abstract.SplineGeometry) – spline geometry
• param (list, tuple) – knot(s) to be inserted in [u, v, w] format
• num (list, tuple) – number of knot insertions in [num_u, num_v, num_w] format

Returns updated spline geometry

geomdl.operations.remove_knot(obj, param, num, **kwargs)

Removes knots n-times from a spline geometry.

The following code snippet illustrates the usage of this function:

```python
# Remove knot u=0.5 from a curve 2 times
operations.remove_knot(curve, [0.5], [2])

# Remove knot v=0.25 from a surface 1 time
operations.remove_knot(surface, [None, 0.25], [0, 1])

# Remove knots u=0.75, v=0.25 from a surface 2 and 1 times, respectively
operations.remove_knot(surface, [0.75, 0.25], [2, 1])

# Remove knot w=0.5 from a volume 1 time
operations.remove_knot(volume, [None, None, 0.5], [0, 0, 1])
```

Please note that input spline geometry object will always be updated if the knot removal operation is successful.

Keyword Arguments:
• check_num: enables/disables operation validity checks. Default: True

Parameters

• `obj (abstract.SplineGeometry)` – spline geometry
• `param (list, tuple)` – knot(s) to be removed in [u, v, w] format
• `num (list, tuple)` – number of knot removals in [num_u, num_v, num_w] format

Returns updated spline geometry

geomdl.operations.refine_knotvector(obj, param, **kwargs)

Refines the knot vector(s) of a spline geometry.

The following code snippet illustrates the usage of this function:

```python
# Refines the knot vector of a curve
operations.refine_knotvector(curve, [1])

# Refines the knot vector on the v-direction of a surface
operations.refine_knotvector(surface, [0, 1])

# Refines the both knot vectors of a surface
operations.refine_knotvector(surface, [1, 1])

# Refines the knot vector on the w-direction of a volume
operations.refine_knotvector(volume, [0, 0, 1])
```

The values of `param` argument can be used to set the knot refinement density. If `density` is bigger than 1, then the algorithm finds the middle knots in each internal knot span to increase the number of knots to be refined.

**Example:** Let the degree is 2 and the knot vector to be refined is [0, 2, 4] with the superfluous knots from the start and end are removed. Knot vectors with the changing density (d) value will be:

• d = 1, knot vector [0, 1, 1, 2, 2, 3, 3, 4]
• d = 2, knot vector [0, 0.5, 0.5, 1, 1, 1.5, 1.5, 2, 2, 2.5, 2.5, 3, 3, 3.5, 3.5, 4]

The following code snippet illustrates the usage of knot refinement densities:

```python
# Refines the knot vector of a curve with density = 3
operations.refine_knotvector(curve, [3])

# Refines the knot vectors of a surface with density for u-dir = 2 and v-dir = 3
operations.refine_knotvector(surface, [2, 3])

# Refines only the knot vector on the v-direction of a surface with density = 1
operations.refine_knotvector(surface, [0, 1])

# Refines the knot vectors of a volume with density for u-dir = 1, v-dir = 3 and w-dir = 2
operations.refine_knotvector(volume, [1, 3, 2])
```

Please refer to `helpers.knot_refinement()` function for more usage options.

Keyword Arguments:

• check_num: enables/disables operation validity checks. Default: True
Parameters

- `obj (abstract.SplineGeometry)` – spline geometry
- `param (list, tuple)` – parametric dimensions to be refined in [u, v, w] format

**Returns** updated spline geometry

`geomdl.operations.add_dimension(obj, **kwargs)`
Elevates the spatial dimension of the spline geometry.

If you pass `inplace=True` keyword argument, the input will be updated. Otherwise, this function does not change the input but returns a new instance with the updated data.

Parameters `obj (abstract.SplineGeometry)` – spline geometry

**Returns** updated spline geometry

**Return type** `abstract.SplineGeometry`

`geomdl.operations.split_curve(obj, param, **kwargs)`
Splits the curve at the input parametric coordinate.

This method splits the curve into two pieces at the given parametric coordinate, generates two different curve objects and returns them. It does not modify the input curve.

**Keyword Arguments:**
- `find_span_func`: FindSpan implementation. **Default:** `helpers.find_span_linear()`
- `insert_knot_func`: knot insertion algorithm implementation. **Default:** `operations.insert_knot()`

Parameters

- `obj (abstract.Curve)` – Curve to be split
- `param (float)` – parameter

**Returns** a list of curve segments

**Return type** `list`

`geomdl.operations.decompose_curve(obj, **kwargs)`
Decomposes the curve into Bezier curve segments of the same degree.

This operation does not modify the input curve, instead it returns the split curve segments.

**Keyword Arguments:**
- `find_span_func`: FindSpan implementation. **Default:** `helpers.find_span_linear()`
- `insert_knot_func`: knot insertion algorithm implementation. **Default:** `operations.insert_knot()`

Parameters `obj (abstract.Curve)` – Curve to be decomposed

**Returns** a list of Bezier segments

**Return type** `list`

`geomdl.operations.derivative_curve(obj)`
Computes the hodograph (first derivative) curve of the input curve.
This function constructs the hodograph (first derivative) curve from the input curve by computing the degrees, knot vectors and the control points of the derivative curve.

**Parameters**
- `obj` ([abstract.Curve](#)) – input curve

**Returns**
- derivative curve

`geomdl.operations.length_curve(obj)`

Computes the approximate length of the parametric curve.

Uses the following equation to compute the approximate length:

\[
\sum_{i=0}^{n-1} \sqrt{P_{i+1}^2 - P_i^2}
\]

where \( n \) is number of evaluated curve points and \( P \) is the \( n \)-dimensional point.

**Parameters**
- `obj` ([abstract.Curve](#)) – input curve

**Returns**
- length

**Return type**
- float

`geomdl.operations.split_surface_u(obj, param, **kwargs)`

Splits the surface at the input parametric coordinate on the u-direction.

This method splits the surface into two pieces at the given parametric coordinate on the u-direction, generates two different surface objects and returns them. It does not modify the input surface.

**Keyword Arguments:**
- `find_span_func`: FindSpan implementation. **Default**: `helpers.find_span_linear()`
- `insert_knot_func`: knot insertion algorithm implementation. **Default**: `operations.insert_knot()`

**Parameters**
- `obj` ([abstract.Surface](#)) – surface
- `param` (float) – parameter for the u-direction

**Returns**
- a list of surface patches

**Return type**
- list

`geomdl.operations.split_surface_v(obj, param, **kwargs)`

Splits the surface at the input parametric coordinate on the v-direction.

This method splits the surface into two pieces at the given parametric coordinate on the v-direction, generates two different surface objects and returns them. It does not modify the input surface.

**Keyword Arguments:**
- `find_span_func`: FindSpan implementation. **Default**: `helpers.find_span_linear()`
- `insert_knot_func`: knot insertion algorithm implementation. **Default**: `operations.insert_knot()`

**Parameters**
- `obj` ([abstract.Surface](#)) – surface
- `param` (float) – parameter for the v-direction
Returns a list of surface patches

Return type list

geomdl.operations.decompose_surface(obj, **kwargs)

Decomposes the surface into Bezier surface patches of the same degree.

This operation does not modify the input surface, instead it returns the surface patches.

Keyword Arguments:

• find_span_func: FindSpan implementation. Default: helpers.find_span_linear()
• insert_knot_func: knot insertion algorithm implementation. Default: operations.insert_knot()

Parameters obj (abstract.Surface) – surface

Returns a list of Bezier patches

Return type list

geomdl.operations.derivative_surface(obj)

Computes the hodograph (first derivative) surface of the input surface.

This function constructs the hodograph (first derivative) surface from the input surface by computing the degrees, knot vectors and the control points of the derivative surface.

The return value of this function is a tuple containing the following derivative surfaces in the given order:

• U-derivative surface (derivative taken only on the u-direction)
• V-derivative surface (derivative taken only on the v-direction)
• UV-derivative surface (derivative taken on both the u- and the v-direction)

Parameters obj (abstract.Surface) – input surface

Returns derivative surfaces w.r.t. u, v and both u-v

Return type tuple

geomdl.operations.find_ctrlpts(obj, u=None, **kwargs)

Finds the control points involved in the evaluation of the curve/surface point defined by the input parameter(s).

Parameters

• obj (abstract.Curve or abstract.Surface) – curve or surface
• u (float) – parameter (for curve), parameter on the u-direction (for surface)
• v (float) – parameter on the v-direction (for surface only)

Returns control points; 1-dimensional array for curve, 2-dimensional array for surface

Return type list

geomdl.operations.tangent(obj, params, **kwargs)

Evaluates the tangent vector of the curves or surfaces at the input parameter values.

This function is designed to evaluate tangent vectors of the B-Spline and NURBS shapes at single or multiple parameter positions.

Parameters

• obj (abstract.Curve or abstract.Surface) – input shape
geomdl.operations.normal(obj, params, **kwargs)
Evaluates the normal vector of the curves or surfaces at the input parameter values.
This function is designed to evaluate normal vectors of the B-Spline and NURBS shapes at single or multiple parameter positions.

Parameters
- obj (abstract.Curve or abstract.Surface) – input geometry
- params (float, list or tuple) – parameters

Returns a list containing “point” and “vector” pairs
Return type tuple

geomdl.operations.binormal(obj, params, **kwargs)
Evaluates the binormal vector of the curves or surfaces at the input parameter values.
This function is designed to evaluate binormal vectors of the B-Spline and NURBS shapes at single or multiple parameter positions.

Parameters
- obj (abstract.Curve or abstract.Surface) – input shape
- params (float, list or tuple) – parameters

Returns a list containing “point” and “vector” pairs
Return type tuple

geomdl.operations.translate(obj, vec, **kwargs)
Translates curves, surfaces or volumes by the input vector.

Keyword Arguments:
- inplace: if False, operation applied to a copy of the object. Default: False

Parameters
- obj (abstract.SplineGeometry or multi.AbstractContainer) – input geometry
- vec (list, tuple) – translation vector

Returns translated geometry object

geomdl.operations.rotate(obj, angle, **kwargs)
Rotates curves, surfaces or volumes about the chosen axis.

Keyword Arguments:
- axis: rotation axis; x, y, z correspond to 0, 1, 2 respectively. Default: 2
- inplace: if False, operation applied to a copy of the object. Default: False

Parameters
- obj (abstract.SplineGeometry, multi.AbstractGeometry) – input geometry

16.1. User API
• **angle** (float) – angle of rotation (in degrees)

Returns rotated geometry object

gemdl.operations.scale(obj, multiplier, **kwargs)
Scales curves, surfaces or volumes by the input multiplier.

Keyword Arguments:
• inplace: if False, operation applied to a copy of the object. Default: False

Parameters
• obj (abstract.SplineGeometry, multi.AbstractGeometry) – input geometry
• multiplier (float) – scaling multiplier

Returns scaled geometry object

gemdl.operations.transpose(surf, **kwargs)
Transposes the input surface(s) by swapping u and v parametric directions.

Keyword Arguments:
• inplace: if False, operation applied to a copy of the object. Default: False

Parameters surf (abstract.Surface, multi.SurfaceContainer) – input surface(s)

Returns transposed surface(s)

gemdl.operations.flip(surf, **kwargs)
Flips the control points grid of the input surface(s).

Keyword Arguments:
• inplace: if False, operation applied to a copy of the object. Default: False

Parameters surf (abstract.Surface, multi.SurfaceContainer) – input surface(s)

Returns flipped surface(s)

16.1.6 Compatibility and Conversion

This module contains conversion operations related to control points, such as flipping arrays and adding weights.

Function Reference

gemdl.compatibility.combine_ctrlpts_weights(ctrlpts, weights=None)
Multipplies control points by the weights to generate weighted control points.

This function is dimension agnostic, i.e. control points can be in any dimension but weights should be 1D.

The weights function parameter can be set to None to let the function generate a weights vector composed of 1.0 values. This feature can be used to convert B-Spline basis to NURBS basis.

Parameters
• ctrlpts (list, tuple) – unweighted control points
• `weights(list, tuple or None)` — weights vector; if set to None, a weights vector of 1.0s will be automatically generated

Returns weighted control points

Return type list

gemdl.compatibility.flip_ctrlpts(ctrlpts, size_u, size_v)
Flips a list of 1-dimensional control points from v-row order to u-row order.

u-row order: each row corresponds to a list of u values

v-row order: each row corresponds to a list of v values

Parameters

• `ctrlpts(list, tuple)` — control points in v-row order
• `size_u(int)` — size in u-direction
• `size_v(int)` — size in v-direction

Returns control points in u-row order

Return type list

gemdl.compatibility.flip_ctrlpts2d(ctrlpts2d, size_u=0, size_v=0)
Flips a list of surface 2-D control points from [u][v] to [v][u] order.

Parameters

• `ctrlpts2d(list, tuple)` — 2-D control points
• `size_u(int)` — size in U-direction (row length)
• `size_v(int)` — size in V-direction (column length)

Returns flipped 2-D control points

Return type list

gemdl.compatibility.flip_ctrlpts2d_file(file_in=", file_out='ctrlpts_flip.txt'")
Flips u and v directions of a 2D control points file and saves flipped coordinates to a file.

Parameters

• `file_in(str)` — name of the input file (to be read)
• `file_out(str)` — name of the output file (to be saved)

 Raises IOError — an error occurred reading or writing the file

gemdl.compatibility.flip_ctrlpts_u(ctrlpts, size_u, size_v)
Flips a list of 1-dimensional control points from u-row order to v-row order.

u-row order: each row corresponds to a list of u values

v-row order: each row corresponds to a list of v values

Parameters

• `ctrlpts(list, tuple)` — control points in u-row order
• `size_u(int)` — size in u-direction
• `size_v(int)` — size in v-direction

Returns control points in v-row order

Return type list
geomdl.compatibility.generate_ctrlpts2d_weights(ctrlpts2d)
Generates unweighted control points from weighted ones in 2-D.

This function
1. Takes in 2-D control points list whose coordinates are organized like (x*w, y*w, z*w, w)
2. Converts the input control points list into (x, y, z, w) format
3. Returns the result

Parameters ctrlpts2d (list) – 2-D control points (P)
Returns 2-D weighted control points (Pw)
Return type list

geomdl.compatibility.generate_ctrlpts2d_weights_file(file_in=", file_out='ctrlpts_weights.txt'
Generates unweighted control points from weighted ones in 2-D.

1. Takes in 2-D control points list whose coordinates are organized like (x*w, y*w, z*w, w)
2. Converts the input control points list into (x, y, z, w) format
3. Saves the result to a file

Parameters
• file_in (str) – name of the input file (to be read)
• file_out (str) – name of the output file (to be saved)

Raises IOError – an error occurred reading or writing the file

geomdl.compatibility.generate_ctrlpts_weights(ctrlpts)
Generates unweighted control points from weighted ones in 1-D.

This function
1. Takes in 1-D control points list whose coordinates are organized in (x*w, y*w, z*w, w) format
2. Converts the input control points list into (x, y, z, w) format
3. Returns the result

Parameters ctrlpts (list) – 1-D control points (P)
Returns 1-D weighted control points (Pw)
Return type list

geomdl.compatibility.generate_ctrlptsw(ctrlpts)
Generates weighted control points from unweighted ones in 1-D.

This function
1. Takes in 1-D control points list whose coordinates are organized in (x, y, z, w) format
2. Converts into (x*w, y*w, z*w, w) format
3. Returns the result

Parameters ctrlpts (list) – 1-D control points (P)
Returns 1-D weighted control points (Pw)
Return type  list

geomdl.compatibility.generate_ctrlptsw2d(ctrlpts2d)
Generates weighted control points from unweighted ones in 2-D.

This function
1. Takes in a 2D control points list whose coordinates are organized in (x, y, z, w) format
2. converts into (x*w, y*w, z*w, w) format
3. Returns the result
Therefore, the returned list could be a direct input of the NURBS.Surface class.

Parameters  ctrlpts2d (list) – 2-D control points (P)
Returns  2-D weighted control points (Pw)
Return type  list

geomdl.compatibility.generate_ctrlptsw2d_file(file_in='ctrlpts.txt', file_out='ctrlptsw.txt')
Generates weighted control points from unweighted ones in 2-D.

This function
1. Takes in a 2-D control points file whose coordinates are organized in (x, y, z, w) format
2. Converts into (x*w, y*w, z*w, w) format
3. Saves the result to a file
Therefore, the resultant file could be a direct input of the NURBS.Surface class.

Parameters
file_in (str) – name of the input file (to be read)
file_out (str) – name of the output file (to be saved)
 Raises IOError – an error occurred reading or writing the file

geomdl.compatibility.separate_ctrlpts_weights(ctrlptsw)
Divides weighted control points by weights to generate unweighted control points and weights vector.

This function is dimension agnostic, i.e. control points can be in any dimension but the last element of the array should indicate the weight.

Parameters  ctrlptsw (list, tuple) – weighted control points
Returns  unweighted control points and weights vector
Return type  list

16.1.7 Geometry Converters

convert module provides functions for converting non-rational and rational geometries to each other.

Function Reference

geomdl.convert.bspline_to_nurbs(obj, **kwargs)
Converts non-rational splines to rational ones.
Parameters `obj` (BSpline.Curve, BSpline.Surface or BSpline.Volume) – non-rational spline geometry

Returns rational spline geometry

Return type `NURBS.Curve`, `NURBS.Surface` or `NURBS.Volume`

Raises TypeError

```python
geomdl.convert.nurbs_to_bspline(obj, **kwargs)
```

Converts rational splines to non-rational ones (if possible).

The possibility of converting a rational spline geometry to a non-rational one depends on the weights vector.

Parameters `obj` (NURBS.Curve, NURBS.Surface or NURBS.Volume) – rational spline geometry

Returns non-rational spline geometry

Return type `BSpline.Curve`, `BSpline.Surface` or `BSpline.Volume`

Raises TypeError

### 16.1.8 Geometry Constructors and Extractors

New in version 5.0.

`construct` module provides functions for constructing and extracting parametric shapes. A surface can be constructed from curves and a volume can be constructed from surfaces. Moreover, a surface can be extracted to curves and a volume can be extracted to surfaces in all parametric directions.

#### Function Reference

```python
geomdl.construct.construct_surface(direction, *args, **kwargs)
```

Generates surfaces from curves.

**Arguments:**
- `args`: a list of curve instances

**Keyword Arguments (optional):**
- `degree`: degree of the 2nd parametric direction
- `knotvector`: knot vector of the 2nd parametric direction
- `rational`: flag to generate rational surfaces

**Parameters** `direction` (str) – the direction that the input curves lies, i.e. u or v

**Returns** Surface constructed from the curves on the given parametric direction

```python
geomdl.construct.construct_volume(direction, *args, **kwargs)
```

Generates volumes from surfaces.

**Arguments:**
- `args`: a list of surface instances

**Keyword Arguments (optional):**
- `degree`: degree of the 3rd parametric direction
• knotvector: knot vector of the 3rd parametric direction
• rational: flag to generate rational volumes

Parameters direction (str) – the direction that the input surfaces lies, i.e. u, v, w

Returns Volume constructed from the surfaces on the given parametric direction

geomdl.construct.extract_curves (psurf, **kwargs)
Extracts curves from a surface.

The return value is a dict object containing the following keys:
• u: the curves which generate u-direction (or which lie on the v-direction)
• v: the curves which generate v-direction (or which lie on the u-direction)

As an example; if a curve lies on the u-direction, then its knotvector is equal to surface’s knotvector on the v-direction and vice versa.

The curve extraction process can be controlled via extract_u and extract_v boolean keyword arguments.

Parameters psurf (abstract.Surface) – input surface

Returns extracted curves

Return type dict

geomdl.construct.extract_isosurface (pvol)
Extracts the largest isosurface from a volume.

The following example illustrates one of the usage scenarios:

```python
from geomdl import construct, multi
from geomdl.visualization import VisMPL

# Assuming that "myvol" variable stores your spline volume information
isosrf = construct.extract_isosurface(myvol)

# Create a surface container to store extracted isosurface
msurf = multi.SurfaceContainer(isosrf)

# Set visualization components
msurf.vis = VisMPL.VisSurface(VisMPL.VisConfig(ctrlpts=False))

# Render isosurface
msurf.render()
```

Parameters pvol (abstract.Volume) – input volume

Returns isosurface (as a tuple of surfaces)

Return type tuple

geomdl.construct.extract_surfaces (pvol)
Extracts surfaces from a volume.

Parameters pvol (abstract.Volume) – input volume

Returns extracted surface

Return type dict
16.1.9 Curve and Surface Fitting

New in version 5.0.

The fitting module provides functions for interpolating and approximating B-spline curves and surfaces from data points. Approximation uses least squares algorithm.

Please see the following functions for details:

- `interpolate_curve()`
- `interpolate_surface()`
- `approximate_curve()`
- `approximate_surface()`

Surface fitting generates control points grid defined in \( u \) and \( v \) parametric dimensions. Therefore, the input requires number of data points to be fitted in both parametric dimensions. In other words, \( \text{size}_u \) and \( \text{size}_v \) arguments are used to fit curves of the surface on the corresponding parametric dimension.

Degree of the output spline geometry is important to determine the knot vector(s), compute the basis functions and build the coefficient matrix, \( A \). Most of the time, fitting to a quadratic (\( \text{degree} = 2 \)) or a cubic (\( \text{degree} = 3 \)) B-spline geometry should be good enough.

In the array structure, the data points on the \( v \)-direction come the first and \( u \)-direction points come. The index of the data points can be found using the following formula:

\[
\text{index} = v + (u \times \text{size}_v)
\]

Function Reference

`geomdl.fitting.interpolate_curve(points, degree, **kwargs)`

Curve interpolation through the data points.

Please refer to Algorithm A9.1 on The NURBS Book (2nd Edition), pp.369-370 for details.

**Keyword Arguments:**

- `centripetal`: activates centripetal parametrization method. *Default: False*

**Parameters**

- `points (list, tuple)` – data points
- `degree (int)` – degree of the output parametric curve

**Returns** interpolated B-Spline curve

**Return type** `BSpline.Curve`

`geomdl.fitting.interpolate_surface(points, size_u, size_v, degree_u, degree_v, **kwargs)`

Surface interpolation through the data points.

Please refer to the Algorithm A9.4 on The NURBS Book (2nd Edition), pp.380 for details.

**Keyword Arguments:**

- `centripetal`: activates centripetal parametrization method. *Default: False*

**Parameters**

- `points (list, tuple)` – data points
• `size_u (int)` – number of data points on the u-direction
• `size_v (int)` – number of data points on the v-direction
• `degree_u (int)` – degree of the output surface for the u-direction
• `degree_v (int)` – degree of the output surface for the v-direction

**Returns**  interpolated B-Spline surface

**Return type**  `BSpline.Surface`

`geomdl.fitting.approximate_curve(points, degree, **kwargs)`

Curve approximation using least squares method with fixed number of control points.

Please refer to The NURBS Book (2nd Edition), pp.410-413 for details.

**Keyword Arguments:**

• `centripetal`: activates centripetal parametrization method. *Default: False*
• `ctrlpts_size`: number of control points. *Default: len(points) - 1*

**Parameters**

• `points (list, tuple)` – data points
• `degree (int)` – degree of the output parametric curve

**Returns**  approximated B-Spline curve

**Return type**  `BSpline.Curve`

`geomdl.fitting.approximate_surface(points, size_u, size_v, degree_u, degree_v, **kwargs)`

Surface approximation using least squares method with fixed number of control points.

This algorithm interpolates the corner control points and approximates the remaining control points. Please refer to Algorithm A9.7 of The NURBS Book (2nd Edition), pp.422-423 for details.

**Keyword Arguments:**

• `centripetal`: activates centripetal parametrization method. *Default: False*
• `ctrlpts_size_u`: number of control points on the u-direction. *Default: size_u - 1*
• `ctrlpts_size_v`: number of control points on the v-direction. *Default: size_v - 1*

**Parameters**

• `points (list, tuple)` – data points
• `size_u (int)` – number of data points on the u-direction, r
• `size_v (int)` – number of data points on the v-direction, s
• `degree_u (int)` – degree of the output surface for the u-direction
• `degree_v (int)` – degree of the output surface for the v-direction

**Returns**  approximated B-Spline surface

**Return type**  `BSpline.Surface`
16.1.10 Tessellation

The `tessellate` module provides tessellation algorithms for surfaces. The following example illustrates the usage scenario of the tessellation algorithms with surfaces.

```python
from geomdl import NURBS
from geomdl import tessellate

# Create a surface instance
surf = NURBS.Surface()

# Set tessellation algorithm (you can use another algorithm)
surf.tessellator = tessellate.TriangularTessellate()

# Tessellate surface
surf.tessellate()
```

NURBS-Python uses `TriangularTessellate` class for surface tessellation by default.

**Note:** To get better results with the surface trimming, you need to use a relatively smaller evaluation delta or a bigger sample size value. Recommended evaluation delta is $d = 0.01$.

### Class Reference

**Abstract Tessellator**

```python
class geomdl.tessellate.AbstractTessellate(**kwargs):
    Bases: object

    Abstract base class for tessellation algorithms.

    arguments
    Arguments passed to the tessellation function.

    This property allows customization of the tessellation algorithm, and mainly designed to allow users to pass additional arguments to the tessellation function or change the behavior of the algorithm at runtime. This property can be thought as a way to input and store extra data for the tessellation functionality.

    Getter  Gets the tessellation arguments (as a dict)
    Setter  Sets the tessellation arguments (as a dict)

    faces
    Objects generated after tessellation.

    Getter  Gets the faces

    Type  elements.AbstractEntity

    is_tessellated()
    Checks if vertices and faces are generated.

    Returns  tessellation status

    Return type  bool

    reset()
    Clears stored vertices and faces.
```
**tessellate** *(points, **kwargs)*

Abstract method for the implementation of the tessellation algorithm. This algorithm should update *vertices* and *faces* properties.

**Note:** This is an abstract method and it must be implemented in the subclass.

**Parameters**

**points** – points to be tessellated

**vertices**

Vertex objects generated after tessellation.

**Getter**

Gets the vertices

**Type**

elements.AbstractEntity

**Triangular Tessellator**

**class**  geomdl.tessellate.TriangularTessellate(**kwargs)**

**Bases:** geomdl.tessellate.AbstractTessellate

Triangular tessellation algorithm for surfaces.

**arguments**

Arguments passed to the tessellation function.

This property allows customization of the tessellation algorithm, and mainly designed to allow users to pass additional arguments to the tessellation function or change the behavior of the algorithm at runtime. This property can be thought as a way to input and store extra data for the tessellation functionality.

**Getter**

Gets the tessellation arguments (as a dict)

**Setter**

Sets the tessellation arguments (as a dict)

**faces**

Objects generated after tessellation.

**Getter**

Gets the faces

**Type**

elements.AbstractEntity

**is_tessellated()**

Checks if vertices and faces are generated.

**Returns**

tessellation status

**Return type**

bool

**reset()**

Clears stored vertices and faces.

**tessellate** *(points, **kwargs)*

Applies triangular tessellation.

This function does not check if the points have already been tessellated.

**Keyword Arguments:**

- `size_u`: number of points on the u-direction
- `size_v`: number of points on the v-direction
Parameters `points (list, tuple)` – array of points

`vertices`
Vertex objects generated after tessellation.
- **Getter** Gets the vertices
- **Type** `elements.AbstractEntity`

**Trim Tessellator**

New in version 5.0.

```python
class geomdl.tessellate.TrimTessellate(**kwargs)
    Bases: geomdl.tessellate.AbstractTessellate
```

Triangular tessellation algorithm for trimmed surfaces.

- **arguments**
  Arguments passed to the tessellation function.
  - This property allows customization of the tessellation algorithm, and mainly designed to allow users to pass additional arguments to the tessellation function or change the behavior of the algorithm at runtime.
  - This property can be thought as a way to input and store extra data for the tessellation functionality.

  - **Getter** Gets the tessellation arguments (as a dict)
  - **Setter** Sets the tessellation arguments (as a dict)

- **faces**
  Objects generated after tessellation.
  - **Getter** Gets the faces
  - **Type** `elements.AbstractEntity`

- **is_tessellated()**
  Checks if vertices and faces are generated.
  - **Returns** tessellation status
  - **Return type** bool

- **reset()**
  Clears stored vertices and faces.

- **tessellate(points, **kwargs)**
  Applies triangular tessellation w/ trimming curves.

  - **Keyword Arguments:**
    - `size_u`: number of points on the u-direction
    - `size_v`: number of points on the v-direction

Parameters `points (list, tuple)` – array of points

`vertices`
Vertex objects generated after tessellation.
- **Getter** Gets the vertices
- **Type** `elements.AbstractEntity`
Quadrilateral Tessellator

New in version 5.2.

```python
class geomdl.tessellate.QuadTessellate(**kwargs):
    Bases: geomdl.tessellate.AbstractTessellate

    Quadrilateral tessellation algorithm for surfaces.

    arguments
    Arguments passed to the tessellation function.
    This property allows customization of the tessellation algorithm, and mainly designed to allow users to pass additional arguments to the tessellation function or change the behavior of the algorithm at runtime. This property can be thought as a way to input and store extra data for the tessellation functionality.
    Getter  Gets the tessellation arguments (as a dict)
    Setter  Sets the tessellation arguments (as a dict)

    faces
    Objects generated after tessellation.
    Getter  Gets the faces
    Type    elements.AbstractEntity

    is_tessellated()
    Checks if vertices and faces are generated.
    Returns  tessellation status
    Return type  bool

tessellate(points, **kwargs)
    Applies quadrilateral tessellation.
    This function does not check if the points have already been tessellated.
    Keyword Arguments:
    • size_u: number of points on the u-direction
    • size_v: number of points on the v-direction
    Parameters points (list, tuple) – array of points

    vertices
    Vertex objects generated after tessellation.
    Getter  Gets the vertices
    Type    elements.AbstractEntity
```

Function Reference

```python
geomdl.tessellate.make_triangle_mesh(points, size_u, size_v, **kwargs)
    Generates a triangular mesh from an array of points.
```
This function generates a triangular mesh for a NURBS or B-Spline surface on its parametric space. The input is the surface points and the number of points on the parametric dimensions u and v, indicated as row and column sizes in the function signature. This function should operate correctly if row and column sizes are input correctly, no matter what the points are v-ordered or u-ordered. Please see the documentation of `ctrlpts` and `ctrlpts2d` properties of the Surface class for more details on point ordering for the surfaces.

This function accepts the following keyword arguments:

- `vertex_spacing`: Defines the size of the triangles via setting the jump value between points
- `trims`: List of trim curves passed to the tessellation function
- `tessellate_func`: Function called for tessellation. Default: `tessellate.surface_tessellate()`
- `tessellate_args`: Arguments passed to the tessellation function (as a dict)

The tessellation function is designed to generate triangles from 4 vertices. It takes 4 `Vertex` objects, index values for setting the triangle and vertex IDs and additional parameters as its function arguments. It returns a tuple of `Vertex` and `Triangle` object lists generated from the input vertices. A default triangle generator is provided as a prototype for implementation in the source code.

The return value of this function is a tuple containing two lists. First one is the list of vertices and the second one is the list of triangles.

**Parameters**

- `points(list, tuple)` – input points
- `size_u(int)` – number of elements on the u-direction
- `size_v(int)` – number of elements on the v-direction

**Returns** a tuple containing lists of vertices and triangles

**Return type** tuple

`geomdl.tessellate.polygon_triangulate(tri_idx, *args)`

Triangulates a monotone polygon defined by a list of vertices.

The input vertices must form a convex polygon and must be arranged in counter-clockwise order.

**Parameters**

- `tri_idx(int)` – triangle numbering start value
- `args(Vertex)` – list of Vertex objects

**Returns** list of Triangle objects

**Return type** list

`geomdl.tessellate.make_quad_mesh(points, size_u, size_v)`

Generates a mesh of quadrilateral elements.

**Parameters**

- `points(list, tuple)` – list of points
- `size_u(int)` – number of points on the u-direction (column)
- `size_v(int)` – number of points on the v-direction (row)

**Returns** a tuple containing lists of vertices and quads

**Return type** tuple
Helper Functions

geomdl.tessellate.surface_tessellate(v1, v2, v3, v4, vidx, tidx, trim_curves, tessellate_args)
Triangular tessellation algorithm for surfaces with no trims.

This function can be directly used as an input to make_triangle_mesh() using tessellate_func keyword argument.

Parameters
- v1 (Vertex) – vertex 1
- v2 (Vertex) – vertex 2
- v3 (Vertex) – vertex 3
- v4 (Vertex) – vertex 4
- vidx (int) – vertex numbering start value
- tidx (int) – triangle numbering start value
- trim_curves – trim curves
- tessellate_args (dict) – tessellation arguments

Type list, tuple

Returns lists of vertex and triangle objects in (vertex_list, triangle_list) format

Type tuple

geomdl.tessellate.surface_trim_tessellate(v1, v2, v3, v4, vidx, tidx, trims, tessellate_args)
Triangular tessellation algorithm for trimmed surfaces.

This function can be directly used as an input to make_triangle_mesh() using tessellate_func keyword argument.

Parameters
- v1 (Vertex) – vertex 1
- v2 (Vertex) – vertex 2
- v3 (Vertex) – vertex 3
- v4 (Vertex) – vertex 4
- vidx (int) – vertex numbering start value
- tidx (int) – triangle numbering start value
- trims (list, tuple) – trim curves
- tessellate_args (dict) – tessellation arguments

Returns lists of vertex and triangle objects in (vertex_list, triangle_list) format

Type tuple

16.1.11 Trimming

Tessellation

Please refer to tessellate.TrimTessellate for tessellating the surfaces with trims.
Function Reference

**Warning:** The functions included in the `trimming` module are still work-in-progress and their functionality can change or they can be removed from the library in the next releases.
Please contact the author if you encounter any problems.

`geomdl.trimming.map_trim_to_geometry` *(obj, trim_idx=-1, **kwargs)*
Generates 3-dimensional mapping of 2-dimensional trimming curves.

**Description:**
Trimming curves are defined on the parametric space of the surfaces. Therefore, all trimming curves are 2-dimensional. The coordinates of the trimming curves correspond to (u, v) parameters of the underlying surface geometry. When these (u, v) values are evaluated with respect to the underlying surface geometry, a 3-dimensional representation of the trimming curves is generated.

The resultant 3-dimensional curve is described using `freeform.Freeform` class. Using the `fitting` module, it is possible to generate the B-spline form of the freeform curve.

**Remarks:**
If `trim_idx=-1`, the function maps all 2-dimensional trims to their 3-dimensional correspondants.

**Parameters**
- `obj` *(abstract.SplineGeometry)* – spline geometry
- `trim_idx` *(int)* – index of the trimming curve in the geometry object

**Returns** 3-dimensional mapping of trimming curve(s)

**Return type** `freeform.Freeform`

`geomdl.trimming.fix_multi_trim_curves` *(obj, **kwargs)*
Fixes direction, connectivity and similar issues of the trim curves.

This function works for surface trims in curve containers, i.e. trims consisting of multiple curves.

**Keyword Arguments:**
- `tol`: tolerance value for comparing floats. *Default: 10e-8*
- `delta`: evaluation delta of the trim curves. *Default: 0.05*

**Parameters**
- `obj` *(abstract_BSplineGeometry, multi.AbstractContainer)* – input surface

**Returns** updated surface

`geomdl.trimming.fix_trim_curves` *(obj)*
Fixes direction, connectivity and similar issues of the trim curves.

This function works for surface trim curves consisting of a single curve.

**Parameters**
- `obj` *(abstract.Surface)* – input surface
16.1.12 Sweeping

**Warning:** sweeping is a highly experimental module. Please use it with caution.

**Function Reference**

```python
geomdl.sweeping.sweep_vector(obj, vec, **kwargs)
```

Sweeps spline geometries along a vector.

This API call generates
- swept surfaces from curves
- swept volumes from surfaces

**Parameters**

- `obj` (abstract.SplineGeometry) – spline geometry
- `vec` (list, tuple) – vector to sweep along

**Returns** swept geometry

16.1.13 Import and Export Data

This module allows users to export/import NURBS shapes in common CAD exchange formats. The functions starting with `import_` are used for generating B-spline and NURBS objects from the input files. The functions starting with `export_` are used for saving B-spline and NURBS objects as files.

The following functions import/export control points or export evaluated points:

- `exchange.import_txt()`
- `exchange.export_txt()`
- `exchange.import_csv()`
- `exchange.export_csv()`

The following functions work with single or multiple surfaces:

- `exchange.import_obj()`
- `exchange.export_obj()`
- `exchange.export_stl()`
- `exchange.export_off()`
- `exchange.import_smesh()`
- `exchange.export_smesh()`

The following functions work with single or multiple volumes:

- `exchange.import_vmesh()`
- `exchange.export_vmesh()`

The following functions can be used to import/export rational or non-rational spline geometries:

- `exchange.import_yaml()`
• exchange.export_yaml()
• exchange.import_cfg()
• exchange.export_cfg()
• exchange.import_json()
• exchange.export_json()

The following functions work with single or multiple curves and surfaces:
• exchange.import_3dm()
• exchange.export_3dm()

Function Reference

```
geomdl.exchange.import_txt(file_name, two_dimensional=False, **kwargs)
```
Reads control points from a text file and generates a 1-dimensional list of control points.

The following code examples illustrate importing different types of text files for curves and surfaces:

1. # Import curve control points from a text file
2. curve_ctrlpts = exchange.import_txt(file_name="control_points.txt")
3. 
4. # Import surface control points from a text file (1-dimensional file)
5. surf_ctrlpts = exchange.import_txt(file_name="control_points.txt")
6. 
7. # Import surface control points from a text file (2-dimensional file)
8. surf_ctrlpts, size_u, size_v = exchange.import_txt(file_name="control_points.txt",
    two_dimensional=True)

If argument jinja2=True is set, then the input file is processed as a Jinja2 template. You can also use the following convenience template functions which correspond to the given mathematical equations:

• \(\sqrt{x}\)
• \(3\sqrt{x}\)
• \(x^y\)

You may set the file delimiters using the keyword arguments separator and col_separator, respectively. separator is the delimiter between the coordinates of the control points. It could be comma 1, 2, 3 or space 1 2 3 or something else. col_separator is the delimiter between the control points and is only valid when two-dimensional is True. Assuming that separator is set to space, then col_operator could be semi-colon 1 2 3; 4 5 6 or pipe 1 2 3| 4 5 6 or comma 1 2 3, 4 5 6 or something else.

The defaults for separator and col_separator are comma (,) and semi-colon (;), respectively.

The following code examples illustrate the usage of the keyword arguments discussed above.

1. # Import curve control points from a text file delimited with space
2. curve_ctrlpts = exchange.import_txt(file_name="control_points.txt", separator=" ")
3. 
4. # Import surface control points from a text file (2-dimensional file) w/ space and comma delimiters
5. surf_ctrlpts, size_u, size_v = exchange.import_txt(file_name="control_points.txt",
    two_dimensional=True,
    separator=" ", col_separator=",")
6. ```
Please note that this function does not check whether the user set delimiters to the same value or not.

Parameters

- `file_name (str)` – file name of the text file
- `two_dimensional (bool)` – type of the text file

Returns

- list of control points, if `two_dimensional`, then also returns size in u- and v-directions

Return type

- list

Raises

- `GeomdlException` – an error occurred reading the file

`geomdl.exchange.export_txt(obj, file_name, two_dimensional=False, **kwargs)`

Exports control points as a text file.

For curves the output is always a list of control points. For surfaces, it is possible to generate a 2-dimensional control point output file using `two_dimensional`.

Please see `exchange.import_txt()` for detailed description of the keyword arguments.

Parameters

- `obj (abstract.SplineGeometry)` – a spline geometry object
- `file_name (str)` – file name of the text file to be saved
- `two_dimensional (bool)` – type of the text file (only works for Surface objects)

 Raises

- `GeomdlException` – an error occurred writing the file

`geomdl.exchange.import_csv(file_name, **kwargs)`

Reads control points from a CSV file and generates a 1-dimensional list of control points.

It is possible to use a different value separator via `separator` keyword argument. The following code segment illustrates the usage of `separator` keyword argument.

```
1 # By default, import_csv uses 'comma' as the value separator
2 ctrlpts = exchange.import_csv("control_points.csv")
3
4 # Alternatively, it is possible to import a file containing tab-separated values
5 ctrlpts = exchange.import_csv("control_points.csv", separator="\t")
```

The only difference of this function from `exchange.import_txt()` is skipping the first line of the input file which generally contains the column headings.

Parameters

- `file_name (str)` – file name of the text file

Returns

- list of control points

Return type

- list

Raises

- `GeomdlException` – an error occurred reading the file

`geomdl.exchange.export_csv(obj, file_name, point_type='evalpts', **kwargs)`

Exports control points or evaluated points as a CSV file.

Parameters

- `obj (abstract.SplineGeometry)` – a spline geometry object
- `file_name (str)` – output file name
- `point_type (str)` – `ctrlpts` for control points or `evalpts` for evaluated points

 Raises

- `GeomdlException` – an error occurred writing the file
geomdl.exchange.import_cfg(file_name, **kwargs)
Imports curves and surfaces from files in libconfig format.

**Note:** Requires libconf package.

Use jinja2=True to activate Jinja2 template processing. Please refer to the documentation for details.

**Parameters**
- **file name** *(str)* – name of the input file

**Returns**
- a list of rational spline geometries

**Return type**
- list

**Raises**
- GeomdlException – an error occurred writing the file

geomdl.exchange.export_cfg(obj, file_name)
Exports curves and surfaces in libconfig format.

**Note:** Requires libconf package.

Libconfig format is also used by the geomdl command-line application as a way to input shape data from the command line.

**Parameters**
- **obj** *(abstract.SplineGeometry, multi.AbstractContainer)* – input geometry
- **file name** *(str)* – name of the output file

**Raises**
- GeomdlException – an error occurred writing the file

geomdl.exchange.import_yaml(file_name, **kwargs)
Imports curves and surfaces from files in YAML format.

**Note:** Requires ruamel.yaml package.

Use jinja2=True to activate Jinja2 template processing. Please refer to the documentation for details.

**Parameters**
- **file name** *(str)* – name of the input file

**Returns**
- a list of rational spline geometries

**Return type**
- list

**Raises**
- GeomdlException – an error occurred reading the file

geomdl.exchange.export_yaml(obj, file_name)
Exports curves and surfaces in YAML format.

**Note:** Requires ruamel.yaml package.

YAML format is also used by the geomdl command-line application as a way to input shape data from the command line.

**Parameters**
• `obj` *(abstract.SplineGeometry, multi.AbstractContainer)* – input geometry

• `file_name` *(str)* – name of the output file

Raises `GeomdlException` – an error occurred writing the file

```python
geomdl.exchange.import_json(file_name, **kwargs)
```

Imports curves and surfaces from files in JSON format.

Use `jinja2=True` to activate Jinja2 template processing. Please refer to the documentation for details.

Parameters `file_name` *(str)* – name of the input file

Returns a list of rational spline geometries

Return type list

Raises `GeomdlException` – an error occurred reading the file

```python
geomdl.exchange.export_json(obj, file_name)
```

Exports curves and surfaces in JSON format.

JSON format is also used by the `geomdl` command-line application as a way to input shape data from the command line.

Parameters

• `obj` *(abstract.SplineGeometry, multi.AbstractContainer)* – input geometry

• `file_name` *(str)* – name of the output file

Raises `GeomdlException` – an error occurred writing the file

```python
geomdl.exchange.import_obj(file_name, **kwargs)
```

Reads .obj files and generates faces.

Keyword Arguments:

• `callback`: reference to the function that processes the faces for customized output

The structure of the callback function is shown below:

```python
def my_callback_function(face_list):
    # "face_list" will be a list of elements.Face class instances
    # The function should return a list
    return list()
```

Parameters `file_name` *(str)* – file name

Returns output of the callback function (default is a list of faces)

Return type list

```python
geomdl.exchange.export_obj(surface, file_name, **kwargs)
```

Exports surface(s) as a .obj file.

Keyword Arguments:

• `vertex_spacing`: size of the triangle edge in terms of surface points sampled. Default: 2

• `vertex_normals`: if True, then computes vertex normals. Default: False

• `parametric_vertices`: if True, then adds parameter space vertices. Default: False

16.1. User API
• update_delta: use multi-surface evaluation delta for all surfaces. Default: True

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises GeomdlException – an error occurred writing the file

geomdl.exchange.export_stl(surface, file_name, **kwargs)
Exports surface(s) as a .stl file in plain text or binary format.

Keyword Arguments:

- binary: flag to generate a binary STL file. Default: True
- vertex_spacing: size of the triangle edge in terms of points sampled on the surface. Default: 1
- update_delta: use multi-surface evaluation delta for all surfaces. Default: True

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises GeomdlException – an error occurred writing the file

geomdl.exchange.export_off(surface, file_name, **kwargs)
Exports surface(s) as a .off file.

Keyword Arguments:

- vertex_spacing: size of the triangle edge in terms of points sampled on the surface. Default: 1
- update_delta: use multi-surface evaluation delta for all surfaces. Default: True

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises GeomdlException – an error occurred writing the file

geomdl.exchange.import_smesh(file)
Generates NURBS surface(s) from surface mesh (smesh) file(s).

smesh files are some text files which contain a set of NURBS surfaces. Each file in the set corresponds to one NURBS surface. Most of the time, you receive multiple smesh files corresponding to an complete object composed of several NURBS surfaces. The files have the extensions of .txt or .dat and they are named as

- smesh.X.Y.txt
- smesh.X.dat

where X and Y correspond to some integer value which defines the set the surface belongs to and part number of the surface inside the complete object.

Parameters file (str) – path to a directory containing mesh files or a single mesh file
Returns list of NURBS surfaces
Return type list
Raises GeomdlException – an error occurred reading the file

```python
goingdl.exchange.export_smesh(surface, file_name, **kwargs)
```
Exports surface(s) as surface mesh (smesh) files.

Please see `import_smesh()` for details on the file format.

Parameters
- `surface` (abstract.Surface or multi.SurfaceContainer) – surface(s) to be exported
- `file_name` (str) – name of the output file

Raises GeomdlException – an error occurred writing the file

```python
goingdl.exchange.import_vmesh(file)
```
Imports NURBS volume(s) from volume mesh (vmesh) file(s).

Parameters
- `file` (str) – path to a directory containing mesh files or a single mesh file

Returns list of NURBS volumes
Return type list
Raises GeomdlException – an error occurred reading the file

```python
goingdl.exchange.export_vmesh(volume, file_name, **kwargs)
```
Exports volume(s) as volume mesh (vmesh) files.

Parameters
- `volume` (abstract.Volume) – volume(s) to be exported
- `file_name` (str) – name of the output file

Raises GeomdlException – an error occurred writing the file

```python
goingdl.exchange.import_3dm(file_name, **kwargs)
```
Imports curves and surfaces from Rhinoceros/OpenNURBS .3dm files.

Deprecated since version 5.2.2: `rw3dm` Python module is replaced by `on2json`. It can be used to convert .3dm files to geomdl JSON format. Please refer to [https://github.com/orbingol/rw3dm](https://github.com/orbingol/rw3dm) for more details.

Parameters `file_name` (str) – input file name

```python
goingdl.exchange.export_3dm(obj, file_name, **kwargs)
```
Exports NURBS curves and surfaces to Rhinoceros/OpenNURBS .3dm files.

Deprecated since version 5.2.2: `rw3dm` Python module is replaced by `json2on`. It can be used to convert geomdl JSON format to .3dm files. Please refer to [https://github.com/orbingol/rw3dm](https://github.com/orbingol/rw3dm) for more details.

Parameters
- `file_name` (str) – file name
VTK Support

The following functions export control points and evaluated points as VTK files (in legacy format).

```python
geomdl.exchange_vtk.export_polydata(obj, file_name, **kwargs)
```

Exports control points or evaluated points in VTK Polydata format.

Please see the following document for details: http://www.vtk.org/VTK/img/file-formats.pdf

**Keyword Arguments:**

- `point_type`: `ctrlpts` for control points or `evalpts` for evaluated points
- `tessellate`: tessellates the points (works only for surfaces)

**Parameters**

- `file_name`: `str` – output file name

**Raises** `GeomdlException` – an error occurred writing the file

### 16.2 Geometry Generators

The following list contains the geometry generators/managers included in the library:

#### 16.2.1 Knot Vector Generator

The `knotvector` module provides utility functions related to knot vector generation and validation.

**Function Reference**

```python
geomdl.knotvector.generate(degree, num_ctrlpts, **kwargs)
```

Generates an equally spaced knot vector.

It uses the following equality to generate knot vector: \( m = n + p + 1 \)

where:

- \( p \), degree
- \( n + 1 \), number of control points
- \( m + 1 \), number of knots

**Keyword Arguments:**

- `clamped`: Flag to choose from clamped or unclamped knot vector options. **Default:** True

**Parameters**

- `degree`: `int` – degree
- `num_ctrlpts`: `int` – number of control points

**Returns** knot vector

**Return type** list
geomdl.knotvector.normalize(knot_vector, decimals=18)

Normalizes the input knot vector to [0, 1] domain.

**Parameters**
- `knot_vector (list, tuple)` – knot vector to be normalized
- `decimals (int)` – rounding number

**Returns**
normalized knot vector

**Return type**
list

geomdl.knotvector.check(degree, knot_vector, num_ctrlpts)

Checks the validity of the input knot vector.

Please refer to The NURBS Book (2nd Edition), p.50 for details.

**Parameters**
- `degree (int)` – degree of the curve or the surface
- `knot_vector (list, tuple)` – knot vector to be checked
- `num_ctrlpts (int)` – number of control points

**Returns**
True if the knot vector is valid, False otherwise

**Return type**
bool

### 16.2.2 Control Points Manager

The control_points module provides helper functions for managing control points. It is a better alternative to the compatibility module for managing control points. Please refer to the following class references for more details.

- `control_points.CurveManager`
- `control_points.SurfaceManager`
- `control_points.VolumeManager`

#### Class Reference

**class geomdl.control_points.AbstractManager(*args, **kwargs)**

Bases: object

Abstract base class for control points manager classes.

Control points manager class provides an easy way to set control points without knowing the internal data structure of the geometry classes. The manager class is initialized with the number of control points in all parametric dimensions.

All classes extending this class should implement the following methods:

- `find_index`

This class provides the following properties:

- `ctrlpts`

This class provides the following methods:

- `get_ctrlpt()`
- `set_ctrlpt()`
• `get_ptdata()`
• `set_ptdata()`

**ctrlpts**
Control points.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the control points

**Setter**  Sets the control points

**find_index** (*args*)
Finds the array index from the given parametric positions.

**Note:** This is an abstract method and it must be implemented in the subclass.

**get_ctrlpt** (*args*)
Gets the control point from the given location in the array.

**get_ptdata** (*dkey*, *args*)
Gets the data attached to the control point.

**Parameters**

- *dkey* – key of the attachment dictionary
- *dkey* – str

**reset**()
Resets/initializes the internal control points array.

**set_ctrlpt** (*pt*, *args*)
Puts the control point to the given location in the array.

**Parameters**

- *pt* (list, tuple) – control point

**set_ptdata** (*adct*, *args*)
Attaches the data to the control point.

**Parameters**

- *adct* – attachment dictionary
- *adct* – dict

**class** geomdl.control_points.CurveManager(*args, **kwargs*)
**Bases:** geomdl.control_points.AbstractManager

Curve control points manager.

Control points manager class provides an easy way to set control points without knowing the internal data structure of the geometry classes. The manager class is initialized with the number of control points in all parametric dimensions.

B-spline curves are defined in one parametric dimension. Therefore, this manager class should be initialized with a single integer value.

```python
# Assuming that the curve has 10 control points
manager = CurveManager(10)
```

Getting the control points:
# Number of control points in all parametric dimensions
size_u = spline.ctrlpts_size_u

# Generate control points manager
cpt_manager = control_points.SurfaceManager(size_u)
cpt_manager.ctrlpts = spline.ctrlpts

# Control points array to be used externally
control_points = []

# Get control points from the spline geometry
for u in range(size_u):
    pt = cpt_manager.get_ctrlpt(u)
    control_points.append(pt)

Setting the control points:

# Number of control points in all parametric dimensions
size_u = 5

# Create control points manager
points = control_points.SurfaceManager(size_u)

# Set control points
for u in range(size_u):
    # 'pt' is the control point, e.g. [10, 15, 12]
    points.set_ctrlpt(pt, u, v)

# Create spline geometry
curve = BSpline.Curve()

# Set control points
curve.ctrlpts = points.ctrlpts

ctrlpts
Control points.

Please refer to the wiki for details on using this class member.

**Getter** Gets the control points

**Setter** Sets the control points

**find_index** (*args*)
Finds the array index from the given parametric positions.

**Note:** This is an abstract method and it must be implemented in the subclass.

**get_ctrlpt** (*args*)
Gets the control point from the given location in the array.

**get_ptdata** (dkey, *args*)
Gets the data attached to the control point.

**Parameters**

- **dkey** – key of the attachment dictionary
- **dkey** – str
reset()

Resets/initializes the internal control points array.

set_ctrlpt(pt, *args)

Puts the control point to the given location in the array.

Parameters

- pt (list, tuple) – control point

set_ptdata(adct, *args)

Attaches the data to the control point.

Parameters

- adct – attachment dictionary
- adct – dict

class geomdl.control_points.SurfaceManager(*args, **kwargs)

Bases: geomdl.control_points.AbstractManager

Surface control points manager.

Control points manager class provides an easy way to set control points without knowing the internal data structure of the geometry classes. The manager class is initialized with the number of control points in all parametric dimensions.

B-spline surfaces are defined in one parametric dimension. Therefore, this manager class should be initialized with two integer values.

```python
# Assuming that the surface has size_u = 5 and size_v = 7 control points
manager = SurfaceManager(5, 7)
```

Getting the control points:

```python
# Number of control points in all parametric dimensions
size_u = spline.ctrlpts_size_u
size_v = spline.ctrlpts_size_v

# Generate control points manager
cpt_manager = control_points.SurfaceManager(size_u, size_v)
cpt_manager.ctrlpts = spline.ctrlpts

# Control points array to be used externally
control_points = []

# Get control points from the spline geometry
for u in range(size_u):
    for v in range(size_v):
        pt = cpt_manager.get_ctrlpt(u, v)
        control_points.append(pt)
```

Setting the control points:

```python
# Number of control points in all parametric dimensions
size_u = 5
size_v = 3

# Create control points manager
points = control_points.SurfaceManager(size_u, size_v)

# Set control points
```
for u in range(size_u):
    for v in range(size_v):
        # 'pt' is the control point, e.g. [10, 15, 12]
        points.set_ctrlpt(pt, u, v)

# Create spline geometry
surf = BSpline.Surface()

# Set control points
surf.ctrlpts = points.ctrlpts

ctrlpts
Control points.

Please refer to the wiki for details on using this class member.

  Getter  Gets the control points
  Setter  Sets the control points

find_index(*args)
Finds the array index from the given parametric positions.

  Note:  This is an abstract method and it must be implemented in the subclass.

get_ctrlpt(*args)
Gets the control point from the given location in the array.

get_ptdata(dkey, *args)
Gets the data attached to the control point.

  Parameters
    • dkey – key of the attachment dictionary
    • dkey – str

reset()
Resets/initializes the internal control points array.

set_ctrlpt(pt, *args)
Puts the control point to the given location in the array.

  Parameters
    pt (list, tuple) – control point

set_ptdata(adct, *args)
Attaches the data to the control point.

  Parameters
    • adct – attachment dictionary
    • adct – dict

class geomdl.control_points.VolumeManager(*args, **kwargs)
Bases: geomdl.control_points.AbstractManager

Volume control points manager.

16.2. Geometry Generators 189
Control points manager class provides an easy way to set control points without knowing the internal data structure of the geometry classes. The manager class is initialized with the number of control points in all parametric dimensions.

B-spline volumes are defined in one parametric dimension. Therefore, this manager class should be initialized with integer values.

```python
# Assuming that the volume has size_u = 5, size_v = 12 and size_w = 3 control points
manager = VolumeManager(5, 12, 3)
```

**Getting the control points:**

```python
# Number of control points in all parametric dimensions
size_u = spline.ctrlpts_size_u
size_v = spline.ctrlpts_size_v
size_w = spline.ctrlpts_size_w

# Generate control points manager
cpt_manager = control_points.SurfaceManager(size_u, size_v, size_w)
cpt_manager.ctrlpts = spline.ctrlpts

# Control points array to be used externally
control_points = []

# Get control points from the spline geometry
for u in range(size_u):
    for v in range(size_v):
        for w in range(size_w):
            pt = cpt_manager.get_ctrlpt(u, v, w)
            control_points.append(pt)
```

**Setting the control points:**

```python
# Number of control points in all parametric dimensions
size_u = 5
size_v = 3
size_w = 2

# Create control points manager
points = control_points.VolumeManager(size_u, size_v, size_w)

# Set control points
for u in range(size_u):
    for v in range(size_v):
        for w in range(size_w):
            # 'pt' is the control point, e.g. [10, 15, 12]
            points.set_ctrlpt(pt, u, v, w)

# Create spline geometry
volume = BSpline.Volume()

# Set control points
volume.ctrlpts = points.ctrlpts
```

`ctrlpts`  
Control points.

Please refer to the wiki for details on using this class member.
Getter  Gets the control points

Setter  Sets the control points

find_index (*args)
Finds the array index from the given parametric positions.

Note: This is an abstract method and it must be implemented in the subclass.

get_ctrlpt (*args)
Gets the control point from the given location in the array.

generate_data (dkey, *args)
Gets the data attached to the control point.

Parameters
• dkey – key of the attachment dictionary
• dkey – str

reset ()
Resets/initializes the internal control points array.

set_ctrlpt (pt, *args)
Puts the control point to the given location in the array.

Parameters
• pt (list, tuple) – control point

set_ptdata (adct, *args)
Attaches the data to the control point.

Parameters
• adct – attachment dictionary
• adct – dict

16.2.3 Surface Generator

CPGen module allows users to generate control points grids as an input to BSpline.Surface and NURBS.Surface classes. This module is designed to enable more testing cases in a very simple way and it doesn’t have the capabilities of a fully-featured grid generator, but it should be enough to be used side by side with BSpline and NURBS modules.

CPGen.Grid class provides an easy way to generate control point grids for use with BSpline.Surface class and CPGen.GridWeighted does the same for NURBS.Surface class.

Grid

class geomdl.CPGen.Grid(size_x, size_y, **kwargs)
Bases: object

Simple control points grid generator to use with non-rational surfaces.

This class stores grid points in [x, y, z] format and the grid (control) points can be retrieved from the grid attribute. The z-coordinate of the control points can be set via the keyword argument z_value while initializing the class.

Parameters
**size_x** (*float*) – width of the grid

**size_y** (*float*) – height of the grid

**bumps** (*num_bumps, **kwargs*)
Generates arbitrary bumps (i.e. hills) on the 2-dimensional grid.

This method generates hills on the grid defined by the **num_bumps** argument. It is possible to control the z-value using **bump_height** argument. **bump_height** can be a positive or negative numeric value or it can be a list of numeric values.

Please note that, not all grids can be modified to have **num_bumps** number of bumps. Therefore, this function uses a brute-force algorithm to determine whether the bumps can be generated or not. For instance:

```python
test_grid = Grid(5, 10)  # generates a 5x10 rectangle
test_grid.generate(4, 4)  # splits the rectangle into 2x2 pieces
test_grid.bumps(100)     # impossible, it will return an error message
test_grid.bumps(1)       # You will get a bump at the center of the generated grid
```

This method accepts the following keyword arguments:

- **bump_height**: z-value of the generated bumps on the grid. *Default: 5.0*
- **base_extent**: extension of the hill base from its center in terms of grid points. *Default: 2*
- **base_adjust**: padding between the bases of the hills. *Default: 0*

**Parameters**

**num_bumps** (*int*) – number of bumps (i.e. hills) to be generated on the 2D grid

**generate** (*num_u, num_v*)
Generates grid using the input division parameters.

**Parameters**

- **num_u** (*int*) – number of divisions in x-direction
- **num_v** (*int*) – number of divisions in y-direction

**grid**
Grid points.

Please refer to the wiki for details on using this class member.

**Getter** Gets the 2-dimensional list of points in [u][v] format

**reset()**
Resets the grid.

**Weighted Grid**

**class** geomdl.CPGen.GridWeighted(*size_x, size_y, **kwargs*)
**Bases:** geomdl.CPGen.Grid

Simple control points grid generator to use with rational surfaces.

This class stores grid points in [x*w, y*w, z*w, w] format and the grid (control) points can be retrieved from the **grid** attribute. The z-coordinate of the control points can be set via the keyword argument **z_value** while initializing the class.

**Parameters**

- **size_x** (*float*) – width of the grid
**size_y** (*float*) – height of the grid

**bumps** (*num_bumps, **kwargs*)
Generates arbitrary bumps (i.e. hills) on the 2-dimensional grid.

This method generates hills on the grid defined by the *num_bumps* argument. It is possible to control the z-value using *bump_height* argument. *bump_height* can be a positive or negative numeric value or it can be a list of numeric values.

Please note that, not all grids can be modified to have *num_bumps* number of bumps. Therefore, this function uses a brute-force algorithm to determine whether the bumps can be generated or not. For instance:

```python
test_grid = Grid(5, 10)  # generates a 5x10 rectangle
test_grid.generate(4, 4)  # splits the rectangle into 2x2 pieces
test_grid.bumps(100)    # impossible, it will return an error message
test_grid.bumps(1)      # You will get a bump at the center of the generated grid
```

This method accepts the following keyword arguments:

- **bump_height**: z-value of the generated bumps on the grid. *Default: 5.0*
- **base_extent**: extension of the hill base from its center in terms of grid points. *Default: 2*
- **base_adjust**: padding between the bases of the hills. *Default: 0*

**Parameters**

- **num_bumps** (*int*) – number of bumps (i.e. hills) to be generated on the 2D grid

**generate** (*num_u, num_v*)
Generates grid using the input division parameters.

**Parameters**

- **num_u** (*int*) – number of divisions in x-direction
- **num_v** (*int*) – number of divisions in y-direction

**grid**
Weighted grid points.

Please refer to the wiki for details on using this class member.

**Getter**
Gets the 2-dimensional list of weighted points in [u][v] format

**reset**()
Resets the grid.

**weight**
Weight (w) component of the grid points.

The input can be a single int or a float value, then all weights will be set to the same value.

Please refer to the wiki for details on using this class member.

**Getter**
Gets the weights vector

**Setter**
Sets the weights vector

## 16.3 Advanced API

The following list contains the modules for advanced use:
16.3.1 Geometry Base

abstract module provides base classes for parametric curves, surfaces and volumes contained in this library and therefore, it provides an easy way to extend the library in the most proper way.

Inheritance Diagram

![](inheritance.png)

Abstract Curve

class geomdl.abstract.Curve(**kwargs)
    Bases: geomdl.abstract.SplineGeometry

Abstract base class for defining spline curves.

Curve ABC is inherited from abc.ABCMeta class which is included in Python standard library by default. Due to differences between Python 2 and 3 on defining a metaclass, the compatibility module six is employed. Using six to set metaclass allows users to use the abstract classes in a correct way.

The abstract base classes in this module are implemented using a feature called Python Properties. This feature allows users to use some of the functions as if they are class fields. You can also consider properties as a pythonic way to set getters and setters. You will see “getter” and “setter” descriptions on the documentation of these properties.

The Curve ABC allows users to set the FindSpan function to be used in evaluations with find_span_func keyword as an input to the class constructor. NURBS-Python includes a binary and a linear search variation of the FindSpan function in the helpers module. You may also implement and use your own FindSpan function. Please see the helpers module for details.

Code segment below illustrates a possible implementation of Curve abstract base class:

```python
from geomdl import abstract

class MyCurveClass(abstract.Curve):
    def __init__(self, **kwargs):
        super(MyCurveClass, self).__init__(**kwargs)
        # Add your constructor code here

    def evaluate(self, **kwargs):
        # Implement this function
        pass

    def evaluate_single(self, uv):
        # Implement this function
        pass
```

(continues on next page)
The properties and functions defined in the abstract base class will be automatically available in the subclasses.

### Keyword Arguments:

- **id**: object ID (as integer)
- **precision**: number of decimal places to round to. Default: 18
- **normalize_kv**: if True, knot vector(s) will be normalized to [0,1] domain. Default: True
- **find_span_func**: default knot span finding algorithm. Default: `helpers.find_span_linear()`

#### bbox
Bounding box.
Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the bounding box
- **Type**: tuple

#### cpsize
Number of control points in all parametric directions.

**Note**: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the number of control points
- **Setter**: Sets the number of control points
- **Type**: list

#### ctrlpts
Control points.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the control points
- **Setter**: Sets the control points
- **Type**: list

#### ctrlpts.size
Total number of control points.

- **Getter**: Gets the total number of control points
- **Type**: int

```python
def evaluate_list(self, uv_list):
    # Implement this function
    pass

def derivatives(self, u, v, order, **kwargs):
    # Implement this function
    pass
```
data

Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

degree

Degree.

Please refer to the wiki for details on using this class member.

    Getter  Gets the degree
    Setter  Sets the degree
    Type    int

delta

Evaluation delta.

Evaluation delta corresponds to the step size while evaluate function iterates on the knot vector to generate curve points. Decreasing step size results in generation of more curve points. Therefore; smaller the delta value, smoother the curve.

The following figure illustrates the working principles of the delta property:

\[
[\begin{array}{c}
  u_{start} \\
  u_{start} + \delta \\
  (u_{start} + \delta) + \delta, \ldots, u_{end}
\end{array}]
\]

Please refer to the wiki for details on using this class member.

    Getter  Gets the delta value
    Setter  Sets the delta value
    Type    float

derivatives(u, order, **kwargs)

Evaluates the derivatives of the curve at parameter u.

Note: This is an abstract method and it must be implemented in the subclass.

Parameters

- **u (float)** – parameter (u)
- **order (int)** – derivative order
dimension

Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

    Getter  Gets the spatial dimension, e.g. 2D, 3D, etc.
    Type    int
domain

Domain.

Domain is determined using the knot vector(s).

    Getter  Gets the domain
evalpts
Evaluated points.

Please refer to the wiki for details on using this class member.

Getter  Gets the coordinates of the evaluated points

Type  list

evaluate(**kwargs)
Evaluates the curve.

Note:  This is an abstract method and it must be implemented in the subclass.

evaluate_list(param_list)
Evaluates the curve for an input range of parameters.

Note:  This is an abstract method and it must be implemented in the subclass.

Parameters  param_list  – array of parameters

evaluate_single(param)
Evaluates the curve at the given parameter.

Note:  This is an abstract method and it must be implemented in the subclass.

Parameters  param  – parameter (u)

evaluator
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

Getter  Gets the current Evaluator instance

Setter  Sets the Evaluator instance

Type  evaluators.AbstractEvaluator

id
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

Getter  Gets the object ID

Setter  Sets the object ID

Type  int

knotvector
Knot vector.

The knot vector will be normalized to [0, 1] domain if the class is initialized with normalize_kv= True argument.
Please refer to the wiki for details on using this class member.

**Getter**  Gets the knot vector  
**Setter**  Sets the knot vector  
**Type**  list

### name
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter**  Gets the object name  
**Setter**  Sets the object name  
**Type**  str

### opt
Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in **key => value** format, where **key** is a string, **value** is any Python object. You can use **opt** property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

delete geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.

**Getter**  Gets the dict  
**Setter**  Adds key and value pair to the dict  
**Deleter**  Deletes the contents of the dict

**opt_get**(value)
Safely query for the value from the **opt** property.

**Parameters**  

**value** (str) – a key in the **opt** property

**Returns**  

the corresponding value, if the key exists. **None**, otherwise.

### order
Order.

Defined as **order** = **degree** + 1

Please refer to the wiki for details on using this class member.

**Getter**  Gets the order  
**Setter**  Sets the order
Type  int

pdimension
Parametric dimension.

Please refer to the wiki for details on using this class member.

    Getter  Gets the parametric dimension
    Type    int

range
Domain range.

    Getter  Gets the range

rational
Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian co-
dinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational
B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

    Getter  Returns True is the B-spline object is rational (NURBS)
    Type    bool

render(**kwargs)
Renders the curve using the visualization component

The visualization component must be set using vis property before calling this method.

Keyword Arguments:

  • cpcolor: sets the color of the control points polygon
  • evalcolor: sets the color of the curve
  • bboxcolor: sets the color of the bounding box
  • filename: saves the plot with the input name
  • plot: controls plot window visibility. Default: True
  • animate: activates animation (if supported). Default: False
  • extras: adds line plots to the figure. Default: None

plot argument is useful when you would like to work on the command line without any window context. If plot flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

event argument can be used to add extra line plots to the figure. This argument expects a list of dicts in
the format described below:

```python
[  
    dict(  # line plot 1
        points=[[1, 2, 3], [4, 5, 6]],  # list of points
        name="My line Plot 1",  # name displayed on the legend
        color="red",  # color of the line plot
        size=6.5  # size of the line plot
    ),
]  
```

(continues on next page)
dict(  # line plot 2
    points=[[7, 8, 9], [10, 11, 12]],  # list of points
    name="My line Plot 2",  # name displayed on the legend
    color="navy",  # color of the line plot
    size=12.5  # size of the line plot
)

Returns the figure object

reset(**kwargs)
Resets control points and/or evaluated points.

Keyword Arguments:
- evalpts: if True, then resets evaluated points
- ctrlpts if True, then resets control points

reverse()
Reverses the curve

sample_size
Sample size.
Sample size defines the number of evaluated points to generate. It also sets the delta property.
The following figure illustrates the working principles of sample size property:

\[
\begin{bmatrix}
\mathbf{u}_{\text{start}}, \ldots, \mathbf{u}_{\text{end}}
\end{bmatrix}
\]

\[n_{\text{sample}}\]

Please refer to the wiki for details on using this class member.

Getter Gets sample size

Setter Sets sample size

Type int

set_ctrlpts(ctrlpts, *args, **kwargs)
Sets control points and checks if the data is consistent.
This method is designed to provide a consistent way to set control points whether they are weighted or not.
It directly sets the control points member of the class, and therefore it doesn’t return any values. The input
will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will
be an array of 3 elements representing \((x, y, z)\) coordinates.

Parameters ctrlpts(list) – input control points as a list of coordinates

type
Geometry type

Please refer to the wiki for details on using this class member.

Getter Gets the geometry type

Type str

vis
Visualization component.

Please refer to the wiki for details on using this class member.
Getter  Gets the visualization component
Setter  Sets the visualization component
Type    vis.VisAbstract

weights
Weights.

Note: Only available for rational spline geometries. Getter return None otherwise.

Please refer to the wiki for details on using this class member.

Getter  Gets the weights
Setter  Sets the weights

Abstract Surface

class geomdl.abstract.Surface(**kwargs)
Bases: geomdl.abstract.SplineGeometry

Abstract base class for defining spline surfaces.

Surface ABC is inherited from abc.ABCMeta class which is included in Python standard library by default. Due to differences between Python 2 and 3 on defining a metaclass, the compatibility module six is employed. Using six to set metaclass allows users to use the abstract classes in a correct way.

The abstract base classes in this module are implemented using a feature called Python Properties. This feature allows users to use some of the functions as if they are class fields. You can also consider properties as a pythonic way to set getters and setters. You will see “getter” and “setter” descriptions on the documentation of these properties.

The Surface ABC allows users to set the FindSpan function to be used in evaluations with find_span_func keyword as an input to the class constructor. NURBS-Python includes a binary and a linear search variation of the FindSpan function in the helpers module. You may also implement and use your own FindSpan function. Please see the helpers module for details.

Code segment below illustrates a possible implementation of Surface abstract base class:

```python
from geomdl import abstract

class MySurfaceClass(abstract.Surface):
    def __init__(self, **kwargs):
        super(MySurfaceClass, self).__init__(**kwargs)
        # Add your constructor code here

    def evaluate(self, **kwargs):
        # Implement this function
        pass

    def evaluate_single(self, uv):
        # Implement this function
        pass

    def evaluate_list(self, uv_list):
        # Implement this function
        pass
```

(continues on next page)
def derivatives(self, u, v, order, **kwargs):
    # Implement this function
    pass

The properties and functions defined in the abstract base class will be automatically available in the subclasses.

Keyword Arguments:

- **id**: object ID (as integer)
- **precision**: number of decimal places to round to. Default: 18
- **normalize_kv**: if True, knot vector(s) will be normalized to [0,1] domain. Default: True
- **find_span_func**: default knot span finding algorithm. Default: helpers.find_span_linear()

add_trim(trim)

Adds a trim to the surface.

A trim is a 2-dimensional curve defined on the parametric domain of the surface. Therefore, x-coordinate of the trimming curve corresponds to u parametric direction of the surface and y-coordinate of the trimming curve corresponds to v parametric direction of the surface.

trims uses this method to add trims to the surface.

Parameters trim (abstract.Geometry) – surface trimming curve

bbox

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

Getter Gets the bounding box

Type tuple

cpsize

Number of control points in all parametric directions.

Note: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

Getter Gets the number of control points

Setter Sets the number of control points

Type list

ctrlpts

1-dimensional array of control points.

Note: The v index varies first. That is, a row of v control points for the first u value is found first. Then, the row of v control points for the next u value.

Please refer to the wiki for details on using this class member.
Getter  Gets the control points
Setter  Sets the control points
Type  list

**ctrlpts_size**
Total number of control points.
Getter  Gets the total number of control points
Type  int

**ctrlpts_size_u**
Number of control points for the u-direction.
Please refer to the wiki for details on using this class member.
Getter  Gets number of control points for the u-direction
Setter  Sets number of control points for the u-direction

**ctrlpts_size_v**
Number of control points for the v-direction.
Please refer to the wiki for details on using this class member.
Getter  Gets number of control points on the v-direction
Setter  Sets number of control points on the v-direction

**data**
Returns a dict which contains the geometry data.
Please refer to the wiki for details on using this class member.

**degree**
Degree for u- and v-directions
Getter  Gets the degree
Setter  Sets the degree
Type  list

**degree_u**
Degree for the u-direction.
Please refer to the wiki for details on using this class member.
Getter  Gets degree for the u-direction
Setter  Sets degree for the u-direction
Type  int

**degree_v**
Degree for the v-direction.
Please refer to the wiki for details on using this class member.
Getter  Gets degree for the v-direction
Setter  Sets degree for the v-direction
Type  int
The following figure illustrates the working principles of the delta property:

\[
[u_0, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end}]
\]

Please refer to the wiki for details on using this class member.

**Getter**  Gets evaluation delta as a tuple of values corresponding to u- and v-directions

**Setter**  Sets evaluation delta for both u- and v-directions

**Type**  float

**derivatives**  \((u, v, order, **kwargs)\)

Evaluates the derivatives of the parametric surface at parameter \((u, v)\).

**Note:**  This is an abstract method and it must be implemented in the subclass.
Parameters

- **u** *(float)* – parameter on the u-direction
- **v** *(float)* – parameter on the v-direction
- **order** *(int)* – derivative order

**dimension**

Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

**Getter** Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

**domain**

Domain.

Domain is determined using the knot vector(s).

**Getter** Gets the domain

**evalpts**

Evaluated points.

Please refer to the wiki for details on using this class member.

**Getter** Gets the coordinates of the evaluated points

**Type** list

**evaluate** (**kwargs**)

Evaluates the parametric surface.

**Note:** This is an abstract method and it must be implemented in the subclass.

**evaluate_list** *(param_list)*

Evaluates the parametric surface for an input range of (u, v) parameters.

**Note:** This is an abstract method and it must be implemented in the subclass.

**Parameters** param_list – array of parameters (u, v)

**evaluate_single** *(param)*

Evaluates the parametric surface at the given (u, v) parameter.

**Note:** This is an abstract method and it must be implemented in the subclass.

**Parameters** param – parameter (u, v)

**evaluator**

Evaluator instance.
Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on `Evaluator` classes.

Please refer to the wiki for details on using this class member.

**Getter** Gets the current Evaluator instance

**Setter** Sets the Evaluator instance

**Type** `evaluators.AbstractEvaluator`

**faces**
Faces (triangles, quads, etc.) generated by the tessellation operation.

If the tessellation component is set to None, the result will be an empty list.

**Getter** Gets the faces

**id**
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter** Gets the object ID

**Setter** Sets the object ID

**Type** `int`

**knotvector**
Knot vector for u- and v-directions

**Getter** Gets the knot vector

**Setter** Sets the knot vector

**Type** `list`

**knotvector_u**
Knot vector for the u-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

**Getter** Gets knot vector for the u-direction

**Setter** Sets knot vector for the u-direction

**Type** `list`

**knotvector_v**
Knot vector for the v-direction.

The knot vector will be normalized to [0, 1] domain if the class is initialized with `initialize_kv=True` argument.

Please refer to the wiki for details on using this class member.

**Getter** Gets knot vector for the v-direction

**Setter** Sets knot vector for the v-direction

**Type** `list`
name
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter** Gets the object name

**Setter** Sets the object name

**Type** `str`

**opt**
Dictionary for storing custom data in the current geometry object.

`opt` is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}  
```

Please refer to the wiki for details on using this class member.

**Getter** Gets the dict

**Setter** Adds key and value pair to the dict

**Deleter** Deletes the contents of the dict

**opt_get(value)**
Safely query for the value from the `opt` property.

**Parameters**

- `value` (`str`) – a key in the `opt` property

**Returns**
the corresponding value, if the key exists. None, otherwise.

**order_u**
Order for the u-direction.

Defined as `order = degree + 1`

Please refer to the wiki for details on using this class member.

**Getter** Gets order for the u-direction

**Setter** Sets order for the u-direction

**Type** `int`

**order_v**
Order for the v-direction.

Defined as `order = degree + 1`
Please refer to the wiki for details on using this class member.

**Getter**  
Gets surface order for the v-direction

**Setter**  
Sets surface order for the v-direction

**Type**  
int

**pdimension**  
Parametric dimension.

Please refer to the wiki for details on using this class member.

**Getter**  
Gets the parametric dimension

**Type**  
int

**range**  
Domain range.

**Getter**  
Gets the range

**rational**  
Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

**Getter**  
Returns True is the B-spline object is rational (NURBS)

**Type**  
bool

**render(**kwargs**)**  
Renders the surface using the visualization component.

The visualization component must be set using `vis` property before calling this method.

**Keyword Arguments:**

- `cpcolor`: sets the color of the control points grid
- `evalcolor`: sets the color of the surface
- `trimcolor`: sets the color of the trim curves
- `filename`: saves the plot with the input name
- `plot`: controls plot window visibility. *Default: True*
- `animate`: activates animation (if supported). *Default: False*
- `extras`: adds line plots to the figure. *Default: None*
- `colormap`: sets the colormap of the surface

The `plot` argument is useful when you would like to work on the command line without any window context. If `plot` flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

`extras` argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:
```python
dict(  # line plot 1
    points=[[1, 2, 3], [4, 5, 6]],  # list of points
    name="My line Plot 1",  # name displayed on the legend
    color="red",  # color of the line plot
    size=6.5  # size of the line plot
),

dict(  # line plot 2
    points=[[7, 8, 9], [10, 11, 12]],  # list of points
    name="My line Plot 2",  # name displayed on the legend
    color="navy",  # color of the line plot
    size=12.5  # size of the line plot
)
```

Please note that colormap argument can only work with visualization classes that support colormaps. As an example, please see VisMPL.VisSurfTriangle() class documentation. This method expects a single colormap input.

**Returns** the figure object

**reset**(**kwargs**)

Resets control points and/or evaluated points.

**Keyword Arguments:**
- evalpts: if True, then resets evaluated points
- ctrlpts if True, then resets control points

**sample_size**

Sample size for both u- and v-directions.

Sample size defines the number of surface points to generate. It also sets the delta property.

The following figure illustrates the working principles of sample size property:

```
\[ u_{start}, \ldots, u_{end} \]
\[ n_{sample} \]
```

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size as a tuple of values corresponding to u- and v-directions

**Setter** Sets sample size for both u- and v-directions

**Type** int

**sample_size_u**

Sample size for the u-direction.

Sample size defines the number of surface points to generate. It also sets the delta_u property.

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the u-direction

**Setter** Sets sample size for the u-direction

**Type** int

**sample_size_v**

Sample size for the v-direction.
Sample size defines the number of surface points to generate. It also sets the `delta_v` property. Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the v-direction  
**Setter** Sets sample size for the v-direction  
**Type** `int`

`set_ctrlpts(ctrlpts, *args, **kwargs)`  
Sets the control points and checks if the data is consistent. This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing `(x, y, z)` coordinates.

**Note:** The v index varies first. That is, a row of v control points for the first u value is found first. Then, the row of v control points for the next u value.

**Parameters**

- `ctrlpts(list)` – input control points as a list of coordinates  
- `args(tuple[int, int])` – number of control points corresponding to each parametric dimension

`tessellate(**kwargs)`  
Tessellates the surface. Keyword arguments are directly passed to the tessellation component.

**tessellator**  
Tessellation component. Please refer to the wiki for details on using this class member.

**Getter** Gets the tessellation component  
**Setter** Sets the tessellation component

**trims**  
Curves for trimming the surface.

Surface trims are 2-dimensional curves which are introduced on the parametric space of the surfaces. Trim curves can be a spline curve, an analytic curve or a 2-dimensional freeform shape. To visualize the trimmed surfaces, you need to use a tessellator that supports trimming. The following code snippet illustrates changing the default surface tessellator to the trimmed surface tessellator, `tessellate.TrimTessellate`.

```python
from geomdl import tessellate

# Assuming that "surf" variable stores the surface instance
surf.tessellator = tessellate.TrimTessellate()
```

In addition, using `trims` initialization argument of the visualization classes, trim curves can be visualized together with their underlying surfaces. Please refer to the visualization configuration class initialization arguments for more details.

Please refer to the wiki for details on using this class member.
Getter  Gets the array of trim curves
Setter  Sets the array of trim curves

type
Geometry type
Please refer to the wiki for details on using this class member.
    Getter  Gets the geometry type
    Type    str

vertices
Vertices generated by the tessellation operation.
If the tessellation component is set to None, the result will be an empty list.
    Getter  Gets the vertices

vis
Visualization component.
Please refer to the wiki for details on using this class member.
    Getter  Gets the visualization component
    Setter  Sets the visualization component
    Type    vis.VisAbstract

weights
Weights.

Note: Only available for rational spline geometries. Getter return None otherwise.

Please refer to the wiki for details on using this class member.
    Getter  Gets the weights
    Setter  Sets the weights

Abstract Volume

class geomdl.abstract.Volume(**kwargs)
    Bases: geomdl.abstract.SplineGeometry

Abstract base class for defining spline volumes.

Volume ABC is inherited from abc.ABCMeta class which is included in Python standard library by default.
Due to differences between Python 2 and 3 on defining a metaclass, the compatibility module six is employed.
Using six to set metaclass allows users to use the abstract classes in a correct way.

The abstract base classes in this module are implemented using a feature called Python Properties. This feature
allows users to use some of the functions as if they are class fields. You can also consider properties as a
pythonic way to set getters and setters. You will see “getter” and “setter” descriptions on the documentation of
these properties.

The Volume ABC allows users to set the FindSpan function to be used in evaluations with find_span_func
keyword as an input to the class constructor. NURBS-Python includes a binary and a linear search variation of
the FindSpan function in the helpers module. You may also implement and use your own FindSpan function.
Please see the helpers module for details.
from geomdl import abstract

class MyVolumeClass(abstract.Volume):
    def __init__(self, **kwargs):
        super(MyVolumeClass, self).__init__(**kwargs)
        # Add your constructor code here

    def evaluate(self, **kwargs):
        # Implement this function
        pass

    def evaluate_single(self, uvw):
        # Implement this function
        pass

    def evaluate_list(self, uvw_list):
        # Implement this function
        pass

The properties and functions defined in the abstract base class will be automatically available in the subclasses.

**Keyword Arguments:**

- **id**: object ID (as integer)
- **precision**: number of decimal places to round to. *Default: 18*
- **normalize_kv**: if True, knot vector(s) will be normalized to [0,1] domain. *Default: True*
- **find_span_func**: default knot span finding algorithm. *Default: helpers.find_span_linear()*

**add_trim**(trim)

Add a trim to the volume.

*trims* uses this method to add trims to the volume.

**Parameters**

- **trim**: (abstract.Surface) – trimming surface

**bbox**

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the bounding box
  - **Type**: tuple

**cpsize**

Number of control points in all parametric directions.

**Note**: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

- **Getter**: Gets the number of control points
  - **Type**: tuple
- **Setter**: Sets the number of control points
Type list

**ctrlpts**
1-dimensional array of control points.

Please refer to the wiki for details on using this class member.

**Getter** Gets the control points

**Setter** Sets the control points

**Type** list

**ctrlpts_size**
Total number of control points.

**Getter** Gets the total number of control points

**Type** int

**ctrlpts_size_u**
Number of control points for the u-direction.

Please refer to the wiki for details on using this class member.

**Getter** Gets number of control points for the u-direction

**Setter** Sets number of control points for the u-direction

**ctrlpts_size_v**
Number of control points for the v-direction.

Please refer to the wiki for details on using this class member.

**Getter** Gets number of control points for the v-direction

**Setter** Sets number of control points for the v-direction

**ctrlpts_size_w**
Number of control points for the w-direction.

Please refer to the wiki for details on using this class member.

**Getter** Gets number of control points for the w-direction

**Setter** Sets number of control points for the w-direction

**data**
Returns a dict which contains the geometry data.

Please refer to the wiki for details on using this class member.

**degree**
Degree for u-, v- and w-directions

**Getter** Gets the degree

**Setter** Sets the degree

**Type** list

**degree_u**
Degree for the u-direction.

Please refer to the wiki for details on using this class member.

**Getter** Gets degree for the u-direction

**Setter** Sets degree for the u-direction
**NURBS-Python Documentation, Release 5.2.9**

**Type**  int

**degree_v**

Degree for the v-direction.

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets degree for the v-direction
- **Setter** Sets degree for the v-direction

**Type**  int

**degree_w**

Degree for the w-direction.

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets degree for the w-direction
- **Setter** Sets degree for the w-direction

**Type**  int

**delta**

Evaluation delta for u-, v- and w-directions.

Evaluation delta corresponds to the *step size* while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta` and `sample_size` properties correspond to the same variable with different descriptions. Therefore, setting `delta` will also set `sample_size`.

The following figure illustrates the working principles of the delta property:

\[ u_0, u_{start} + \delta, (u_{start} + \delta) + \delta, \ldots, u_{end} \]

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets evaluation delta as a tuple of values corresponding to u-, v- and w-directions
- **Setter** Sets evaluation delta for u-, v- and w-directions

**Type**  float

**delta_u**

Evaluation delta for the u-direction.

Evaluation delta corresponds to the *step size* while `evaluate()` function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that `delta_u` and `sample_size_u` properties correspond to the same variable with different descriptions. Therefore, setting `delta_u` will also set `sample_size_u`.

Please refer to the [wiki](#) for details on using this class member.

- **Getter** Gets evaluation delta for the u-direction
- **Setter** Sets evaluation delta for the u-direction

**Type**  float
delta_v
Evaluation delta for the v-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_v and sample_size_v properties correspond to the same variable with different descriptions. Therefore, setting delta_v will also set sample_size_v.

Please refer to the wiki for details on using this class member.

    Getter  Gets evaluation delta for the v-direction
    Setter  Sets evaluation delta for the v-direction
    Type    float

delta_w
Evaluation delta for the w-direction.

Evaluation delta corresponds to the step size while evaluate() function iterates on the knot vector to generate surface points. Decreasing step size results in generation of more surface points. Therefore; smaller the delta value, smoother the surface.

Please note that delta_w and sample_size_w properties correspond to the same variable with different descriptions. Therefore, setting delta_w will also set sample_size_w.

Please refer to the wiki for details on using this class member.

    Getter  Gets evaluation delta for the w-direction
    Setter  Sets evaluation delta for the w-direction
    Type    float

dimension
Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.

    Getter  Gets the spatial dimension, e.g. 2D, 3D, etc.
    Type    int
domain
Domain.

Domain is determined using the knot vector(s).

    Getter  Gets the domain
evalpts
Evaluated points.

Please refer to the wiki for details on using this class member.

    Getter  Gets the coordinates of the evaluated points
    Type    list
evaluate(**kwargs)
Evaluates the parametric volume.
evaluate_list (param_list)
Evaluates the parametric volume for an input range of \((u, v, w)\) parameter pairs.

Note: This is an abstract method and it must be implemented in the subclass.

Parameters param_list – array of parameter pairs \((u, v, w)\)

evaluate_single (param)
Evaluates the parametric surface at the given \((u, v, w)\) parameter.

Note: This is an abstract method and it must be implemented in the subclass.

Parameters param – parameter pair \((u, v, w)\)

evaluator
Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

  * **Getter** Gets the current Evaluator instance
  * **Setter** Sets the Evaluator instance
  * **Type** evaluators.AbstractEvaluator

id
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

  * **Getter** Gets the object ID
  * **Setter** Sets the object ID
  * **Type** int

knotvector
Knot vector for \(u\)-, \(v\)- and \(w\)-directions

  * **Getter** Gets the knot vector
  * **Setter** Sets the knot vector
  * **Type** list

knotvector_u
Knot vector for the \(u\)-direction.

The knot vector will be normalized to \([0, 1]\) domain if the class is initialized with \texttt{normalize\_kv=True} argument.

Please refer to the wiki for details on using this class member.
### knotvector_v
Knot vector for the v-direction.

The knot vector will be normalized to $[0, 1]$ domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

**Getter** Gets knot vector for the v-direction

**Setter** Sets knot vector for the v-direction

**Type** list

### knotvector_w
Knot vector for the w-direction.

The knot vector will be normalized to $[0, 1]$ domain if the class is initialized with `normalize_kv=True` argument.

Please refer to the wiki for details on using this class member.

**Getter** Gets knot vector for the w-direction

**Setter** Sets knot vector for the w-direction

**Type** list

### name
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter** Gets the object name

**Setter** Sets the object name

**Type** str

### opt
Dictionary for storing custom data in the current geometry object.

`opt` is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}
```

(continues on next page)
geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
range
Domain range.

Getter Gets the range

rational
Defines the rational and non-rational B-spline shapes.

Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

Getter Returns True is the B-spline object is rational (NURBS)

Type bool

render(**kwargs)
Renders the volume using the visualization component.

The visualization component must be set using vis property before calling this method.

Keyword Arguments:

• cpcolor: sets the color of the control points
• evalcolor: sets the color of the volume
• filename: saves the plot with the input name
• plot: controls plot window visibility. Default: True
• animate: activates animation (if supported). Default: False
• grid_size: grid size for voxelization. Default: (8, 8, 8)
• use_cubes: use cube voxels instead of cuboid ones. Default: False
• num_procs: number of concurrent processes for voxelization. Default: 1

The plot argument is useful when you would like to work on the command line without any window context. If plot flag is False, this method saves the plot as an image file (.png file where possible) and disables plot window popping out. If you don’t provide a file name, the name of the image file will be pulled from the configuration class.

extras argument can be used to add extra line plots to the figure. This argument expects a list of dicts in the format described below:

```python
    [dict(  # line plot 1
          points=[[1, 2, 3], [4, 5, 6]],  # list of points
          name="My line Plot 1",  # name displayed on the legend
          color="red",  # color of the line plot
          size=6.5  # size of the line plot
          ),
     dict(  # line plot 2
          points=[[7, 8, 9], [10, 11, 12]],  # list of points
          name="My line Plot 2",  # name displayed on the legend
          color="navy",  # color of the line plot
          size=12.5  # size of the line plot
          )
    ]
```
Returns the figure object

```python
reset(**kwargs)
```
Resets control points and/or evaluated points.

Keyword Arguments:
- `evalpts`: if True, then resets evaluated points
- `ctrlpts`: if True, then resets control points

**sample_size**
Sample size for both u- and v-directions.
Sample size defines the number of surface points to generate. It also sets the `delta` property.
The following figure illustrates the working principles of sample size property:

\[
\begin{align*}
\left[ u_{\text{start}}, \ldots, u_{\text{end}} \right] \\
\text{n}\_\text{sample}
\end{align*}
\]

Please refer to the wiki for details on using this class member.

**Getter** Gets sample size as a tuple of values corresponding to u-, v- and w-directions

**Setter** Sets sample size value for both u-, v- and w-directions

**Type** int

**sample_size_u**
Sample size for the u-direction.
Sample size defines the number of evaluated points to generate. It also sets the `delta_u` property.
Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the u-direction

**Setter** Sets sample size for the u-direction

**Type** int

**sample_size_v**
Sample size for the v-direction.
Sample size defines the number of evaluated points to generate. It also sets the `delta_v` property.
Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the v-direction

**Setter** Sets sample size for the v-direction

**Type** int

**sample_size_w**
Sample size for the w-direction.
Sample size defines the number of evaluated points to generate. It also sets the `delta_w` property.
Please refer to the wiki for details on using this class member.

**Getter** Gets sample size for the w-direction

**Setter** Sets sample size for the w-direction

**Type** int
set_ctrlpts

\(set\_ctrlpts\) \((ctrlpts, \*args, \*\*kwargs)\)

Sets the control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not.
It directly sets the control points member of the class, and therefore it doesn’t return any values. The input
will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will
be an array of 3 elements representing \((x, y, z)\) coordinates.

**Parameters**

- `ctrlpts` \((list)\) – input control points as a list of coordinates
- `args` \((tuple[int, int, int])\) – number of control points corresponding to each
  parametric dimension

**trims**

Trimming surfaces.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the array of trim surfaces
- **Setter** Sets the array of trim surfaces

**type**

Geometry type

Please refer to the wiki for details on using this class member.

- **Getter** Gets the geometry type
- **Type** `str`

**vis**

Visualization component.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the visualization component
- **Setter** Sets the visualization component
- **Type** `vis.VisAbstract`

**weights**

Weights.

**Note:** Only available for rational spline geometries. Getter return \(None\) otherwise.

Please refer to the wiki for details on using this class member.

- **Getter** Gets the weights
- **Setter** Sets the weights

**Low Level API**

The following classes provide the low level API for the geometry abstract base.

- `GeomdlBase`
- `Geometry`
- `SplineGeometry`
Geometry abstract base class can be used for implementation of any geometry object, whereas SplineGeometry abstract base class is designed specifically for spline geometries, including basis splines.

```python
class geomdl.abstract.GemdlBase(**kwargs)
    Bases: object

    Abstract base class for defining geomdl objects.
    This class provides the following properties:
    • type
    • id
    • name
    • dimension
    • opt

Keyword Arguments:
• id: object ID (as integer)
• precision: number of decimal places to round to. Default: 18

dimension
    Spatial dimension.

Please refer to the wiki for details on using this class member.
    Getter  Gets the spatial dimension, e.g. 2D, 3D, etc.
    Type    int

id
    Object ID (as an integer).

Please refer to the wiki for details on using this class member.
    Getter  Gets the object ID
    Setter  Sets the object ID
    Type    int

name
    Object name (as a string)

Please refer to the wiki for details on using this class member.
    Getter  Gets the object name
    Setter  Sets the object name
    Type    str

opt
    Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
```

(continues on next page)
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}  

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}

Please refer to the wiki for details on using this class member.

**Getter**  Gets the dict  
**Setter**  Adds key and value pair to the dict  
**Deleter**  Deletes the contents of the dict  

```python
opt_get(value)
```
Safely query for the value from the opt property.

**Parameters**  
value (str) – a key in the opt property  

**Returns**  
the corresponding value, if the key exists. None, otherwise.

**type**  
Geometry type  
Please refer to the wiki for details on using this class member.

**Getter**  Gets the geometry type  

**Type**  str  

```python
class geomdl.abstract.Geometry(**kwargs)
```

Bases: geomdl.abstract.GeomdlBase  

Abstract base class for defining geometry objects.  

This class provides the following properties:

- **type**  
- **id**  
- **name**  
- **dimension**  
- **evalpts**  
- **opt**  

**Keyword Arguments:**

- id: object ID (as integer)  
- precision: number of decimal places to round to.  

**Default:** 18  

**dimension**  
Spatial dimension.  

Please refer to the wiki for details on using this class member.
**Getter**  Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

**evalpts**  
Evaluated points.

Please refer to the wiki for details on using this class member.

**Getter**  Gets the coordinates of the evaluated points

**Type** list

**evaluate(** **kwargs**

Abstract method for the implementation of evaluation algorithm.

**Note:** This is an abstract method and it must be implemented in the subclass.

**id**  
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter**  Gets the object ID

**Setter**  Sets the object ID

**Type** int

**name**  
Object name (as a string)

Please refer to the wiki for details on using this class member.

**Getter**  Gets the object name

**Setter**  Sets the object name

**Type** str

**opt**  
Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in *key => value* format, where *key* is string, *value* is any Python object. You can use **opt** property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

delete geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.
Getter Gets the dict
Setter Adds key and value pair to the dict
Deleter Deletes the contents of the dict

```python
def opt_get(value):
    """Safely query for the value from the opt property."
    Parameters value (str) – a key in the opt property
    Returns the corresponding value, if the key exists. None, otherwise.
```

def type:
    """Geometry type"""
    Please refer to the wiki for details on using this class member.

    Getter Gets the geometry type

    Type str

```python
class SplineGeometry(**kwargs):
    """Abstract base class for defining spline geometry objects."""
    Bases: geomdl.abstract.Geometry

    Abstract class provides the following properties:
    • type = spline
    • id
    • name
    • rational
    • dimension
    • pdimension
    • degree
    • knotvector
    • ctrlpts
    • ctrlpts_size
    • weights (for completeness with the rational spline implementations)
    • evalpts
    • bbox
    • evaluator
    • vis
    • opt

    Keyword Arguments:
    • id: object ID (as integer)
    • precision: number of decimal places to round to. Default: 18
    • normalize_kv: if True, knot vector(s) will be normalized to [0,1] domain. Default: True
```
• **find_span_func**: default knot span finding algorithm. Default: helpers.find_span_linear()

**bbox**

Bounding box.

Evaluates the bounding box and returns the minimum and maximum coordinates.

Please refer to the wiki for details on using this class member.

  Getter  Gets the bounding box
  Type    tuple

**cpsize**

Number of control points in all parametric directions.

Note: This is an expert property for getting and setting control point size(s) of the geometry.

Please refer to the wiki for details on using this class member.

  Getter  Gets the number of control points
  Setter  Sets the number of control points
  Type    list

**ctrlpts**

Control points.

Please refer to the wiki for details on using this class member.

  Getter  Gets the control points
  Setter  Sets the control points
  Type    list

**ctrlpts_size**

Total number of control points.

  Getter  Gets the total number of control points
  Type    int

**degree**

Degree

Note: This is an expert property for getting and setting the degree(s) of the geometry.

Please refer to the wiki for details on using this class member.

  Getter  Gets the degree
  Setter  Sets the degree
  Type    list

**dimension**

Spatial dimension.

Spatial dimension will be automatically estimated from the first element of the control points array.

Please refer to the wiki for details on using this class member.
getter

Gets the spatial dimension, e.g. 2D, 3D, etc.

**Type** int

domain

Domain.

Domain is determined using the knot vector(s).

getter

Gets the domain

evalpts

Evaluated points.

Please refer to the wiki for details on using this class member.

getter

Gets the coordinates of the evaluated points

**Type** list

evaluate (**kwargs)

Abstract method for the implementation of evaluation algorithm.

**Note:** This is an abstract method and it must be implemented in the subclass.

evaluator

Evaluator instance.

Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.

Please refer to the wiki for details on using this class member.

getter

Gets the current Evaluator instance

setter

Sets the Evaluator instance

**Type** evaluators.AbstractEvaluator

id

Object ID (as an integer).

Please refer to the wiki for details on using this class member.

getter

Gets the object ID

setter

Sets the object ID

**Type** int

knotvector

Knot vector

**Note:** This is an expert property for getting and setting the knot vector(s) of the geometry.

Please refer to the wiki for details on using this class member.

getter

Gets the knot vector

setter

Sets the knot vector

**Type** list
name
Object name (as a string)
Please refer to the wiki for details on using this class member.

Getter  Gets the object name
Setter  Sets the object name
Type  str

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Please refer to the wiki for details on using this class member.

Getter  Gets the dict
Setter  Adds key and value pair to the dict
Deleter  Deletes the contents of the dict

```python
opt_get(value)
Safely query for the value from the opt property.

Parameters  value (str) – a key in the opt property

Returns  the corresponding value, if the key exists. None, otherwise.
```

pdimension
Parametric dimension.
Please refer to the wiki for details on using this class member.

Getter  Gets the parametric dimension
Type  int

range
Domain range.

Getter  Gets the range

rational
Defines the rational and non-rational B-spline shapes.
Rational shapes use homogeneous coordinates which includes a weight alongside with the Cartesian coordinates. Rational B-splines are also named as NURBS (Non-uniform rational basis spline) and non-rational B-splines are sometimes named as NUBS (Non-uniform basis spline) or directly as B-splines.

Please refer to the wiki for details on using this class member.

**Getter** Returns True if the B-spline object is rational (NURBS)

**Type** bool

**render(**kwargs**)
Abstract method for spline rendering and visualization.

**Note:** This is an abstract method and it must be implemented in the subclass.

**set_ctrlpts**(ctrlpts, *args, **kwargs)
Sets control points and checks if the data is consistent.

This method is designed to provide a consistent way to set control points whether they are weighted or not. It directly sets the control points member of the class, and therefore it doesn’t return any values. The input will be an array of coordinates. If you are working in the 3-dimensional space, then your coordinates will be an array of 3 elements representing \((x, y, z)\) coordinates.

**Keyword Arguments:**
- array_init: initializes the control points array in the instance
- array_check_for: defines the types for input validation
- callback: defines the callback function for processing input points
- dimension: defines the spatial dimension of the input points

**Parameters**
- **ctrlpts** *(list)* – input control points as a list of coordinates
- **args** *(tuple)* – number of control points corresponding to each parametric dimension

**type**
Geometry type

Please refer to the wiki for details on using this class member.

**Getter** Gets the geometry type

**Type** str

**vis**
Visualization component.

Please refer to the wiki for details on using this class member.

**Getter** Gets the visualization component

**Setter** Sets the visualization component

**Type** vis.VisAbstract

**weights**
Weights.
Note: Only available for rational spline geometries. Getter return None otherwise.

Please refer to the wiki for details on using this class member.

**Getter** Gets the weights

**Setter** Sets the weights

### 16.3.2 Evaluators

Evaluators allow users to change the evaluation algorithms that are used to evaluate curves, surfaces and volumes, take derivatives and more. All geometry classes set an evaluator by default. Users may switch between the evaluation algorithms at runtime. It is also possible to implement different algorithms (e.g. T-splines) or extend existing ones.

**How to Use**

All geometry classes come with a default specialized `evaluator` class, the algorithms are generally different for rational and non-rational geometries. The evaluator class instance can be accessed and/or updated using `evaluator` property. For instance, the following code snippet changes the evaluator of a B-Spline curve.

```
from geomdl import BSpline
from geomdl import evaluators

crv = BSpline.Curve()
cevaltr = evaluators.CurveEvaluator2()
crv.evaluator = cevaltr

# Curve "evaluate" method will use CurveEvaluator2.evaluate() method
crv.evaluate()

# Get evaluated points
curve_points = crv.evalpts
```

**Implementing Evaluators**

All evaluators should be extended from `evaluators.AbstractEvaluator` abstract base class. This class provides a point evaluation and a derivative computation methods. Both methods take a `data` input which contains the geometry data as a `dict` object (refer to `BSpline.Surface.data` property as an example). The derivative computation method also takes additional arguments, such as the parametric position and the derivative order.
Abstract Base

class geomdl.evaluators.AbstractEvaluator(**kwargs)
    Bases: object
    
    Abstract base class for implementations of fundamental spline algorithms, such as evaluate and derivative.

    **Abstract Methods:**
    - evaluate is used for computation of the complete spline shape
    - derivative_single is used for computation of derivatives at a single parametric coordinate

    Please note that this class requires the keyword argument find_span_func to be set to a valid find_span function implementation. Please see helpers module for details.

    derivatives(data_dict, parpos, deriv_order=0, **kwargs)
    
    Abstract method for evaluation of the n-th order derivatives at the input parametric position.

    **Note:** This is an abstract method and it must be implemented in the subclass.

    **Parameters**
    - **data_dict (dict)** – data dictionary containing the necessary variables
    - **parpos (list, tuple)** – parametric position where the derivatives will be computed
    - **deriv_order (int)** – derivative order; to get the i-th derivative

    evaluate(data_dict, **kwargs)
    
    Abstract method for evaluation of points on the spline geometry.

    **Note:** This is an abstract method and it must be implemented in the subclass.

    **Parameters**
    - **data_dict (dict)** – data dictionary containing the necessary variables
name
Evaluator name.
  
  **Getter**  Gets the name of the evaluator
  
  **Type**  str

Curve Evaluators

class geomdl.evaluators.CurveEvaluator(**kwargs)
  Bases: geomdl.evaluators.AbstractEvaluator
Sequential curve evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.1: CurvePoint
- Algorithm A3.2: CurveDerivsAlg1

Please note that knot vector span finding function may be changed by setting find_span_func keyword argument during the initialization. By default, this function is set to helpers.find_span_linear(). Please see Helpers Module Documentation for more details.

derivatives (datadict, parpos, deriv_order=0, **kwargs)
Evaluates the n-th order derivatives at the input parametric position.

Parameters

- **datadict** (dict) – data dictionary containing the necessary variables
- **parpos** (list, tuple) – parametric position where the derivatives will be computed
- **deriv_order** (int) – derivative order; to get the i-th derivative

Returns  evaluated derivatives
  
  Return type  list

evaluate (datadict, **kwargs)
Evaluates the curve.

**Keyword Arguments:**

- **start**: starting parametric position for evaluation
- **stop**: ending parametric position for evaluation

Parameters  datadict (dict) – data dictionary containing the necessary variables

Returns  evaluated points
  
  Return type  list

name
Evaluator name.
  
  **Getter**  Gets the name of the evaluator
  
  **Type**  str

class geomdl.evaluators.CurveEvaluatorRational(**kwargs)
  Bases: geomdl.evaluators.CurveEvaluator
Sequential rational curve evaluation algorithms.
This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.1: CurvePoint
- Algorithm A4.2: RatCurveDerivs

Please note that knot vector span finding function may be changed by setting find_span_func keyword argument during the initialization. By default, this function is set to helpers.find_span_linear(). Please see Helpers Module Documentation for more details.

**derivatives** (datadict, parpos, deriv_order=0, **kwargs)

Evaluates the n-th order derivatives at the input parametric position.

**Parameters**

- **datadict** (dict) – data dictionary containing the necessary variables
- **parpos** (list, tuple) – parametric position where the derivatives will be computed
- **deriv_order** (int) – derivative order; to get the i-th derivative

**Returns** evaluated derivatives

**Return type** list

**evaluate** (datadict, **kwargs)

Evaluates the rational curve.

**Keyword Arguments:**

- **start**: starting parametric position for evaluation
- **stop**: ending parametric position for evaluation

**Parameters** **datadict** (dict) – data dictionary containing the necessary variables

**Returns** evaluated points

**Return type** list

**name**

Evaluator name.

**Getter** Gets the name of the evaluator

**Type** str
Parameters

- `datadict (dict)` – data dictionary containing the necessary variables
- `parpos (list, tuple)` – parametric position where the derivatives will be computed
- `deriv_order (int)` – derivative order; to get the i-th derivative

Returns evaluated derivatives

Return type list

### evaluate (datadict, **kwargs)
Evaluates the curve.

Keyword Arguments:

- `start`: starting parametric position for evaluation
- `stop`: ending parametric position for evaluation

Parameters `datadict (dict)` – data dictionary containing the necessary variables

Returns evaluated points

Return type list

### name
Evaluator name.

**Getter** Gets the name of the evaluator

**Type** str

### Surface Evaluators

**class** geomdl.evaluators.SurfaceEvaluator(**kwargs)

**Bases:** geomdl.evaluators.AbstractEvaluator

Sequential surface evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.5: SurfacePoint
- Algorithm A3.6: SurfaceDerivsAlg1

Please note that knot vector span finding function may be changed by setting `find_span_func` keyword argument during the initialization. By default, this function is set to `helpers.find_span_linear()`. Please see Helpers Module Documentation for more details.

### derivatives (datadict, parpos, deriv_order=0, **kwargs)
Evaluates the n-th order derivatives at the input parametric position.

Parameters

- `datadict (dict)` – data dictionary containing the necessary variables
- `parpos (list, tuple)` – parametric position where the derivatives will be computed
- `deriv_order (int)` – derivative order; to get the i-th derivative

Returns evaluated derivatives

Return type list
**evaluate** *(datadict, **kwargs)*

Evaluates the surface.

**Keyword Arguments:**
- **start**: starting parametric position for evaluation
- **stop**: ending parametric position for evaluation

**Parameters**
- **datadict**(dict) – data dictionary containing the necessary variables

**Returns**
- evaluated points

**Return type**
- list

**name**

Evaluator name.

**Getter**

- Gets the name of the evaluator

**Type**

- str

**class**

`geomdl.evaluators.SurfaceEvaluatorRational(**kwargs)`

**Bases**: `geomdl.evaluators.SurfaceEvaluator`

Sequential rational surface evaluation algorithms.

This evaluator implements the following algorithms from **The NURBS Book**:
- Algorithm A4.3: SurfacePoint
- Algorithm A4.4: RatSurfaceDerivs

Please note that knot vector span finding function may be changed by setting `find_span_func` keyword argument during the initialization. By default, this function is set to `helpers.find_span_linear()`. Please see [Helpers Module Documentation](#) for more details.

**derivatives** *(datadict, parpos, deriv_order=0, **kwargs)*

Evaluates the n-th order derivatives at the input parametric position.

**Parameters**
- **datadict**(dict) – data dictionary containing the necessary variables
- **parpos**(list, tuple) – parametric position where the derivatives will be computed
- **deriv_order**(int) – derivative order; to get the i-th derivative

**Returns**
- evaluated derivatives

**Return type**
- list

**evaluate** *(datadict, **kwargs)*

Evaluates the rational surface.

**Keyword Arguments:**
- **start**: starting parametric position for evaluation
- **stop**: ending parametric position for evaluation

**Parameters**
- **datadict**(dict) – data dictionary containing the necessary variables

**Returns**
- evaluated points

**Return type**
- list
name
Evaluator name.

**Getter**  Gets the name of the evaluator

**Type**  str

class geomdl.evaluators.SurfaceEvaluator2(**kwargs)
Bases: geomdl.evaluators.SurfaceEvaluator

Sequential surface evaluation algorithms (alternative).

This evaluator implements the following algorithms from *The NURBS Book*:

- Algorithm A3.5: SurfacePoint
- Algorithm A3.7: SurfaceDerivCpts
- Algorithm A3.8: SurfaceDerivsAlg2

Please note that knot vector span finding function may be changed by setting find_span_func keyword argument during the initialization. By default, this function is set to `helpers.find_span_linear()`. Please see *Helpers Module Documentation* for more details.

def derivatives(datadict, parpos, deriv_order=0, **kwargs)
Evaluates the n-th order derivatives at the input parametric position.

**Parameters**

- **datadict (dict)** – data dictionary containing the necessary variables
- **parpos (list, tuple)** – parametric position where the derivatives will be computed
- **deriv_order (int)** – derivative order; to get the i-th derivative

**Returns**  evaluated derivatives

**Return type**  list

def evaluate(datadict, **kwargs)
Evaluates the surface.

**Keyword Arguments:**

- **start**: starting parametric position for evaluation
- **stop**: ending parametric position for evaluation

**Parameters**  datadict (dict) – data dictionary containing the necessary variables

**Returns**  evaluated points

**Return type**  list

name
Evaluator name.

**Getter**  Gets the name of the evaluator

**Type**  str
Volume Evaluators

class geomdl.evaluators.VolumeEvaluator(**kwargs)
    Bases: geomdl.evaluators.AbstractEvaluator

Sequential volume evaluation algorithms.

Please note that knot vector span finding function may be changed by setting find_span_func keyword argument during the initialization. By default, this function is set to helpers.find_span_linear(). Please see Helpers Module Documentation for more details.

derivatives(**kwargs)
    Evaluates the derivative at the given parametric coordinate.

evaluate (datadict, **kwargs)
    Evaluates the volume.

Keyword Arguments:

• start: starting parametric position for evaluation
• stop: ending parametric position for evaluation

Parameters datadict (dict) – data dictionary containing the necessary variables

Returns evaluated points

Return type list

name
    Evaluator name.

    Getter Gets the name of the evaluator

    Type str

class geomdl.evaluators.VolumeEvaluatorRational(**kwargs)
    Bases: geomdl.evaluators.VolumeEvaluator

Sequential rational volume evaluation algorithms.

Please note that knot vector span finding function may be changed by setting find_span_func keyword argument during the initialization. By default, this function is set to helpers.find_span_linear(). Please see Helpers Module Documentation for more details.

derivatives(**kwargs)
    Evaluates the derivatives at the input parameter.

evaluate (datadict, **kwargs)
    Evaluates the rational volume.

Keyword Arguments:

• start: starting parametric position for evaluation
• stop: ending parametric position for evaluation

Parameters datadict (dict) – data dictionary containing the necessary variables

Returns evaluated points

Return type list
name
Evaluated name.

Getter Gets the name of the evaluator

Type str

16.3.3 Utility Functions

These modules contain common utility and helper functions for B-Spline / NURBS curve and surface evaluation operations.

Utilities

The utilities module contains common utility functions for NURBS-Python library and its extensions.

geomdl.utilities.check_params(params)
Checks if the parameters are defined in the domain [0, 1].

Parameters params (list, tuple) – parameters (u, v, w)

Returns True if defined in the domain [0, 1]. False, otherwise.

Return type bool

geomdl.utilities.color_generator(seed=None)
Generates random colors for control and evaluated curve/surface points plots.

The seed argument is used to set the random seed by directly passing the value to random.seed() function. Please see the Python documentation for more details on the random module.

Inspired from https://stackoverflow.com/a/14019260

Parameters seed – Sets the random seed

Returns list of color strings in hex format

Return type list

geomdl.utilities.evaluate_bounding_box(ctrlpts)
Computes the minimum bounding box of the point set.

The (minimum) bounding box is the smallest enclosure in which all the input points lie.

Parameters ctrlpts (list, tuple) – points

Returns bounding box in the format [min, max]

Return type tuple

geomdl.utilities.make_quad(points, size_u, size_v)
Converts linear sequence of input points into a quad structure.

Parameters

• points (list, tuple) – list of points to be ordered
• size_v (int) – number of elements in a row
• size_u (int) – number of elements in a column

Returns re-ordered points

Return type list
Generates a quadtree-like structure from surface control points.

This function generates a 2-dimensional list of control point coordinates. Considering the object-oriented representation of a quadtree data structure, first dimension of the generated list corresponds to a list of QuadTree classes. Second dimension of the generated list corresponds to a QuadTree data structure. The first element of the 2nd dimension is the mid-point of the bounding box and the remaining elements are corner points of the bounding box organized in counter-clockwise order.

To maintain stability for the data structure on the edges and corners, the function accepts `extrapolate` keyword argument. If it is `True`, then the function extrapolates the surface on the corners and edges to complete the quad-like structure for each control point. If it is `False`, no extrapolation will be applied. By default, `extrapolate` is set to `True`.

Please note that this function’s intention is not generating a real quadtree structure but reorganizing the control points in a very similar fashion to make them available for various geometric operations.

**Parameters**

- `points` *(list, tuple)* – 1-dimensional array of surface control points
- `size_u` *(int)* – number of control points on the u-direction
- `size_v` *(int)* – number of control points on the v-direction

**Returns** control points organized in a quadtree-like structure

**Return type** tuple

---

**geomdl.utilities.make_zigzag(points, num_cols)**

Converts linear sequence of points into a zig-zag shape.

This function is designed to create input for the visualization software. It orders the points to draw a zig-zag shape which enables generating properly connected lines without any scanlines. Please see the below sketch on the functionality of the `num_cols` parameter:

```
|-------<-------|
|-------<-------|
|------->-------|
|-------<-------|
|------->-------|
|-------<-------|
|------->-------|
|-------<-------|
```

Please note that this function does not detect the ordering of the input points to detect the input points have already been processed to generate a zig-zag shape.

**Parameters**

- `points` *(list)* – list of points to be ordered
- `num_cols` *(int)* – number of elements in a row which the zig-zag is generated

**Returns** re-ordered points

**Return type** list

---

The `helpers` module contains common functions required for evaluating both surfaces and curves, such as basis function computations, knot vector span finding, etc.
geomdl.helpers.basis_function (degree, knot_vector, span, knot)
Computes the non-vanishing basis functions for a single parameter.

Implementation of Algorithm A2.2 from The NURBS Book by Piegl & Tiller. Uses recurrence to compute the basis functions, also known as Cox - de Boor recursion formula.

Parameters
- degree (int) – degree, $p$
- knot_vector (list, tuple) – knot vector, $U$
- span (int) – knot span, $i$
- knot (float) – knot or parameter, $u$

Returns basis functions

Return type list

geomdl.helpers.basis_function_all (degree, knot_vector, span, knot)
Computes all non-zero basis functions of all degrees from 0 up to the input degree for a single parameter.

A slightly modified version of Algorithm A2.2 from The NURBS Book by Piegl & Tiller. Wrapper for helpers.basis_function() to compute multiple basis functions. Uses recurrence to compute the basis functions, also known as Cox - de Boor recursion formula.

For instance; if degree = 2, then this function will compute the basis function values of degrees 0, 1 and 2 for the knot value at the input knot span of the knot_vector.

Parameters
- degree (int) – degree, $p$
- knot_vector (list, tuple) – knot vector, $U$
- span (int) – knot span, $i$
- knot (float) – knot or parameter, $u$

Returns basis functions

Return type list

geomdl.helpers.basis_function_ders (degree, knot_vector, span, knot, order)
Computes derivatives of the basis functions for a single parameter.

Implementation of Algorithm A2.3 from The NURBS Book by Piegl & Tiller.

Parameters
- degree (int) – degree, $p$
- knot_vector (list, tuple) – knot vector, $U$
- span (int) – knot span, $i$
- knot (float) – knot or parameter, $u$
- order (int) – order of the derivative

Returns derivatives of the basis functions

Return type list

geomdl.helpers.basis_function_ders_one (degree, knot_vector, span, knot, order)
Computes the derivative of one basis functions for a single parameter.

Implementation of Algorithm A2.5 from The NURBS Book by Piegl & Tiller.
Parameters

- **degree** (*int*) – degree, \( p \)
- **knot_vector** (*list, tuple*) – knot vector, \( U \)
- **span** (*int*) – knot span, \( i \)
- **knot** (*float*) – knot or parameter, \( u \)
- **order** (*int*) – order of the derivative

Returns basis function derivatives

Return type list

```python
geomdl.helpers.basis_function_one(degree, knot_vector, span, knot)
```
Computes the value of a basis function for a single parameter.

Implementation of Algorithm 2.4 from The NURBS Book by Piegl & Tiller.

Parameters

- **degree** (*int*) – degree, \( p \)
- **knot_vector** (*list, tuple*) – knot vector
- **span** (*int*) – knot span, \( i \)
- **knot** (*float*) – knot or parameter, \( u \)

Returns basis function, \( N_{i,p} \)

Return type float

```python
geomdl.helpers.basis_functions(degree, knot_vector, spans, knots)
```
Computes the non-vanishing basis functions for a list of parameters.

Wrapper for `helpers.basis_function()` to process multiple span and knot values. Uses recurrence to compute the basis functions, also known as Cox - de Boor recursion formula.

Parameters

- **degree** (*int*) – degree, \( p \)
- **knot_vector** (*list, tuple*) – knot vector
- **spans** (*list, tuple*) – list of knot spans
- **knots** (*list, tuple*) – list of knots or parameters

Returns basis functions

Return type list

```python
geomdl.helpers.basis_functions_ders(degree, knot_vector, spans, knots, order)
```
Computes derivatives of the basis functions for a list of parameters.

Wrapper for `helpers.basis_function_ders()` to process multiple span and knot values.

Parameters

- **degree** (*int*) – degree, \( p \)
- **knot_vector** (*list, tuple*) – knot vector, \( U \)
- **spans** (*list, tuple*) – list of knot spans
- **knots** (*list, tuple*) – list of knots or parameters
• order (int) – order of the derivative

Returns derivatives of the basis functions

Return type list

geomdl.helpers.curve_deriv_cpts(dim, degree, kv, cpts, rs, deriv_order=0)
Compute control points of curve derivatives.

Implementation of Algorithm A3.3 from The NURBS Book by Piegl & Tiller.

Parameters
• dim (int) – spatial dimension of the curve
• degree (int) – degree of the curve
• kv (list, tuple) – knot vector
• cpts (list, tuple) – control points
• rs – minimum (r1) and maximum (r2) knot spans that the curve derivative will be computed
• deriv_order (int) – derivative order, i.e. the i-th derivative

Returns control points of the derivative curve over the input knot span range

Return type list

geomdl.helpers.degree_elevation(degree, ctrlpts, **kwargs)
Computes the control points of the rational/non-rational spline after degree elevation.

Implementation of Eq. 5.36 of The NURBS Book by Piegl & Tiller, 2nd Edition, p.205

Keyword Arguments:
• num: number of degree elevations

Please note that degree elevation algorithm can only operate on Bezier shapes, i.e. curves, surfaces, volumes.

Parameters
• degree (int) – degree
• ctrlpts (list, tuple) – control points

Returns control points of the degree-elevated shape

Return type list

geomdl.helpers.degree_reduction(degree, ctrlpts, **kwargs)
Computes the control points of the rational/non-rational spline after degree reduction.

Implementation of Eqs. 5.41 and 5.42 of The NURBS Book by Piegl & Tiller, 2nd Edition, p.220

Please note that degree reduction algorithm can only operate on Bezier shapes, i.e. curves, surfaces, volumes and this implementation does NOT compute the maximum error tolerance as described via Eqs. 5.45 and 5.46 of The NURBS Book by Piegl & Tiller, 2nd Edition, p.221 to determine whether the shape is degree reducible or not.

Parameters
• degree (int) – degree
• ctrlpts (list, tuple) – control points

Returns control points of the degree-reduced shape

Return type list
geomdl.helpers.find_multiplicity(knot, knot_vector, **kwargs)
Finds knot multiplicity over the knot vector.

**Keyword Arguments:**
- tol: tolerance (delta) value for equality checking

**Parameters**
- knot (float) – knot or parameter, u
- knot_vector (list, tuple) – knot vector, U

**Returns**
- knot multiplicity, s

**Return type**
- int

geomdl.helpers.find_span_binsearch(degree, knot_vector, num_ctrlpts, knot, **kwargs)
Finds the span of the knot over the input knot vector using binary search.

Implementation of Algorithm A2.1 from The NURBS Book by Piegl & Tiller.

The NURBS Book states that the knot span index always starts from zero, i.e. for a knot vector [0, 0, 1, 1]; if FindSpan returns 1, then the knot is between the half-open interval [0, 1).

**Parameters**
- degree (int) – degree, p
- knot_vector (list, tuple) – knot vector, U
- num_ctrlpts (int) – number of control points, n + 1
- knot (float) – knot or parameter, u

**Returns**
- knot span

**Return type**
- int

geomdl.helpers.find_span_linear(degree, knot_vector, num_ctrlpts, knot, **kwargs)
Finds the span of a single knot over the knot vector using linear search.

Alternative implementation for the Algorithm A2.1 from The NURBS Book by Piegl & Tiller.

**Parameters**
- degree (int) – degree, p
- knot_vector (list, tuple) – knot vector, U
- num_ctrlpts (int) – number of control points, n + 1
- knot (float) – knot or parameter, u

**Returns**
- knot span

**Return type**
- int

geomdl.helpers.find_spans(degree, knot_vector, num_ctrlpts, knots, func=find_span_linear)
Finds spans of a list of knots over the knot vector.

**Parameters**
- degree (int) – degree, p
- knot_vector (list, tuple) – knot vector, U
- num_ctrlpts (int) – number of control points, n + 1
- **knots** *(list, tuple)* – list of knots or parameters
- **func** – function for span finding, e.g. linear or binary search

**Returns** list of spans

**Return type** list

```
geomdl.helpers.knot_insertion (degree, knotvector, ctrlpts, u, **kwargs)
```

Computes the control points of the rational/non-rational spline after knot insertion.

Part of Algorithm A5.1 of The NURBS Book by Piegl & Tiller, 2nd Edition.

**Keyword Arguments:**
- **num**: number of knot insertions. *Default: 1*
- **s**: multiplicity of the knot. *Default: computed via :func:`find_multiplicity`*
- **span**: knot span. *Default: computed via :func:`find_span_linear`*

**Parameters**
- **degree** *(int)* – degree
- **knotvector** *(list, tuple)* – knot vector
- **ctrlpts** *(list)* – control points
- **u** *(float)* – knot to be inserted

**Returns** updated control points

**Return type** list

```
geomdl.helpers.knot_insertion_alpha (u, knotvector, span, idx, leg)
```

Computes \( \alpha \) coefficient for knot insertion algorithm.

**Parameters**
- **u** *(float)* – knot
- **knotvector** *(tuple)* – knot vector
- **span** *(int)* – knot span
- **idx** *(int)* – index value (degree-dependent)
- **leg** *(int)* – i-th leg of the control points polygon

**Returns** coefficient value

**Return type** float

```
geomdl.helpers.knot_insertion_kv (knotvector, u, span, r)
```

Computes the knot vector of the rational/non-rational spline after knot insertion.

Part of Algorithm A5.1 of The NURBS Book by Piegl & Tiller, 2nd Edition.

**Parameters**
- **knotvector** *(list, tuple)* – knot vector
- **u** *(float)* – knot
- **span** *(int)* – knot span
- **r** *(int)* – number of knot insertions
Returns updated knot vector

Return type list

`geomdl.helpers.knot_refinement(degree, knotvector, ctrlpts, **kwargs)`

Computes the knot vector and the control points of the rational/non-rational spline after knot refinement.

Implementation of Algorithm A5.4 of The NURBS Book by Piegl & Tiller, 2nd Edition.

The algorithm automatically find the knots to be refined, i.e. the middle knots in the knot vector, and their multiplicities, i.e. number of same knots in the knot vector. This is the basis of knot refinement algorithm. This operation can be overridden by providing a list of knots via `knot_list` argument. In addition, users can provide a list of additional knots to be inserted in the knot vector via `add_knot_list` argument.

Moreover, a numerical `density` argument can be used to automate extra knot insertions. If `density` is bigger than 1, then the algorithm finds the middle knots in each internal knot span to increase the number of knots to be refined.

Example: Let the degree is 2 and the knot vector to be refined is \([0, 2, 4]\) with the superfluous knots from the start and end are removed. Knot vectors with the changing `density` \((d)\) value will be:

- \(d = 1\), knot vector \([0, 1, 1, 2, 2, 3, 3, 4]\)
- \(d = 2\), knot vector \([0, 0.5, 0.5, 1, 1, 1.5, 1.5, 2, 2, 2.5, 2.5, 3, 3, 3.5, 3.5, 4]\)

Keyword Arguments:
- `knot_list`: knot list to be refined. Default: list of internal knots
- `add_knot_list`: additional list of knots to be refined. Default: []
- `density`: Density of the knots. Default: 1

Parameters
- `degree (int)` – degree
- `knotvector (list, tuple)` – knot vector
- `ctrlpts` – control points

Returns updated control points and knot vector

Return type tuple

`geomdl.helpers.knot_removal(degree, knotvector, ctrlpts, u, **kwargs)`

Computes the control points of the rational/non-rational spline after knot removal.

Implementation based on Algorithm A5.8 and Equation 5.28 of The NURBS Book by Piegl & Tiller

Keyword Arguments:
- `num`: number of knot removals

Parameters
- `degree (int)` – degree
- `knotvector (list, tuple)` – knot vector
- `ctrlpts (list)` – control points
- `u (float)` – knot to be removed

Returns updated control points
Return type  list

```
geomdl.helpers.knot_removal_alpha_i
Computes $\alpha_i$ coefficient for knot removal algorithm.

Please refer to Eq. 5.29 of The NURBS Book by Piegl & Tiller, 2nd Edition, p.184 for details.

Parameters
• $u$ (float) – knot
• degree (int) – degree
• knotvector (tuple) – knot vector
• num (int) – knot removal index
• idx (int) – iterator index

Returns  coefficient value

Return type  float

geomdl.helpers.knot_removal_alpha_j
Computes $\alpha_j$ coefficient for knot removal algorithm.

Please refer to Eq. 5.29 of The NURBS Book by Piegl & Tiller, 2nd Edition, p.184 for details.

Parameters
• $u$ (float) – knot
• degree (int) – degree
• knotvector (tuple) – knot vector
• num (int) – knot removal index
• idx (int) – iterator index

Returns  coefficient value

Return type  float

geomdl.helpers.knot_removal_kv (knotvector, span, r)
Computes the knot vector of the rational/non-rational spline after knot removal.

Part of Algorithm A5.8 of The NURBS Book by Piegl & Tiller, 2nd Edition.

Parameters
• knotvector (list, tuple) – knot vector
• span (int) – knot span
• r (int) – number of knot removals

Returns  updated knot vector

Return type  list

geomdl.helpers.surface_deriv_cpts (dim, degree, kv, cpts, cpsize, rs, ss, deriv_order=0)
Compute control points of surface derivatives.

Implementation of Algorithm A3.7 from The NURBS Book by Piegl & Tiller.

Parameters
• dim (int) – spatial dimension of the surface
• **degree** (list, tuple) – degrees
• **kv** (list, tuple) – knot vectors
• **cpts** (list, tuple) – control points
• **cpsize** (list, tuple) – number of control points in all parametric directions
• **rs** (list, tuple) – minimum (r1) and maximum (r2) knot spans for the u-direction
• **ss** (list, tuple) – minimum (s1) and maximum (s2) knot spans for the v-direction
• **deriv_order** (int) – derivative order, i.e. the i-th derivative

**Returns** control points of the derivative surface over the input knot span ranges

**Return type** list

---

**Linear Algebra**

The `linalg` module contains some basic functions for point, vector and matrix operations.

Although most of the functions are designed for internal usage, the users can still use some of the functions for their advantage, especially the point and vector manipulation and generation functions. Functions related to point manipulation have `point_` prefix and the ones related to vectors have `vector_` prefix.

`geomdl.linalg.backward_substitution(matrix_u, matrix_y)`

Backward substitution method for the solution of linear systems.

Solves the equation $Ux = y$ using backward substitution method where $U$ is a upper triangular matrix and $y$ is a column matrix.

**Parameters**

• **matrix_u** (list, tuple) – U, upper triangular matrix
• **matrix_y** (list, tuple) – y, column matrix

**Returns** x, column matrix

**Return type** list

`geomdl.linalg.binomial_coefficient(k, i)`

Computes the binomial coefficient (denoted by $k \text{ choose } i$).

Please see the following website for details: http://mathworld.wolfram.com/BinomialCoefficient.html

**Parameters**

• **k** (int) – size of the set of distinct elements
• **i** (int) – size of the subsets

**Returns** combination of $k$ and $i$

**Return type** float

`geomdl.linalg.convex_hull(points)`

Returns points on convex hull in counterclockwise order according to Graham’s scan algorithm.

Reference: https://gist.github.com/arthur-e/5cf52962341310f438e96c1f3c3398b8

**Note:** This implementation only works in 2-dimensional space.
**Parameters**  
points\((list, tuple)\) – list of 2-dimensional points

**Returns**  
convex hull of the input points

**Return type**  
list

geomdl.linalg.forward_substitution\((matrix\_l, matrix\_b)\)
Forward substitution method for the solution of linear systems.
Solves the equation \(Ly = b\) using forward substitution method where \(L\) is a lower triangular matrix and \(b\) is a column matrix.

**Parameters**
- matrix\_l\((list, tuple)\) – \(L\), lower triangular matrix
- matrix\_b\((list, tuple)\) – \(b\), column matrix

**Returns**  
y, column matrix

**Return type**  
list

geomdl.linalg.frange\((start, stop, step=1.0)\)
Implementation of Python’s `range()` function which works with floats.
Reference to this implementation: https://stackoverflow.com/a/36091634

**Parameters**
- start\((float)\) – start value
- stop\((float)\) – end value
- step\((float)\) – increment

**Returns**  
float

**Return type**  
generator

geomdl.linalg.is_left\((point0, point1, point2)\)
Tests if a point is Left|On|Right of an infinite line.

Ported from the C++ version: on http://geomalgorithms.com/a03-_inclusion.html

**Note:** This implementation only works in 2-dimensional space.

**Parameters**
- point0 – Point P0
- point1 – Point P1
- point2 – Point P2

**Returns**  
\(>0\) for \(P2\) left of the line through \(P0\) and \(P1\) =0 for \(P2\) on the line \(<0\) for \(P2\) right of the line

geomdl.linalg.linspace\((start, stop, num, decimals=18)\)
Returns a list of evenly spaced numbers over a specified interval.

Inspired from Numpy’s `linspace` function: https://github.com/numpy/numpy/blob/master/numpy/core/function_base.py

**Parameters**
- start\((float)\) – starting value
• **stop** (*float*) – end value
• **num** (*int*) – number of samples to generate
• **decimals** (*int*) – number of significands

Returns a list of equally spaced numbers

Return type list

```
geomdl.linalg.lu_decomposition(matrix_a)
```

LU-Factorization method using Doolittle’s Method for solution of linear systems.

Decomposes the matrix $A$ such that $A = LU$.

The input matrix is represented by a list or a tuple. The input matrix is **2-dimensional**, i.e. list of lists of integers and/or floats.

Parameters

- **matrix_a** (*list, tuple*) – Input matrix (must be a square matrix)

Returns a tuple containing matrices $L$ and $U$

Return type tuple

```
geomdl.linalg.lu_factor(matrix_a, b)
```

Computes the solution to a system of linear equations with partial pivoting.

This function solves $Ax = b$ using LUP decomposition. $A$ is a $N \times N$ matrix, $b$ is $N \times M$ matrix of $M$ column vectors. Each column of $x$ is a solution for corresponding column of $b$.

Parameters

- **matrix_a** – matrix $A$
- **b** (*list*) – matrix of $M$ column vectors

Returns $x$, the solution matrix

Return type list

```
geomdl.linalg.lu_solve(matrix_a, b)
```

Computes the solution to a system of linear equations.

This function solves $Ax = b$ using LU decomposition. $A$ is a $N \times N$ matrix, $b$ is $N \times M$ matrix of $M$ column vectors. Each column of $x$ is a solution for corresponding column of $b$.

Parameters

- **matrix_a** – matrix $A$
- **b** (*list*) – matrix of $M$ column vectors

Returns $x$, the solution matrix

Return type list

```
geomdl.linalg.matrix_determinant(m)
```

Computes the determinant of the square matrix $M$ via LUP decomposition.

Parameters

- **m** (*list, tuple*) – input matrix

Returns determinant of the matrix

Return type float

```
geomdl.linalg.matrix_identity
```

Generates a $N \times N$ identity matrix.

Parameters

- **n** (*int*) – size of the matrix
Returns identity matrix
Return type list

geomdl.linalg.matrix_inverse(m)
Computes the inverse of the matrix via LUP decomposition.
Parameters m (list, tuple) – input matrix
Returns inverse of the matrix
Return type list

geomdl.linalg.matrix_multiply(mat1, mat2)
Matrix multiplication (iterative algorithm).
The running time of the iterative matrix multiplication algorithm is \(O(n^3)\).
Parameters
• mat1 (list, tuple) – 1st matrix with dimensions \((n \times p)\)
• mat2 (list, tuple) – 2nd matrix with dimensions \((p \times m)\)
Returns resultant matrix with dimensions \((n \times m)\)
Return type list

geomdl.linalg.matrix_pivot(m, sign=False)
Computes the pivot matrix for \(M\), a square matrix.
This function computes
• the permutation matrix, \(P\)
• the product of \(M\) and \(P\), \(M \times P\)
• determinant of \(P\), \(det(P)\) if \(sign = True\)
Parameters
• m (list, tuple) – input matrix
• sign (bool) – flag to return the determinant of the permutation matrix, \(P\)
Returns a tuple containing the matrix product of \(M \times P\), \(P\) and \(det(P)\)
Return type tuple

geomdl.linalg.matrix_scalar(m, sc)
Matrix multiplication by a scalar value (iterative algorithm).
The running time of the iterative matrix multiplication algorithm is \(O(n^2)\).
Parameters
• m (list, tuple) – input matrix
• sc (int, float) – scalar value
Returns resultant matrix
Return type list

geomdl.linalg.matrix_transpose(m)
Transposes the input matrix.
The input matrix \(m\) is a 2-dimensional array.
**Parameters**  
\( m (\text{list, tuple}) \) – input matrix with dimensions \( (n \times m) \)

**Returns**  
transpose matrix with dimensions \( (m \times n) \)

**Return type**  
list

**geomdl.linalg.point_distance(pt1, pt2)**  
Computes distance between two points.

**Parameters**  
- \( \text{pt1 (list, tuple)} \) – point 1
- \( \text{pt2 (list, tuple)} \) – point 2

**Returns**  
distance between input points

**Return type**  
float

**geomdl.linalg.point_mid(pt1, pt2)**  
Computes the midpoint of the input points.

**Parameters**  
- \( \text{pt1 (list, tuple)} \) – point 1
- \( \text{pt2 (list, tuple)} \) – point 2

**Returns**  
midpoint

**Return type**  
list

**geomdl.linalg.point_translate(point_in, vector_in)**  
Translates the input points using the input vector.

**Parameters**  
- \( \text{point_in (list, tuple)} \) – input point
- \( \text{vector_in (list, tuple)} \) – input vector

**Returns**  
translated point

**Return type**  
list

**geomdl.linalg.triangle_center(tri, uv=False)**  
Computes the center of mass of the input triangle.

**Parameters**  
- \( \text{tri (elements.Triangle)} \) – triangle object
- \( \text{uv (bool)} \) – if True, then finds parametric position of the center of mass

**Returns**  
center of mass of the triangle

**Return type**  
tuple

**geomdl.linalg.triangle_normal(tri)**  
Computes the (approximate) normal vector of the input triangle.

**Parameters**  
- \( \text{tri (elements.Triangle)} \) – triangle object

**Returns**  
normal vector of the triangle

**Return type**  
tuple
geomdl.linalg.vector_angle_between(vector1, vector2, **kwargs)
Computes the angle between the two input vectors.

If the keyword argument degrees is set to True, then the angle will be in degrees. Otherwise, it will be in radians. By default, degrees is set to True.

Parameters
- vector1 (list, tuple) – vector
- vector2 (list, tuple) – vector

Returns angle between the vectors

Return type float

geomdl.linalg.vector_cross(vector1, vector2)
Computes the cross-product of the input vectors.

Parameters
- vector1 (list, tuple) – input vector 1
- vector2 (list, tuple) – input vector 2

Returns result of the cross product

Return type tuple

geomdl.linalg.vector_dot(vector1, vector2)
Computes the dot-product of the input vectors.

Parameters
- vector1 (list, tuple) – input vector 1
- vector2 (list, tuple) – input vector 2

Returns result of the dot product

Return type float

geomdl.linalg.vector_generate(start_pt, end_pt, normalize=False)
Generates a vector from 2 input points.

Parameters
- start_pt (list, tuple) – start point of the vector
- end_pt (list, tuple) – end point of the vector
- normalize (bool) – if True, the generated vector is normalized

Returns a vector from start_pt to end_pt

Return type list

geomdl.linalg.vector_is_zero(vector_in, tol=1e-07)
Checks if the input vector is a zero vector.

Parameters
- vector_in (list, tuple) – input vector
- tol (float) – tolerance value

Returns True if the input vector is zero, False otherwise

Return type bool
geomdl.linalg.vector_magnitude(vector_in)

Computes the magnitude of the input vector.

**Parameters**

- **vector_in** *(list, tuple)* – input vector

**Returns**

magnitude of the vector

**Return type**

float

geomdl.linalg.vector_mean(*args)

Computes the mean (average) of a list of vectors.

The function computes the arithmetic mean of a list of vectors, which are also organized as a list of integers or floating point numbers.

```python
# Import geomdl.utilities module
import geomdl.utilities

# Create a list of vectors as an example
vector_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Compute mean vector
mean_vector = utilities.vector_mean(*vector_list)

# Alternative usage example (same as above):
mean_vector = utilities.vector_mean([1, 2, 3], [4, 5, 6], [7, 8, 9])
```

**Parameters**

- **args** *(list, tuple)* – list of vectors

**Returns**

mean vector

**Return type**

list

geomdl.linalg.vector_multiply(vector_in, scalar)

Multiplies the vector with a scalar value.

This operation is also called *vector scaling*.

**Parameters**

- **vector_in** *(list, tuple)* – vector
- **scalar** *(int, float)* – scalar value

**Returns**

updated vector

**Return type**

tuple

geomdl.linalg.vector_normalize(vector_in, decimals=18)

Generates a unit vector from the input.

**Parameters**

- **vector_in** *(list, tuple)* – vector to be normalized
- **decimals** *(int)* – number of significands

**Returns**

the normalized vector (i.e. the unit vector)

**Return type**

list

geomdl.linalg.vector_sum(vector1, vector2, coeff=1.0)

Sums the vectors.
This function computes the result of the vector operation $\mathbf{v}_1 + c \cdot \mathbf{v}_2$, where $\mathbf{v}_1$ is `vector1`, $\mathbf{v}_2$ is `vector2` and $c$ is `coeff`.

**Parameters**
- `vector1 (list, tuple)` – vector 1
- `vector2 (list, tuple)` – vector 2
- `coeff (float)` – multiplier for vector 2

**Returns** updated vector

**Return type** `list`

```python
geomdl.linalg.wn_poly(point, vertices)
```

Winding number test for a point in a polygon.

Ported from the C++ version: [http://geomalgorithms.com/a03-_inclusion.html](http://geomalgorithms.com/a03-_inclusion.html)

**Note:** This implementation only works in 2-dimensional space.

**Parameters**
- `point (list, tuple)` – point to be tested
- `vertices (list, tuple)` – vertex points of a polygon $\text{vertices}[n+1] = \text{vertices}[0]$

**Returns** True if the point is inside the input polygon, False otherwise

**Return type** `bool`

### 16.3.4 Voxelization

New in version 5.0.

`voxelize` module provides functions for voxelizing NURBS volumes. `voxelize()` also supports multi-threaded operations via `multiprocessing` module.

**Function Reference**

```python
geomdl.voxelize.voxelize(obj, **kwargs)
```

Generates binary voxel representation of the surfaces and volumes.

**Keyword Arguments:**
- `grid_size`: size of the voxel grid. *Default:* (8, 8, 8)
- `padding`: voxel padding for in-outs finding. *Default:* 10e-8
- `use_cubes`: use cube voxels instead of cuboid ones. *Default:* `False`
- `num_procs`: number of concurrent processes for voxelization. *Default:* 1

**Parameters** `obj (abstract.Surface or abstract.Volume)` – input surface(s) or volume(s)

**Returns** voxel grid and filled information

**Return type** `tuple`
NURBS-Python Documentation, Release 5.2.9

geomdl.voxelize.save_voxel_grid(voxel_grid, file_name)
Saves binary voxel grid as a binary file.

The binary file is structured in little-endian unsigned int format.

Parameters
• voxel_grid(list, tuple) – binary voxel grid
• file_name(str) – file name to save

16.3.5 Geometric Entities

The geometric entities are used for advanced algorithms, such as tessellation. The AbstractEntity class provides the abstract base for all geometric and topological entities.

This module provides the following geometric and topological entities:
• Vertex
• Triangle
• Quad
• Face
• Body

Class Reference

class geomdl.elements.Vertex(*args, **kwargs)
Bases: geomdl.elements.AbstractEntity

3-dimensional Vertex entity with spatial and parametric position.

data
(x,y,z) components of the vertex.

Getter Gets the 3-dimensional components

Setter Sets the 3-dimensional components

id
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

Getter Gets the object ID

Setter Sets the object ID

Type int

inside
Inside-outside flag

Getter Gets the flag

Setter Sets the flag

Type bool
name
Object name (as a string)

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name
- **Type** str

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
g geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
g geom.opt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}

geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}

geom.opt = ['body_id', None]  # deletes "body_id"
print(geom.opt)  # will print: {}```

- **Getter** Gets the dict
- **Setter** Adds key and value pair to the dict
- **Deleter** Deletes the contents of the dict

```python
defopt_get(value)
Safely query for the value from the opt property.

- **Parameters** value (str) – a key in the opt property
- **Returns** the corresponding value, if the key exists. None, otherwise.
```

u
Parametric u-component of the vertex

- **Getter** Gets the u-component of the vertex
- **Setter** Sets the u-component of the vertex
- **Type** float

uv
Parametric (u,v) pair of the vertex

- **Getter** Gets the uv-component of the vertex
- **Setter** Sets the uv-component of the vertex
- **Type** list, tuple
**v**
Parametric v-component of the vertex

- **Getter** Gets the v-component of the vertex
- **Setter** Sets the v-component of the vertex
- **Type** float

**x**
x-component of the vertex

- **Getter** Gets the x-component of the vertex
- **Setter** Sets the x-component of the vertex
- **Type** float

**y**
y-component of the vertex

- **Getter** Gets the y-component of the vertex
- **Setter** Sets the y-component of the vertex
- **Type** float

**z**
z-component of the vertex

- **Getter** Gets the z-component of the vertex
- **Setter** Sets the z-component of the vertex
- **Type** float

```python
class geomdl.elements.Triangle(*args, **kwargs)

Bases: geomdl.elements.AbstractEntity

Triangle entity which represents a triangle composed of vertices.

A Triangle entity stores the vertices in its data structure. `data` returns the vertex IDs and `vertices` return the `Vertex` instances that compose the triangular structure.

**add_vertex(**args**)

Adds vertices to the Triangle object.

This method takes a single or a list of vertices as its function arguments.

**data**

Vertices composing the triangular structure.

- **Getter** Gets the vertex indices (as int values)
- **Setter** Sets the vertices (as Vertex objects)

**edges**

Edges of the triangle

- **Getter** Gets the list of vertices that generates the edges of the triangle
- **Type** list

**id**

Object ID (as an integer).

Please refer to the wiki for details on using this class member.
Getter  Gets the object ID
Setter  Sets the object ID
Type    int

inside
Inside-outside flag
  Getter  Gets the flag
  Setter  Sets the flag
  Type    bool

name
Object name (as a string)
Please refer to the wiki for details on using this class member.
  Getter  Gets the object name
  Setter  Sets the object name
  Type    str

opt
Dictionary for storing custom data in the current geometry object.

opt is a wrapper to a dict in key => value format, where key is string, value is any Python object. You can use opt property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}
geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}
geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

Getter  Gets the dict
Setter  Adds key and value pair to the dict
Deleter  Deletes the contents of the dict

opt_get (value)
Safely query for the value from the opt property.

Parameters value (str) – a key in the opt property

Returns  the corresponding value, if the key exists. None, otherwise.

vertex_ids
Vertex indices
**Note:** Please use `data` instead of this property.

**Getter** Gets the vertex indices  
**Type** list

**vertices**  
Vertices of the triangle  
**Getter** Gets the list of vertices  
**Type** tuple

**vertices_closed**  
Vertices which generates a closed triangle  
Adds the first vertex as a last element of the return value (good for plotting)  
**Getter** Gets the list of vertices  
**Type** list

```python
class geomdl.elements.Quad(*args, **kwargs)
Bases: geomdl.elements.AbstractEntity
```

Quad entity which represents a quadrilateral structure composed of vertices.  
A Quad entity stores the vertices in its data structure. `data` returns the vertex IDs and `vertices` return the `Vertex` instances that compose the quadrilateral structure.

**add_vertex** (*args*)  
Adds vertices to the Quad object.  
This method takes a single or a list of vertices as its function arguments.

**data**  
Vertices composing the quadrilateral structure.  
**Getter** Gets the vertex indices (as int values)  
**Setter** Sets the vertices (as Vertex objects)

**id**  
Object ID (as an integer).  
Please refer to the wiki for details on using this class member.  
**Getter** Gets the object ID  
**Setter** Sets the object ID  
**Type** int

**name**  
Object name (as a string)  
Please refer to the wiki for details on using this class member.  
**Getter** Gets the object name  
**Setter** Sets the object name  
**Type** str

16.3. Advanced API
**opt**

Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
gem.geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
gem.geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(gem.geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}
del gem.geom.opt  # deletes the contents of the hash map
print(gem.geom.opt)  # will print: {}
gem.geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
gem.geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(gem.geom.opt)  # will print: {'body_id': 12}
gem.geom.opt = ["body_id", None]  # deletes "body_id"
print(gem.geom.opt)  # will print: {}
```

**Getter**  Gets the dict  
**Setter**  Adds key and value pair to the dict  
**Deleter**  Deletes the contents of the dict

**opt_get**(value)

Safely query for the value from the `opt` property.

**Parameters**  
`value (str)` – a key in the `opt` property  

**Returns**  the corresponding value, if the key exists. None, otherwise.

**vertices**  
Vertices composing the quadrilateral structure.

**Getter**  Gets the vertices

**class**  
`geomdl.elements.Face(*args, **kwargs)`  
Bases: `geomdl.elements.AbstractEntity`

Representation of Face entity which is composed of triangles or quads.

**add_triangle**(args)

Adds triangles to the Face object.

This method takes a single or a list of triangles as its function arguments.

**id**  
Object ID (as an integer).

Please refer to the wiki for details on using this class member.

**Getter**  Gets the object ID  
**Setter**  Sets the object ID  
**Type**  int

**name**  
Object name (as a string)
Please refer to the wiki for details on using this class member.

**Getter**  Gets the object name

**Setter**  Sets the object name

**Type**  str

**opt**

Dictionary for storing custom data in the current geometry object.

**opt** is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
geom.opt = ["face_id", 4]  # creates "face_id" key and sets its value to an integer
geom.opt = ["contents", "data values"]  # creates "face_id" key and sets its value to a string
print(geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}
del geom.opt  # deletes the contents of the hash map
print(geom.opt)  # will print: {}
geom.opt = ["body_id", 1]  # creates "body_id" key and sets its value to 1
geom.opt = ["body_id", 12]  # changes the value of "body_id" to 12
print(geom.opt)  # will print: {'body_id': 12}
geom.opt = ["body_id", None]  # deletes "body_id"
print(geom.opt)  # will print: {}
```

**Getter**  Gets the dict

**Setter**  Adds key and value pair to the dict

**Deleter**  Deletes the contents of the dict

**opt_get** (*value*)

Safely query for the value from the `opt` property.

**Parameters**  `value` (*str*) – a key in the `opt` property

**Returns**  the corresponding value, if the key exists. `None`, otherwise.

**triangles**

Triangles of the face

**Getter**  Gets the list of triangles

**Type**  tuple

**Class**  `geomdl.elements.Body` (*args, **kwargs*)

**Bases:**  `geomdl.elements.AbstractEntity`

Representation of Body entity which is composed of faces.

**add_face** (*args*)

Adds faces to the Body object.

This method takes a single or a list of faces as its function arguments.

**faces**

Faces of the body

**Getter**  Gets the list of faces
Type `tuple`

**id**

Object ID (as an integer).

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object ID
- **Setter** Sets the object ID

Type `int`

**name**

Object name (as a string)

Please refer to the wiki for details on using this class member.

- **Getter** Gets the object name
- **Setter** Sets the object name

Type `str`

**opt**

Dictionary for storing custom data in the current geometry object.

`opt` is a wrapper to a dict in `key => value` format, where `key` is string, `value` is any Python object. You can use `opt` property to store custom data inside the geometry object. For instance:

```python
gem.geom.opt = ['face_id', 4]  # creates "face_id" key and sets its value to an integer
geom.geom.opt = ['contents', 'data values']  # creates "face_id" key and sets its value to a string
print(geom.geom.opt)  # will print: {'face_id': 4, 'contents': 'data values'}

del geom.geom.opt  # deletes the contents of the hash map
print(geom.geom.opt)  # will print: {}

geom.geom.opt = ['body_id', 1]  # creates "body_id" key and sets its value to 1
geom.geom.opt = ['body_id', 12]  # changes the value of "body_id" to 12
print(geom.geom.opt)  # will print: {'body_id': 12}

geom.geom.opt = ['body_id', None]  # deletes "body_id"
print(geom.geom.opt)  # will print: {}
```

- **Getter** Gets the dict
- **Setter** Adds key and value pair to the dict
- **Deleter** Deletes the contents of the dict

```python
def opt_get(value):
    """ Safely query for the value from the opt property. """
    Parameters
    value (str) -- a key in the opt property
    Returns
    the corresponding value, if the key exists. None, otherwise.
```
16.3.6 Ray Module

*ray* module provides utilities for ray operations. A ray (half-line) is defined by two distinct points represented by *Ray* class. This module also provides a function to compute intersection of 2 rays.

**Function and Class Reference**

```python
class geomdl.ray.Ray(point1, point2)
    Representation of a n-dimensional ray generated from 2 points.
    A ray is defined by r(t) = p1 + t × d where 't' is the parameter value, d = p2 − p1 is the vector component of the ray, p1 is the origin point and p2 is the second point which is required to define a line segment.

    Parameters
    • point1 (list, tuple) – 1st point of the line segment
    • point2 (list, tuple) – 2nd point of the line segment

    d
    Vector component of the ray (d)

    Please refer to the wiki for details on using this class member.

    Getter  Gets the vector component of the ray

dimension
    Spatial dimension of the ray

    Please refer to the wiki for details on using this class member.

    Getter  Gets the dimension of the ray

eval(t=0)
    Finds the point on the line segment defined by the input parameter.
    t = 0 returns the origin (1st) point, defined by the input argument point1 and t = 1 returns the end (2nd) point, defined by the input argument point2.

    Parameters  t (float) – parameter

    Returns  point at the parameter value

    Return type  tuple

P
    Origin point of the ray (p)

    Please refer to the wiki for details on using this class member.

    Getter  Gets the origin point of the ray

points
    Start and end points of the line segment that the ray was generated

    Please refer to the wiki for details on using this class member.

    Getter  Gets the points
```

```python
class geomdl.ray.RayIntersection
    The status of the ray intersection operation
```
geomdl.ray.intersect(ray1, ray2, **kwargs)

Finds intersection of 2 rays.

This function finds the parameter values for the 1st and 2nd input rays and returns a tuple of (parameter for ray1, parameter for ray2, intersection status). status value is a enum type which reports the case which the intersection operation encounters.

The intersection operation can encounter 3 different cases:

- Intersecting: This is the anticipated solution. Returns (t1, t2, RayIntersection.INTERSECT)
- Colinear: The rays can be parallel or coincident. Returns (t1, t2, RayIntersection.COLINEAR)
- Skew: The rays are neither parallel nor intersecting. Returns (t1, t2, RayIntersection.SKEW)

For the colinear case, t1 and t2 are the parameter values that give the starting point of the ray2 and ray1, respectively. Therefore;

```python
ray1.eval(t1) == ray2.p
ray2.eval(t2) == ray1.p
```

Please note that this operation is only implemented for 2- and 3-dimensional rays.

**Parameters**

- ray1 – 1st ray
- ray2 – 2nd ray

**Returns** a tuple of the parameter (t) for ray1 and ray2, and status of the intersection

**Return type** tuple
NURBS-Python provides an abstract base for visualization modules. It is a part of the Core Library and it can be used to implement various visualization backends.

NURBS-Python comes with the following visualization modules:

### 17.1 Visualization Base

The visualization component in the NURBS-Python package provides an easy way to visualise the surfaces and the 2D/3D curves generated using the library. The following are the list of abstract classes for the visualization system and its configuration.

#### 17.1.1 Class Reference

Abstract base class for visualization

Defines an abstract base for NURBS-Python (geomdl) visualization modules.

**param config** configuration class

**type config** VisConfigAbstract

geomdl.vis.VisAbstract.ctrlpts_offset

Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

**Getter** Gets the offset value

**Setter** Sets the offset value

**Type** float

geomdl.vis.VisAbstract.mconf

Configuration directives for the visualization module (internal).
This property controls the internal configuration of the visualization module. It is for advanced use and testing only.

The visualization module is mainly designed to plot the control points (\texttt{ctrlpts}) and the surface points (\texttt{evalpts}). These are called as \textit{plot types}. However, there is more than one way to plot the control points and the surface points. For instance, a control points plot can be a scatter plot or a quad mesh, and a surface points plot can be a scatter plot or a tessellated surface plot.

This function allows you to change the type of the plot, e.g. from scatter plot to tessellated surface plot. On the other than, some visualization modules also defines some specialized classes for this purpose as it might not be possible to change the type of the plot at the runtime due to visualization library internal API differences (i.e. different backends for 2- and 3-dimensional plots).

By default, the following plot types and values are available:

\textbf{Curve}:
- For control points (\texttt{ctrlpts}): points
- For evaluated points (\texttt{evalpts}): points

\textbf{Surface}:
- For control points (\texttt{ctrlpts}): points, quads
- For evaluated points (\texttt{evalpts}): points, quads, triangles

\textbf{Volume}:
- For control points (\texttt{ctrlpts}): points
- For evaluated points (\texttt{evalpts}): points, voxels

\textbf{Getter} Gets the visualization module configuration
\textbf{Setter} Sets the visualization module configuration

\texttt{geomdl.vis.VisAbstract.vconf}
User configuration class for visualization
\textbf{Getter} Gets the user configuration class
\textbf{Type} \texttt{vis.VisConfigAbstract}

Abstract base class for user configuration of the visualization module
Defines an abstract base for NURBS-Python (geomdl) visualization configuration.

## 17.2 Matplotlib Implementation

This module provides Matplotlib visualization implementation for NURBS-Python.

\textbf{Note}: Please make sure that you have installed \texttt{matplotlib} package before using this visualization module.

### 17.2.1 Class Reference

\texttt{class geomdl.visualization.VisMPL.VisConfig(**kwargs)}
\texttt{Bases: geomdl.vis.VisConfigAbstract}
Configuration class for Matplotlib visualization module.

This class is only required when you would like to change the visual defaults of the plots and the figure, such as hiding control points plot or legend.

The VisMPL module has the following configuration variables:

- `ctrlpts` (bool): Control points polygon/grid visibility. Default: True
- `evalpts` (bool): Curve/surface points visibility. Default: True
- `bbox` (bool): Bounding box visibility. Default: False
- `legend` (bool): Figure legend visibility. Default: True
- `axes` (bool): Axes and figure grid visibility. Default: True
- `labels` (bool): Axis labels visibility. Default: True
- `trims` (bool): Trim curves visibility. Default: True
- `axes_equal` (bool): Enables or disables equal aspect ratio for the axes. Default: True
- `figure_size` (list): Size of the figure in (x, y). Default: [10, 8]
- `figure_dpi` (int): Resolution of the figure in DPI. Default: 96
- `trim_size` (int): Size of the trim curves. Default: 20
- `alpha` (float): Opacity of the evaluated points. Default: 1.0

There is also a `debug` configuration variable which currently adds quiver plots to 2-dimensional curves to show their directions.

The following example illustrates the usage of the configuration class.

```python
# Create a curve (or a surface) instance
curve = NURBS.Curve()

# Skipping degree, knot vector and control points assignments

# Create a visualization configuration instance with no legend, no axes and set_the resolution to 120 dpi
vis_config = VisMPL.VisConfig(legend=False, axes=False, figure_dpi=120)

# Create a visualization method instance using the configuration above
vis_obj = VisMPL.VisCurve2D(vis_config)

# Set the visualization method of the curve object
curve.vis = vis_obj

# Plot the curve
curve.render()
```

Please refer to the Examples Repository for more details.

```python
static save_figure_as(fig, filename)
    Saves the figure as a file.
    Parameters
    - `fig` – a Matplotlib figure instance
    - `filename` – file name to save
```
static set_axes_equal(ax)
Sets equal aspect ratio across the three axes of a 3D plot.

Contributed by Xuefeng Zhao.

Parameters

ax – a Matplotlib axis, e.g., as output from plt.gca().

class geomdl.visualization.VisMPL.VisCurve2D(config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)
Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for 2D curves

add(ptsarr, plot_type, name=", color="", idx=0)
Adds points sets to the visualization instance for plotting.

Parameters

• ptsarr (list, tuple) – control or evaluated points
• plot_type (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
• name (str) – name of the plot displayed on the legend
• color (int) – plot color
• color – plot index

animate(**kwargs)
Generates animated plots (if supported).

If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call render() method by default.

clear()
Clears the points, colors and names lists.

ctrlpts_offset
Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

Getter
Gets the offset value

Setter
Sets the offset value

Type
float

render(**kwargs)
Plots the 2D curve and the control points polygon.

size(plot_type)
Returns the number of plots defined by the plot type.

Parameters

plot_type (str) – plot type

Returns
number of plots defined by the plot type

Return type
int

vconf
User configuration class for visualization

Getter
Gets the user configuration class

Type
vis.VisConfigAbstract
class geomdl.visualization.VisMPL.VisCurve3D(config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for 3D curves.

add(ptsarr, plot_type, name='', color='', idx=0)

Adds points sets to the visualization instance for plotting.

Parameters

- **ptsarr** (list, tuple) – control or evaluated points
- **plot_type** (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- **name** (str) – name of the plot displayed on the legend
- **color** (int) – plot color
- **idx** – plot index

animate(**kwargs)

Generates animated plots (if supported).

If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call render() method by default.

clear()

Clears the points, colors and names lists.

ctrlpts_offset

Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

- **Getter** Gets the offset value
- **Setter** Sets the offset value

Type float

render(**kwargs)

Plots the 3D curve and the control points polygon.

size(plot_type)

Returns the number of plots defined by the plot type.

Parameters **plot_type** (str) – plot type

Returns number of plots defined by the plot type

Return type int

vconf

User configuration class for visualization

- **Getter** Gets the user configuration class

Type vis.VisConfigAbstract

class geomdl.visualization.VisMPL.VisSurfScatter(config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for surfaces.

Wireframe plot for the control points and scatter plot for the surface points.
add\(\) \(\text{ptsarr, plot\_type, name=", color=", idx=0}\) 
Adds points sets to the visualization instance for plotting.

**Parameters**

- `ptsarr` (list, tuple) – control or evaluated points
- `plot\_type` (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- `name` (str) – name of the plot displayed on the legend
- `color` (int) – plot color
- `idx` – plot index

**animate\(\)** \((**kwargs))
Generates animated plots (if supported).
If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call `render\()` method by default.

**clear\()**
Clears the points, colors and names lists.

**ctrlpts\_offset**
Defines an offset value for the control points grid plots
Only makes sense to use with surfaces with dense control points grid.

**getter**
Gets the offset value
**setter**
Sets the offset value
**Type** float

**render\(\)** \((**kwargs))
Plots the surface and the control points grid.

**size\(\)\(\)\(\text{plot\_type}\)**
Returns the number of plots defined by the plot type.

**getter**
Gets the number of plots defined by the plot type
**Type** int

**vconf**
User configuration class for visualization

**getter**
Gets the user configuration class
**Type** `vis.VisConfigAbstract`

```
geomdl.visualization.VisMPL.VisSurfTriangle
alias of geomdl.visualization.VisMPL.VisSurface

class geomdl.visualization.VisMPL.VisSurfWireframe (config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)
```

Matplotlib visualization module for surfaces.
Scatter plot for the control points and wireframe plot for the surface points.

add\(\) \(\text{ptsarr, plot\_type, name=", color=", idx=0}\)
Adds points sets to the visualization instance for plotting.
Parameters

- **ptsarr** ([list, tuple]) – control or evaluated points
- **plot_type** (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- **name** (str) – name of the plot displayed on the legend
- **color** (int) – plot color
- **color** – plot index

**animate(**kwargs)**
Generates animated plots (if supported).
If the implemented visualization module supports animations, this function will create an animated figure.
Otherwise, it will call **render()** method by default.

**clear()**
Clears the points, colors and names lists.

**ctrlpts_offset**
Defines an offset value for the control points grid plots
Only makes sense to use with surfaces with dense control points grid.
- Getter Gets the offset value
- Setter Sets the offset value
- Type float

**render(**kwargs)**
Plots the surface and the control points grid.

**size(plot_type)**
Returns the number of plots defined by the plot type.
- Parameters plot_type (str) – plot type
- Returns number of plots defined by the plot type
- Return type int

**vconf**
User configuration class for visualization
- Getter Gets the user configuration class
- Type vis.VisConfigAbstract

```python
class geomdl.visualization.VisMPL.VisSurface(config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)
```

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for surfaces.

Wireframe plot for the control points and triangulated plot (using **plot_trisurf**) for the surface points. The surface is triangulated externally using **utilities.make_triangle_mesh()** function.

**add(ptsarr, plot_type, name=", color="", idx=0)**
Adds points sets to the visualization instance for plotting.

- Parameters
  - **ptsarr** ([list, tuple]) – control or evaluated points
  - **plot_type** (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
• **name** *(str)* – name of the plot displayed on the legend

• **color** *(int)* – plot color

• **color** – plot index

**animate** (**kwargs**)

Animates the surface.

This function only animates the triangulated surface. There will be no other elements, such as control points grid or bounding box.

**Keyword arguments:**

• **colormap**: applies colormap to the surface

Colormaps are a visualization feature of Matplotlib. They can be used for several types of surface plots via the following import statement: `from matplotlib import cm`

The following link displays the list of Matplotlib colormaps and some examples on colormaps: [https://matplotlib.org/tutorials/colors/colormaps.html](https://matplotlib.org/tutorials/colors/colormaps.html)

**clear** ()

Clears the points, colors and names lists.

**ctrlpts_offset**

Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

**Getter** Gets the offset value

**Setter** Sets the offset value

**Type** float

**render** (**kwargs**)

Plots the surface and the control points grid.

**Keyword arguments:**

• **colormap**: applies colormap to the surface

Colormaps are a visualization feature of Matplotlib. They can be used for several types of surface plots via the following import statement: `from matplotlib import cm`

The following link displays the list of Matplotlib colormaps and some examples on colormaps: [https://matplotlib.org/tutorials/colors/colormaps.html](https://matplotlib.org/tutorials/colors/colormaps.html)

**size** *(plot_type)*

Returns the number of plots defined by the plot type.

**Parameters** **plot_type** *(str)* – plot type

**Returns** number of plots defined by the plot type

**Return type** int

**vconf**

User configuration class for visualization

**Getter** Gets the user configuration class

**Type** vis.VisConfigAbstract
class geomdl.visualization.VisMPL.VisVolume (config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for volumes.

add(ptsarr, plot_type, name='', color='', idx=0)

Adds points sets to the visualization instance for plotting.

Parameters

- ptsarr (list, tuple) – control or evaluated points
- plot_type (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- name (str) – name of the plot displayed on the legend
- color (int) – plot color
- color – plot index

animate(**kwargs)

Generates animated plots (if supported).

If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call render() method by default.

clear()  
Clears the points, colors and names lists.

ctrlpts_offset

Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

    Getter  Gets the offset value
    Setter  Sets the offset value
    Type  float

render(**kwargs)

Plots the volume and the control points.

size(plot_type)

Returns the number of plots defined by the plot type.

    Parameters plot_type (str) – plot type
    Returns number of plots defined by the plot type
    Return type  int

vconf

User configuration class for visualization

    Getter  Gets the user configuration class
    Type  vis.VisConfigAbstract

class geomdl.visualization.VisMPL.VisVoxel (config=<geomdl.visualization.VisMPL.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for voxel representation of the volumes.

add(ptsarr, plot_type, name='', color='', idx=0)

Adds points sets to the visualization instance for plotting.
Parameters

- `ptsarr (list, tuple)` – control or evaluated points
- `plot_type (str)` – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- `name (str)` – name of the plot displayed on the legend
- `color (int)` – plot color
- `color` – plot index

`animate (**kwargs)`
Generates animated plots (if supported).

If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call `render()` method by default.

`clear()`
Clears the points, colors and names lists.

`ctrlpts_offset`
Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

- **Getter** Gets the offset value
- **Setter** Sets the offset value
- **Type** float

`render (**kwargs)`
Displays the voxels and the control points.

`size (plot_type)`
Returns the number of plots defined by the plot type.

- **Parameters** `plot_type (str)` – plot type
- **Returns** number of plots defined by the plot type
- **Return type** int

`vconf`
User configuration class for visualization

- **Getter** Gets the user configuration class
- **Type** vis.VisConfigAbstract

### 17.3 Plotly Implementation

This module provides Plotly visualization implementation for NURBS-Python.

**Note:** Please make sure that you have installed plotly package before using this visualization module.
17.3.1 Class Reference

17.4 VTK Implementation

New in version 5.0.

This module provides VTK visualization implementation for NURBS-Python.

Note: Please make sure that you have installed vtk package before using this visualization module.

17.4.1 Class Reference

class geomdl.visualization.VisVTK.VisConfig(**kwargs)

Bases: geomdl.vis.VisConfigAbstract

Configuration class for VTK visualization module.

This class is only required when you would like to change the visual defaults of the plots and the figure.

The VisVTK module has the following configuration variables:

- ctrlpts (bool): Control points polygon/grid visibility. Default: True
- evalpts (bool): Curve/surface points visibility. Default: True
- trims (bool): Trim curve visibility. Default: True
- trim_size (int): Size of the trim curves. Default: 4
- figure_size (list): Size of the figure in (x, y). Default: (800, 600)
- line_width (int): Thickness of the lines on the figure. Default: 1.0

keypress_callback (obj, ev)

VTK callback for keypress events.

Keypress events:

- e: exit the application
- p: pick object (hover the mouse and then press to pick)
- f: fly to point (click somewhere in the window and press to fly)
- r: reset the camera
- s and w: switch between solid and wireframe modes
- b: change background color
- m: change color of the picked object
- d: print debug information (of picked object, point, etc.)
- h: change object visibility
- n: reset object visibility
- arrow keys: pan the model

Please refer to vtkInteractorStyle class reference for more details.

Parameters
geomdl.visualization.VisVTK.VisCurve2D
alias of geomdl.visualization.VisVTK.VisCurve3D

class geomdl.visualization.VisVTK.VisCurve3D (config=<geomdl.visualization.VisVTK.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

VTK visualization module for curves.

add (ptsarr, plot_type, name=", color=", idx=0)
    Adds points sets to the visualization instance for plotting.

Parameters

- ptsarr (list, tuple) – control or evaluated points
- plot_type (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- name (str) – name of the plot displayed on the legend
- color (int) – plot color
- color – plot index

animate (**kwargs)
    Generates animated plots (if supported).

    If the implemented visualization module supports animations, this function will create an animated figure.
    Otherwise, it will call render() method by default.

clear ()
    Clears the points, colors and names lists.

ctrlpts_offset
    Defines an offset value for the control points grid plots

    Only makes sense to use with surfaces with dense control points grid.

    Getter  Gets the offset value
    Setter   Sets the offset value
    Type     float

render (**kwargs)
    Plots the curve and the control points polygon.

size (plot_type)
    Returns the number of plots defined by the plot type.

    Parameters plot_type (str) – plot type
    Returns number of plots defined by the plot type
    Return type  int

vconf
    User configuration class for visualization

    Getter  Gets the user configuration class
    Type    vis.VisConfigAbstract
class geomdl.visualization.VisVTK.VisSurface(config=<geomdl.visualization.VisVTK.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

VTK visualization module for surfaces.

add(ptsarr, plot_type, name='', color='', idx=0)

Adds points sets to the visualization instance for plotting.

Parameters

- ptsarr (list, tuple) – control or evaluated points
- plot_type (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- name (str) – name of the plot displayed on the legend
- color (int) – plot color
- color – plot index

animate(**kwargs)

Generates animated plots (if supported).

If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call render() method by default.

clear()

Clears the points, colors and names lists.

ctrlpts_offset

Defines an offset value for the control points grid plots

Only makes sense to use with surfaces with dense control points grid.

Getter Gets the offset value

Setter Sets the offset value

Type float

render(**kwargs)

Plots the surface and the control points grid.

size(plot_type)

Returns the number of plots defined by the plot type.

Parameters plot_type (str) – plot type

Returns number of plots defined by the plot type

Return type int

vconf

User configuration class for visualization

Getter Gets the user configuration class

Type vis.VisConfigAbstract

class geomdl.visualization.VisVTK.VisVolume(config=<geomdl.visualization.VisVTK.VisConfig object>, **kwargs)

Bases: geomdl.vis.VisAbstract

VTK visualization module for volumes.

add(ptsarr, plot_type, name='', color='', idx=0)

Adds points sets to the visualization instance for plotting.
Parameters

- **ptsarr** (list, tuple) – control or evaluated points
- **plot_type** (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
- **name** (str) – name of the plot displayed on the legend
- **color** (int) – plot color
- **color** – plot index

**animate(** **kwargs**)**
Generates animated plots (if supported).
If the implemented visualization module supports animations, this function will create an animated figure.
Otherwise, it will call **render()** method by default.

**clear()**
Clears the points, colors and names lists.

**ctrlpts_offset**
Defines an offset value for the control points grid plots
Only makes sense to use with surfaces with dense control points grid.
- **Getter** Gets the offset value
- **Setter** Sets the offset value
- **Type** float

**render(** **kwargs**)**
Plots the volume and the control points.

**size(plot_type)**
Returns the number of plots defined by the plot type.
- **Parameters** **plot_type** (str) – plot type
- **Returns** number of plots defined by the plot type
- **Return type** int

**vconf**
User configuration class for visualization
- **Getter** Gets the user configuration class
- **Type** vis.VisConfigAbstract

**class geomdl.visualization.VisVTK.VisVoxel(config=<geomdl.visualization.VisVTK.VisConfig object>, **kwargs)**
Bases: geomdl.vis.VisAbstract
VTK visualization module for voxel representation of the volumes.

**add(ptsarr, plot_type, name="", color="", idx=0)**
Adds points sets to the visualization instance for plotting.
- **Parameters**
  - **ptsarr** (list, tuple) – control or evaluated points
  - **plot_type** (str) – type of the plot, e.g. ctrlpts, evalpts, bbox, etc.
  - **name** (str) – name of the plot displayed on the legend
  - **color** (int) – plot color
• **color** – plot index

**animate(**kwargs)**
Generates animated plots (if supported).
If the implemented visualization module supports animations, this function will create an animated figure. Otherwise, it will call `render()` method by default.

**clear()**
Clears the points, colors and names lists.

**ctrlpts_offset**
Defines an offset value for the control points grid plots
Only makes sense to use with surfaces with dense control points grid.
- **Getter** Gets the offset value
- **Setter** Sets the offset value
- **Type** float

**render(**kwargs)**
Plots the volume and the control points.

**size(plot_type)**
Returns the number of plots defined by the plot type.
- **Parameters** **plot_type**(str) – plot type
- **Returns** number of plots defined by the plot type
- **Return type** int

**vconf**
User configuration class for visualization
- **Getter** Gets the user configuration class
- **Type** vis.VisConfigAbstract

geomdl.visualization.VisVTK.random() → x in the interval [0, 1).
The users are not limited with these visualization backends. For instance, control points and evaluated points can be in various formats. Please refer to the *Exchange module documentation* for details.
You can use NURBS-Python (geomdl) with the command-line application `geomdl-cli`. The command-line application is designed for automation and input files are highly customizable using Jinja2 templates.

`geomdl-cli` is highly extensible via via the configuration file. It is very easy to generate custom commands as well as variables to change behavior of the existing commands or independently use for the custom commands. Since it runs inside the user’s Python environment, it is possible to create commands that use the existing Python libraries and even integrate NURBS-Python (geomdl) with these libraries.

### 18.1 Installation

The easiest method to install is via `pip`. It will install all the required modules.

```bash
$ pip install --user geomdl-cli
```

Please refer to `geomdl-cli` documentation for more installation options.

### 18.2 Documentation

`geomdl-cli` has a very detailed online documentation which describes the usage and customization options of the command-line application.

### 18.3 References

- PyPI: https://pypi.org/project/geomdl-cli
- Documentation: https://geomdl-cli.readthedocs.io
- Development: https://github.com/orbingol/geomdl-cli
The shapes module provides simple functions to generate commonly used analytic and spline geometries using NURBS-Python (geomdl).

Prior to NURBS-Python (geomdl) v5.0.0, the shapes module was automatically installed with the main package. Currently, it is maintained as a separate package.

### 19.1 Installation

The easiest method to install is via pip.

```
$ pip install --user geomdl.shapes
```

Please refer to geomdl-shapes documentation for more installation options.

### 19.2 Documentation

You can find the class and function references in the geomdl-shapes documentation.

### 19.3 References

- **PyPI**: [https://pypi.org/project/geomdl.shapes](https://pypi.org/project/geomdl.shapes)
- **Documentation**: [https://geomdl-shapes.readthedocs.io](https://geomdl-shapes.readthedocs.io)
- **Development**: [https://github.com/orbingol/geomdl-shapes](https://github.com/orbingol/geomdl-shapes)
The **Rhino importer/exporter**, rw3dm uses OpenNURBS to read and write .3dm files.

rw3dm comes with the following list of programs:

- `on2json` converts OpenNURBS .3dm files to geomdl JSON format
- `json2on` converts geomdl JSON format to OpenNURBS .3dm files

### 20.1 Use Cases

- Import geometry data from .3dm files and use it with `exchange.import_json()`
- Export geometry data with `exchange.export_json()` and convert to a .3dm file
- Convert OpenNURBS file format to OBJ, STL, OFF and other formats supported by geomdl

### 20.2 Installation

Please refer to the rw3dm repository for installation options. The binary files can be downloaded under Releases section of the GitHub repository.

### 20.3 Using with geomdl

The following code snippet illustrates importing the surface data converted from .3dm file:

```python
from geomdl import exchange
from geomdl import multi
from geomdl.visualization import VisMPL as vis
```
# Import converted data

```python
data = exchange.import_json("converted_rhino.json")
```

# Add the imported data to a surface container

```python
surf_cont = multi.SurfaceContainer(data)
surf_cont.sample_size = 30
```

# Visualize

```python
surf_cont.vis = vis.VisSurface(ctrlpts=False, trims=False)
surf_cont.render()
```

## 20.4 References

- **Development**: [https://github.com/orbingol/rw3dm](https://github.com/orbingol/rw3dm)
- **Downloads**: [https://github.com/orbingol/rw3dm/releases](https://github.com/orbingol/rw3dm/releases)
The **ACIS importer**, rwsat uses 3D ACIS Modeler to convert .sat files to geomdl JSON format. rwsat comes with the following list of programs:

- **sat2json** converts ACIS .sat files to geomdl JSON format
- **satgen** generates sample geometries

### 21.1 Use Cases

- Import geometry data from .sat files and use it with `exchange.import_json()`
- Convert ACIS file format to OBJ, STL, OFF and other formats supported by geomdl

### 21.2 Installation

Please refer to the rwsat repository for installation options. Due to ACIS licensing, no binary files are distributed within the repository.

### 21.3 Using with geomdl

The following code snippet illustrates importing the surface data converted from .sat file:

```python
from geomdl import exchange
def geomdl import multi
from geomdl.visualization import VisMPL as vis
# Import converted data
data = exchange.import_json("converted_acis.json")
```

(continues on next page)
```python
# Add the imported data to a surface container
surf_cont = multi.SurfaceContainer(data)
surf_cont.sample_size = 30

# Visualize
surf_cont.vis = vis.VisSurface(ctrlpts=False, trims=False)
surf_cont.render()
```

## 21.4 References

- **Development**: [https://github.com/orbingol/rwsat](https://github.com/orbingol/rwsat)
- **Documentation**: [https://github.com/orbingol/rwsat](https://github.com/orbingol/rwsat)
Python Module Index

c
compatibility (Unix, Windows), 162
construct (Unix, Windows), 166
control_points (Unix, Windows), 185
convert (Unix, Windows), 165
e
elements (Unix, Windows), 255
exchange (Unix, Windows), 178
exchange_vtk (Unix, Windows), 184
g
geomdl.compatibility, 162
geomdl.construct, 166
geomdl.control_points, 185
geomdl.convert, 165
geomdl.elements, 255
geomdl.exchange, 178
geomdl.exchange_vtk, 184
geomdl.fitting, 168
geomdl.helpers, 239
geomdl.knotvector, 184
geomdl.linalg, 247
geomdl.operations, 156
geomdl.ray, 263
geomdl.sweeping, 177
geomdl.trimming, 176
geomdl.utilities, 238
geomdl.vis.VisAbstract, 265
geomdl.vis.VisConfigAbstract, 266
geomdl.visualization.VisMPL, 266
geomdl.visualization.VisVTK, 275
geomdl.voxelize, 254
k
knotvector (Unix, Windows), 184
l
linalg (Unix, Windows), 247
o
operations (Unix, Windows), 156
r
ray (Unix, Windows), 263
s
sweeping (Unix, Windows), 177
t
trimming (Unix, Windows), 176
u
utilities (Unix, Windows), 238
v
VisMPL (Unix, Windows), 266
VisVTK (Unix, Windows), 275
voxelize (Unix, Windows), 254
h
helpers (Unix, Windows), 239
i
interpolate (Unix, Windows), 168
Index

A

AbstractContainer (class in geomdl.multi), 136
AbstractEvaluator (class in geomdl.evaluators), 231
AbstractManager (class in geomdl.control_points), 185
AbstractTessellate (class in geomdl.tessellate), 170
add() (geomdl.multi.AbstractContainer method), 137
add() (geomdl.multi.CurveContainer method), 140
add() (geomdl.multi.SurfaceContainer method), 145
add() (geomdl.multi.VolumeContainer method), 151
add() (geomdl.visualization.VisMPL.VisCurve2D method), 268
add() (geomdl.visualization.VisMPL.VisCurve3D method), 269
add() (geomdl.visualization.VisMPL.VisSurface method), 271
add() (geomdl.visualization.VisMPL.VisSurfScatter method), 269
add() (geomdl.visualization.VisMPL.VisSurfWireframe method), 270
add() (geomdl.visualization.VisMPL.VisVolume method), 273
add() (geomdl.visualization.VisMPL.VisVoxel method), 273
add() (geomdl.visualization.VisVTK.VisCurve3D method), 276
add() (geomdl.visualization.VisVTK.VisSurface method), 277
add() (geomdl.visualization.VisVTK.VisVolume method), 277
add() (geomdl.visualization.VisVTK.VisVoxel method), 278
add_dimension() (in module geomdl.operations), 158
add_face() (geomdl.elements.Body method), 261
add_triangle() (geomdl.elements.Face method), 260
add_trim() (geomdl.abstract.Surface method), 202
add_trim() (geomdl.abstract.Volume method), 212
add_trim() (geomdl.BSpline.Surface method), 74
add_trim() (geomdl.BSpline.Volume method), 87
add_trim() (geomdl.NURBS.Surface method), 109
add_trim() (geomdl.NURBS.Volume method), 123
add_vertex() (geomdl.elements.Quad method), 259
add_vertex() (geomdl.elements.Triangle method), 257
animate() (geomdl.visualization.VisMPL.VisCurve2D method), 268
animate() (geomdl.visualization.VisMPL.VisCurve3D method), 269
animate() (geomdl.visualization.VisMPL.VisSurface method), 272
animate() (geomdl.visualization.VisMPL.VisSurfScatter method), 270
animate() (geomdl.visualization.VisMPL.VisSurfWireframe method), 271
animate() (geomdl.visualization.VisMPL.VisVolume method), 273
animate() (geomdl.visualization.VisMPL.VisVoxel method), 274
animate() (geomdl.visualization.VisVTK.VisCurve3D method), 276
animate() (geomdl.visualization.VisVTK.VisSurface method), 277
animate() (geomdl.visualization.VisVTK.VisVolume method), 278
animate() (geomdl.visualization.VisVTK.VisVoxel method), 279
append() (geomdl.multi.AbstractContainer method), 137
append() (geomdl.multi.CurveContainer method), 141
append() (geomdl.multi.SurfaceContainer method), 145
append() (geomdl.multi.VolumeContainer method), 151
approximate_curve() (in module geomdl.fitting), 169

291
approximate_surface() (in module geomdl.fitting), 169
arguments (geomdl.tessellate.AbstractTessellate attribute), 170
arguments (geomdl.tessellate.QuadTessellate attribute), 173
arguments (geomdl.tessellate.TriangularTessellate attribute), 171
arguments (geomdl.tessellate.TrimTessellate attribute), 172

backward_substitution() (in module geomdl.linalg), 247
basis_function() (in module geomdl.helpers), 239
basis_function_all() (in module geomdl.helpers), 240
basis_function_ders() (in module geomdl.helpers), 240
basis_function_ders_one() (in module geomdl.helpers), 240
basis_function_one() (in module geomdl.helpers), 241
basis_functions() (in module geomdl.helpers), 241
basis_functions_ders() (in module geomdl.helpers), 241
bbox (geomdl.abstract.Curve attribute), 195
bbox (geomdl.abstract.SplineGeometry attribute), 226
bbox (geomdl.abstract.Surface attribute), 202
bbox (geomdl.abstract.Volume attribute), 212
bbox (geomdl.BSpline.Curve attribute), 65
bbox (geomdl.BSpline.Surface attribute), 74
bbox (geomdl.BSpline.Volume attribute), 88
bbox (geomdl.multi.AbstractContainer attribute), 137
bbox (geomdl.multi.CurveContainer attribute), 141
bbox (geomdl.multi.SurfaceContainer attribute), 145
bbox (geomdl.multi.VolumeContainer attribute), 151
bbox (geomdl.NURBS.Curve attribute), 99
bbox (geomdl.NURBS.Surface attribute), 109
bbox (geomdl.NURBS.Volume attribute), 123
binomial_coefficient (in module geomdl.linalg), 247
binormal() (geomdl.BSpline.Curve method), 65
binormal() (geomdl.NURBS.Curve method), 99
binormal() (in module geomdl.operations), 161
Body (class in geomdl.elements), 261
bspline_to_nurbs() (in module geomdl.convert), 165
bumps() (geomdl.CPGen.Grid method), 192
bumps() (geomdl.CPGen.GridWeighted method), 193

c
check() (in module geomdl.knotvector), 185
check_params() (in module geomdl.utilities), 238
clear() (geomdl.visualization.VisMPL.VisCurve2D method), 268
clear() (geomdl.visualization.VisMPL.VisCurve3D method), 269
clear() (geomdl.visualization.VisMPL.VisSurface method), 272
clear() (geomdl.visualization.VisMPL.VisSurfScatter method), 270
clear() (geomdl.visualization.VisMPL.VisSurfWireframe method), 271
clear() (geomdl.visualization.VisMPL.VisVolume method), 273
clear() (geomdl.visualization.VisVTK.VisSurf method), 274
clear() (geomdl.visualization.VisVTK.VisSurface method), 277
clear() (geomdl.visualization.VisVTK.VisVolume method), 278
clear() (geomdl.visualization.VisVTK.VisVoxel method), 279
color_generator() (in module geomdl.utilities), 238
combine_ctrlpts_weights() (in module geomdl.compatibility), 162
compatibility (module), 162
construct (module), 166
construct_surface() (in module geomdl.construct), 166
construct_volume() (in module geomdl.construct), 166
crntrol_points (module), 185
convert (module), 165
convex_hull() (in module geomdl.linalg), 247
cpsize (geomdl.abstract.Curve attribute), 195
cpsize (geomdl.abstract.SplineGeometry attribute), 226
cpsize (geomdl.abstract.Surface attribute), 202
cpsize (geomdl.abstract.Volume attribute), 212
cpsize (geomdl.BSpline.Curve attribute), 65
cpsize (geomdl.BSpline.Surface attribute), 75
cpsize (geomdl.BSpline.Volume attribute), 88
cpsize (geomdl.NURBS.Curve attribute), 100
cpsize (geomdl.NURBS.Surface attribute), 110
cpsize (geomdl.NURBS.Volume attribute), 123
ctrllpts (geomdl.abstract.Curve attribute), 195
ctrllpts (geomdl.abstract.SplineGeometry attribute), 226
ctrllpts (geomdl.abstract.Surface attribute), 202
ctrllpts (geomdl.abstract.Volume attribute), 213
ctrllpts (geomdl.BSpline.Curve attribute), 65
ctrllpts (geomdl.BSpline.Surface attribute), 75
Index
data (geomdl.abstract.Curve attribute), 195
data (geomdl.abstract.Surface attribute), 203
data (geomdl.abstract.Volume attribute), 213
data (geomdl.BSpline.Curve attribute), 66
data (geomdl.BSpline.Surface attribute), 76
data (geomdl.BSpline.Volume attribute), 89
data (geomdl.elements.Quad attribute), 259
data (geomdl.elements.Triangle attribute), 257
data (geomdl.elements.Vertex attribute), 255
data (geomdl.freeform.Vertex attribute), 133
data (geomdl.multi.AbstractContainer attribute), 137
data (geomdl.multi.CurveContainer attribute), 141
data (geomdl.multi.SurfaceContainer attribute), 145
data (geomdl.multi.VolumeContainer attribute), 151
data (geomdl.NURBS.Curve attribute), 100
data (geomdl.NURBS.Surface attribute), 111
data (geomdl.NURBS.Volume attribute), 124
decompose_curve() (in module geomdl.operations), 158
decompose_surface() (in module geomdl.operations), 160
degree (geomdl.abstract.Curve attribute), 196
degree (geomdl.abstract.SplineGeometry attribute), 226
degree (geomdl.abstract.Surface attribute), 203
degree (geomdl.abstract.Volume attribute), 213
degree (geomdl.BSpline.Curve attribute), 66
degree (geomdl.BSpline.Surface attribute), 76
degree (geomdl.BSpline.Volume attribute), 89
degree (geomdl.NURBS.Curve attribute), 101
degree (geomdl.NURBS.Surface attribute), 111
degree (geomdl.NURBS.Volume attribute), 124
degree_elevation() (in module geomdl.helpers), 242
degree_reduction() (in module geomdl.helpers), 242
degree_u (geomdl.abstract.Surface attribute), 203
degree_u (geomdl.abstract.Volume attribute), 213
degree_u (geomdl.BSpline.Surface attribute), 77
degree_u (geomdl.NURBS.Surface attribute), 112
degree_u (geomdl.NURBS.Volume attribute), 124
degree_v (geomdl.abstract.Surface attribute), 203
degree_v (geomdl.abstract.Volume attribute), 214
degree_v (geomdl.BSpline.Surface attribute), 77
degree_v (geomdl.BSpline.Volume attribute), 89
degree_v (geomdl.NURBS.Surface attribute), 112
degree_v (geomdl.NURBS.Volume attribute), 124
degree_w (geomdl.abstract.Volume attribute), 214
degree_w (geomdl.BSpline.Volume attribute), 89
degree_w (geomdl.NURBS.Volume attribute), 124
delta (geomdl.abstract.Curve attribute), 196
delta (geomdl.abstract.Surface attribute), 203
delta (geomdl.abstract.Volume attribute), 214
delta (geomdl.BSpline.Curve attribute), 66
delta (geomdl.BSpline.Surface attribute), 77
delta (geomdl.BSpline.Volume attribute), 89
delta (geomdl.multi.AbstractContainer attribute), 137
delta (geomdl.multi.CurveContainer attribute), 145
delta (geomdl.multi.SurfaceContainer attribute), 151
delta (geomdl.NURBS.Curve attribute), 101
delta (geomdl.NURBS.Surface attribute), 112
delta (geomdl.NURBS.Volume attribute), 125
delta_u (geomdl.abstract.Surface attribute), 204
delta_u (geomdl.abstract.Volume attribute), 214
delta_u (geomdl.BSpline.Surface attribute), 77
delta_u (geomdl.BSpline.Volume attribute), 90
delta_u (geomdl.multi.SurfaceContainer attribute), 145
delta_u (geomdl.multi.VolumeContainer attribute), 151
delta_v (geomdl.abstract.Surface attribute), 204
delta_v (geomdl.abstract.Volume attribute), 214
delta_v (geomdl.BSpline.Surface attribute), 77
delta_v (geomdl.BSpline.Volume attribute), 90
delta_v (geomdl.multi.SurfaceContainer attribute), 146
delta_v (geomdl.multi.VolumeContainer attribute), 152
delta_v (geomdl.NURBS.Surface attribute), 113
delta_v (geomdl.NURBS.Volume attribute), 125
delta_w (geomdl.abstract.Volume attribute), 215
delta_w (geomdl.BSpline.Volume attribute), 90
delta_w (geomdl.multi.VolumeContainer attribute), 152
delta_w (geomdl.NURBS.Volume attribute), 126
derivative_curve() (in module geomdl.operations), 158
derivative_surface() (in module geomdl.operations), 160
derivatives() (geomdl.abstract.Curve method), 196
derivatives() (geomdl.abstract.Surface method), 204
derivatives() (geomdl.BSpline.Curve method), 66
derivatives() (geomdl.BSpline.Surface method), 78
derivatives() (geomdl.evaluators.AbstractEvaluator method), 231
derivatives() (geomdl.evaluators.CurveEvaluator method), 232
derivatives() (geomdl.evaluators.CurveEvaluator2 method), 233
derivatives() (geomdl.evaluators.VolumeEvaluator method), 235

d (geomdl.ray.Ray attribute), 263

delta (geomdl.BSpline.Curve attribute), 66
delta (geomdl.BSpline.Surface attribute), 77
evaluate() (geomdl.NURBS.Volume method), 126
evaluate_bounding_box() (in module geomdl.utilities), 238
evaluate_list() (geomdl.abstract.Curve method), 197
evaluate_list() (geomdl.abstract.Surface method), 205
evaluate_list() (geomdl.abstract.Volume method), 216
evaluate_list() (geomdl.BSpline.Curve method), 67
evaluate_list() (geomdl.BSpline.Surface method), 79
evaluate_list() (geomdl.BSpline.Volume method), 91
evaluate_list() (geomdl.NURBS.Curve method), 102
evaluate_list() (geomdl.NURBS.Surface method), 114
evaluate_list() (geomdl.NURBS.Volume method), 126
evaluate_single() (geomdl.abstract.Curve method), 197
evaluate_single() (geomdl.abstract.Curve method), 205
evaluate_single() (geomdl.abstract.Volume method), 216
evaluate_single() (geomdl.BSpline.Curve method), 67
evaluate_single() (geomdl.BSpline.Surface method), 79
evaluate_single() (geomdl.BSpline.Volume method), 91
evaluate_single() (geomdl.NURBS.Curve method), 102
evaluate_single() (geomdl.NURBS.Surface method), 114
evaluate_single() (geomdl.NURBS.Volume method), 127
evaluator (geomdl.abstract.Curve attribute), 197
evaluator (geomdl.abstract.CurveGeometry attribute), 227
evaluator (geomdl.abstract.Surface attribute), 205
evaluator (geomdl.abstract.Volume attribute), 216
evaluator (geomdl.BSpline.Curve attribute), 68
evaluator (geomdl.BSpline.Surface attribute), 79
evaluator (geomdl.BSpline.Volume attribute), 91
evaluator (geomdl.NURBS.Curve attribute), 102
evaluator (geomdl.NURBS.Surface attribute), 114
evaluator (geomdl.NURBS.Volume attribute), 127
exchange (module), 178
exchange_vtk (module), 184
export_3dm() (in module geomdl.exchange), 183
export_cfg() (in module geomdl.exchange), 180
export_csv() (in module geomdl.exchange), 179
export_json() (in module geomdl.exchange), 181
export_obj() (in module geomdl.exchange), 181
export_off() (in module geomdl.exchange), 182
export_polydata() (in module geomdl.exchange_vtk), 184
export_smesh() (in module geomdl.exchange), 183
export_stl() (in module geomdl.exchange), 182
export_txt() (in module geomdl.exchange), 179
export_vmesh() (in module geomdl.exchange), 183
export_yaml() (in module geomdl.exchange), 180
extract_curves() (in module geomdl.construct), 167
extract_isosurface() (in module geomdl.construct), 167
extract_surfaces() (in module geomdl.construct), 167
F
Face (class in geomdl.elements), 260
faces (geomdl.abstract.Surface attribute), 206
faces (geomdl.BSpline.Surface attribute), 79
faces (geomdl.elements.Body attribute), 261
faces (geomdl.multi.SurfaceContainer attribute), 146
faces (geomdl.NURBS.Surface attribute), 114
faces (geomdl.tessellate.AbstractTessellate attribute), 170
faces (geomdl.tessellate.QuadTessellate attribute), 173
faces (geomdl.tessellate.TriangularTessellate attribute), 171
faces (geomdl.tessellate.TrimTessellate attribute), 172
find_ctrlpts() (in module geomdl.operations), 160
find_index() (geomdl.control_points.AbstractManager method), 186
find_index() (geomdl.control_points.CurveManager method), 187
find_index() (geomdl.control_points.SurfaceManager method), 189
find_index() (geomdl.control_points.VolumeManager method), 191
find_multiplicity() (in module geomdl.helpers), 242
find_span_binsearch() (in module geomdl.helpers), 243
find_span_linear() (in module geomdl.helpers), 243
find_spans() (in module geomdl.helpers), 243
fix_multi_trim_curves() (in module geomdl.trimming), 176
fix_trim_curves() (in module geomdl.trimming), 176
flip() (in module geomdl.operations), 162
flip_ctrlpts() (in module geomdl.compatibility), 163
Index

NURBS-Python Documentation, Release 5.2.9

geomdl.NURBS.Volume attribute, 103
gemdl.NURBS.Surface attribute, 115
gemdl.NURBS.Volume attribute, 127
import_3dm() (in module geomdl.exchange), 183
import cfg() (in module geomdl.exchange), 179
import csv() (in module geomdl.exchange), 275
import json() (in module geomdl.exchange), 181
import obj() (in module geomdl.exchange), 183
import smesh() (in module geomdl.exchange), 182
import txt() (in module geomdl.exchange), 178
import vmesh() (in module geomdl.exchange), 180
import yml() (in module geomdl.exchange), 180

G

generate() (geomdl.CPGen.Grid method), 92
generate() (in module geomdl.CPGen.GridWeighted method), 193
generate() (in module geomdl.knotvector), 184
generate_ctrlpts2d_weights() (in module geomdl.compatibility), 163
generate_ctrlpts2d_weights_file() (in module geomdl.compatibility), 164
generate_ctrlpts_weights() (in module geomdl.compatibility), 164
generate_ctrlpts() (in module geomdl.compatibility), 164
generate() (in module geomdl.CPGen.GridWeighted method), 193
generate() (in module geomdl.knotvector), 184
generate_ctrlpts2d_weights() (in module geomdl.compatibility), 163
generate_ctrlpts2d_weights_file() (in module geomdl.compatibility), 164
generate_ctrlpts_weights() (in module geomdl.compatibility), 164
generate_ctrlpts() (in module geomdl.compatibility), 164
generate() (in module geomdl.CPGen.GridWeighted method), 193
generate() (in module geomdl.knotvector), 184
generate_ctrlpts2d_weights() (in module geomdl.compatibility), 163
generate_ctrlpts2d_weights_file() (in module geomdl.compatibility), 164
generate_ctrlpts_weights() (in module geomdl.compatibility), 164
generate_ctrlpts() (in module geomdl.compatibility), 164

H

Helpers (module)

I

id (geomdl.abstract.Curve attribute), 197
id (geomdl.abstract.GeomdlBase attribute), 222
id (geomdl.abstract.Geometry attribute), 224
id (geomdl.abstract.SplineGeometry attribute), 227
id (geomdl.abstract.Surface attribute), 206
id (geomdl.abstract.Volume attribute), 216
id (geomdl.BSpline.Curve attribute), 68
id (geomdl.BSpline.Surface attribute), 80
id (geomdl.BSpline.Volume attribute), 92
id (geomdl.elements.Body attribute), 262
id (geomdl.elements.Face attribute), 260
id (geomdl.elements.Quad attribute), 259
id (geomdl.elements.Triangle attribute), 257
id (geomdl.elements.Vertex attribute), 255
id (geomdl.freeform.Freeform attribute), 134
id (geomdl.multi.AbstractContainer attribute), 138
id (geomdl.multi.CurveContainer attribute), 142
id (geomdl.multi.SurfaceContainer attribute), 146
id (geomdl.multi.VolumeContainer attribute), 153
id (geomdl.NURBS.Curve attribute), 103
id (geomdl.NURBS.Surface attribute), 115
id (geomdl.NURBS.Volume attribute), 127

GeomdlBase (class in geomdl.abstract), 222
Geometry (class in geomdl.abstract), 223
get_ctrlpt() (geomdl.control_points.AbstractManager method), 186
get_ctrlpt() (geomdl.control_points.CurveManager method), 187
get_ctrlpt() (geomdl.control_points.SurfaceManager method), 189
get_ctrlpt() (geomdl.control_points.VolumeManager method), 191
get_ptdata() (geomdl.control_points.AbstractManager method), 186
get_ptdata() (geomdl.control_points.CurveManager method), 187
get_ptdata() (geomdl.control_points.SurfaceManager method), 189
get_ptdata() (geomdl.control_points.VolumeManager method), 191

GeomdlBase (class in geomdl.CPGen), 191
grid (geomdl.CPGen.Grid attribute), 192
grid_weighted (class in geomdl.CPGen), 193

Index 297
insert_knot() (geomdl.BSpline.Curve method), 68
insert_knot() (geomdl.BSpline.Surface method), 80
insert_knot() (geomdl.BSpline.Volume method), 92
insert_knot() (geomdl.NURBS.Curve method), 103
insert_knot() (geomdl.NURBS.Surface method), 115
insert_knot() (geomdl.NURBS.Volume method), 127
insert_knot() (in module geomdl.operations), 156
inside (geomdl.elements.Triangle attribute), 258
inside (geomdl.elements.Vertex attribute), 255
interpolate (module), 168
interpolate_curve() (in module geomdl.fitting), 168
interpolate_surface() (in module geomdl.fitting), 168
intersect() (in module geomdl.ray), 263
is_left() (in module geomdl.linalg), 248
is_tessellated() (geomdl.tessellate.AbstractTessellate method), 170
is_tessellated() (geomdl.tessellate.QuadTessellate method), 173
is_tessellated() (geomdl.tessellate.TriangularTessellate method), 171
is_tessellated() (geomdl.tessellate.TrimTessellate method), 172

K

keypress_callback() (geomdl.visualization.VisVTK.VisConfig method), 275
knot_insertion() (in module geomdl.helpers), 244
knot_insertion_alpha (in module geomdl.helpers), 244
knot_insertion_kv() (in module geomdl.helpers), 244
knot_refinement() (in module geomdl.helpers), 245
knot_removal() (in module geomdl.helpers), 245
knot_removal_alpha_i (in module geomdl.helpers), 246
knot_removal_alpha_j (in module geomdl.helpers), 246
knot_removal_kv() (in module geomdl.helpers), 246
knotvector (geomdl.abstract.Curve attribute), 197
knotvector (geomdl.abstract.SplineGeometry attribute), 227
knotvector (geomdl.abstract.Surface attribute), 206
knotvector (geomdl.abstract.Volume attribute), 216
knotvector (geomdl.BSpline.Curve attribute), 68
knotvector (geomdl.BSpline.Volume attribute), 80
knotvector (geomdl.BSpline.Volume attribute), 92
knotvector (geomdl.NURBS.Curve attribute), 103
knotvector (geomdl.NURBS.Surface attribute), 115
knotvector (geomdl.NURBS.Volume attribute), 127
knotvector (module), 184
knotvector_u (geomdl.abstract.Surface attribute), 206
knotvector_u (geomdl.abstract.Volume attribute), 216
knotvector_u (geomdl.BSpline.Surface attribute), 80
knotvector_u (geomdl.BSpline.Volume attribute), 92
knotvector_u (geomdl.NURBS.Surface attribute), 115
knotvector_v (geomdl.NURBS.Volume attribute), 127
knotvector_v (geomdl.abstract.Surface attribute), 206
knotvector_v (geomdl.abstract.Volume attribute), 217
knotvector_v (geomdl.BSpline.Surface attribute), 80
knotvector_v (geomdl.BSpline.Volume attribute), 92
knotvector_v (geomdl.NURBS.Surface attribute), 115
knotvector_v (geomdl.NURBS.Volume attribute), 128
knotvector_w (geomdl.abstract.Volume attribute), 217
knotvector_w (geomdl.BSpline.Volume attribute), 93
knotvector_w (geomdl.NURBS.Volume attribute), 128

L

length_curve() (in module geomdl.operations), 159
linalg (module), 247
linspace() (in module geomdl.linalg), 248
load() (geomdl.BSpline.Curve method), 68
load() (geomdl.BSpline.Surface method), 80
load() (geomdl.BSpline.Volume method), 93
load() (geomdl.NURBS.Curve method), 103
load() (geomdl.NURBS.Surface method), 116
load() (geomdl.NURBS.Volume method), 128
lu_decomposition() (in module geomdl.linalg), 249
lu_factor() (in module geomdl.linalg), 249
lu_solve() (in module geomdl.linalg), 249

M

make_quad() (in module geomdl.utilities), 238
make_quad_mesh() (in module geomdl.tessellate), 174
make_quadtree() (in module geomdl.tessellate), 238
make_triangle_mesh() (in module geomdl.tessellate), 173
make_zigzag() (in module geomdl.utilities), 239
map_trim_to_geometry() (in module geomdl.trimming), 176
matrix_determinant() (in module geomdl.linalg), 249
matrix_inverse() (in module geomdl.linalg), 249
matrix_multiply() (in module geomdl.linalg), 250
matrix_pivot() (in module geomdl.linalg), 250
matrix_scalar() (in module geomdl.linalg), 250
matrix_transpose() (in module geomdl.linalg), 250
mconf (in module geomdl.vis.VisAbstract), 265

N
name (geomdl.abstract.Curve attribute), 198
name (geomdl.abstract.GeomdlBase attribute), 222
name (geomdl.abstract.Geometry attribute), 224
name (geomdl.abstract.SplineGeometry attribute), 227
name (geomdl.abstract.Surface attribute), 206
name (geomdl.abstract.Volume attribute), 217
name (geomdl.BSpline.Curve attribute), 68
name (geomdl.BSpline.Surface attribute), 81
name (geomdl.BSpline.Volume attribute), 93
name (geomdl.elements.Body attribute), 262
name (geomdl.elements.Face attribute), 260
name (geomdl.elements.Quad attribute), 259
name (geomdl.elements.Triangle attribute), 258
name (geomdl.elements.Vertex attribute), 255
name (geomdl.evaluators.AbstractEvaluator attribute), 231
name (geomdl.evaluators.CurveEvaluator attribute), 232
name (geomdl.evaluators.CurveEvaluator2 attribute), 234
name (geomdl.evaluators.CurveEvaluatorRational attribute), 233
name (geomdl.evaluators.SurfaceEvaluator attribute), 235
name (geomdl.evaluators.SurfaceEvaluator2 attribute), 236
name (geomdl.evaluators.SurfaceEvaluatorRational attribute), 235
name (geomdl.evaluators.VolumeEvaluator attribute), 237
name (geomdl.evaluators.VolumeEvaluatorRational attribute), 237
name (geomdl.freeform.Freeform attribute), 134
name (geomdl.multi.AbstractContainer attribute), 138
name (geomdl.multi.CurveContainer attribute), 142
name (geomdl.multi.SurfaceContainer attribute), 147
name (geomdl.multi.VolumeContainer attribute), 153
name (geomdl.NURBS.Curve attribute), 103
name (geomdl.NURBS.Surface attribute), 116
name (geomdl.NURBS.Volume attribute), 128

normal() (geomdl.BSpline.Curve method), 69
normal() (geomdl.BSpline.Surface method), 81
normal() (geomdl.NURBS.Curve method), 103
normal() (geomdl.NURBS.Surface method), 116
normal() (in module geomdl.operations), 161
normalize() (in module geomdl.knotvector), 185
nurbs_to_bspline() (in module geomdl.convert), 166

O
operations (module), 156
opt (geomdl.abstract.Curve attribute), 198
opt (geomdl.abstract.GeomdlBase attribute), 222
opt (geomdl.abstract.Geometry attribute), 224
opt (geomdl.abstract.SplineGeometry attribute), 228
opt (geomdl.abstract.Surface attribute), 207
opt (geomdl.abstract.Volume attribute), 217
opt (geomdl.BSpline.Curve attribute), 69
opt (geomdl.BSpline.Surface attribute), 81
opt (geomdl.BSpline.Volume attribute), 93
opt (geomdl.elements.Body attribute), 262
opt (geomdl.elements.Face attribute), 261
opt (geomdl.elements.Quad attribute), 259
opt (geomdl.elements.Triangle attribute), 258
opt (geomdl.elements.Vertex attribute), 256
opt (geomdl.freeform.Freeform attribute), 134
opt (geomdl.multi.AbstractContainer attribute), 138
opt (geomdl.multi.CurveContainer attribute), 142
opt (geomdl.multi.SurfaceContainer attribute), 147
opt (geomdl.multi.VolumeContainer attribute), 153
opt (geomdl.NURBS.Curve attribute), 104
opt (geomdl.NURBS.Surface attribute), 116
opt (geomdl.NURBS.Volume attribute), 128
opt_get() (geomdl.abstract.Curve method), 198
opt_get() (geomdl.abstract.GeomdlBase method), 223
opt_get() (geomdl.abstract.Geometry method), 225
opt_get() (geomdl.abstract.SplineGeometry method), 228
opt_get() (geomdl.abstract.Surface method), 207
opt_get() (geomdl.abstract.Volume method), 218
opt_get() (geomdl.BSpline.Curve method), 69
opt_get() (geomdl.BSpline.Surface method), 82
opt_get() (geomdl.BSpline.Volume method), 94
opt_get() (geomdl.elements.Body method), 262
opt_get() (geomdl.elements.Face method), 261
opt_get() (geomdl.elements.Quad method), 260
opt_get() (geomdl.elements.Triangle method), 258
opt_get() (geomdl.elements.Vertex method), 256
opt_get() (geomdl.freeform.Freeform method), 135
opt_get() (geomdl.multi.AbstractContainer method), 139
opt_get() (geomdl.multi.CurveContainer method), 142

Index

299
opt_get() (geometl.multi.SurfaceContainer method), 147
opt_get() (geometl.multi.VolumeContainer method), 153
opt_get() (geometl.NURBS.Curve method), 104
opt_get() (geometl.NURBS.Surface method), 117
opt_get() (geometl.NURBS.Volume method), 129
order (geometl.abstract.Curve attribute), 198
order (geometl.BSpline.Curve attribute), 70
order (geometl.NURBS.Curve attribute), 104
order_u (geometl.abstract.Surface attribute), 207
order_u (geometl.BSpline.Surface attribute), 82
order_u (geometl.NURBS.Surface attribute), 117
order_v (geometl.abstract.Volume attribute), 129
order_v (geometl.BSpline.Volume attribute), 94
order_v (geometl.NURBS.Volume attribute), 130
p (geometl-ray-Ray attribute), 263
pdimension (geometl.abstract.Curve attribute), 199
pdimension (geometl.abstract.SplineGeometry attribute), 228
pdimension (geometl.abstract.Surface attribute), 208
pdimension (geometl.BSpline.Curve attribute), 70
pdimension (geometl.BSpline.Surface attribute), 82
pdimension (geometl.BSpline.Volume attribute), 94
pdimension (geometl.multi.AbstractContainer attribute), 139
pdimension (geometl.multi.CurveContainer attribute), 143
pdimension (geometl.multi.SurfaceContainer attribute), 147
pdimension (geometl.multi.VolumeContainer attribute), 154
pdimension (geometl.NURBS.Curve attribute), 105
pdimension (geometl.NURBS.Surface attribute), 117
pdimension (geometl.NURBS.Volume attribute), 130
point_distance() (in module geometl.linalg), 251
point_mid() (in module geometl.linalg), 251
point_translate() (in module geometl.linalg), 251
points (geometl-ray-Ray attribute), 263
polygon_triangulate() (in module geometl.tessellate), 174
Q
Quad (class in geometl.elements), 259
QuadTessellate (class in geometl.tessellate), 173
R
random() (in module geometl.visualization.VisVTK), 279
range (geometl.abstract.Curve attribute), 199
range (geometl.abstract.SplineGeometry attribute), 228
range (geometl.abstract.Surface attribute), 208
range (geometl.abstract.Volume attribute), 218
range (geometl.BSpline.Curve attribute), 70
range (geometl.BSpline.Surface attribute), 82
range (geometl.BSpline.Volume attribute), 94
range (geometl.NURBS.Curve attribute), 105
range (geometl.NURBS.Surface attribute), 117
range (geometl.NURBS.Volume attribute), 130
rational (geometl.abstract.Curve attribute), 199
rational (geometl.abstract.SplineGeometry attribute), 228
rational (geometl.abstract.Surface attribute), 208
rational (geometl.abstract.Volume attribute), 219
rational (geometl.BSpline.Curve attribute), 70
rational (geometl.BSpline.Surface attribute), 82
rational (geometl.BSpline.Volume attribute), 94
rational (geometl.NURBS.Curve attribute), 105
rational (geometl.NURBS.Surface attribute), 117
rational (geometl.NURBS.Volume attribute), 130
Ray (class in geometl-ray), 263
RayIntersection (class in geometl-ray), 263
refine_knotvector() (in module geometl.operations), 157
remove_knot() (geometl.BSpline.Curve method), 70
remove_knot() (geometl.BSpline.Surface method), 82
remove_knot() (geometl.BSpline.Volume method), 95
remove_knot() (geometl.NURBS.Curve method), 117
remove_knot() (geometl.NURBS.Surface method), 130
remove_knot() (in module geometl.operations), 156
render() (geometl.abstract.Curve method), 199
render() (geometl.abstract.SplineGeometry method), 229
render() (geometl.abstract.Surface method), 208
render() (geometl.abstract.Volume method), 219
render() (geometl.BSpline.Curve method), 70
render() (geometl.BSpline.Surface method), 83
render() (geometl.BSpline.Volume method), 95
render() (geometl.multi.AbstractContainer method), 139
render() (geometl.multi.CurveContainer method), 143
Index
sample_size_v (geomdl.multi.SurfaceContainer attribute), 149
sample_size_v (geomdl.multi.VolumeContainer attribute), 155
sample_size_v (geomdl.NURBS.Surface attribute), 119
sample_size_v (geomdl.NURBS.Volume attribute), 132
sample_size_w (geomdl.abstract.Volume attribute), 220
sample_size_w (geomdl.BSpline.Volume attribute), 96
sample_size_w (geomdl.multi.VolumeContainer attribute), 155
sample_size_w (geomdl.NURBS.Volume attribute), 132
save() (geomdl.BSpline.Curve method), 72
save() (geomdl.BSpline.Surface method), 84
save() (geomdl.BSpline.Volume method), 97
save() (geomdl.NURBS.Curve method), 106
save() (geomdl.NURBS.Surface method), 119
save() (geomdl.NURBS.Volume method), 132
save_figure_as() (geomdl.visualization.VisMPL.VisConfig static method), 267
save_voxel_grid() (in module geomdl.voxelize), 255
scale() (in module geomdl.operations), 162
separate_ctrlpts_weights() (in module geomdl.compatibility), 165
set_axes_equal() (geomdl.visualization.VisMPL.VisCurve2D method), 268
set_axes_equal() (geomdl.visualization.VisMPL.VisCurve3D method), 269
set_axes_equal() (geomdl.visualization.VisMPL.VisVolume method), 273
set_axes_equal() (geomdl.visualization.VisVTK.VisVolume method), 279
set_ctrlpts() (geomdl.NURBS.Volume method), 132
set_ctrlpts() (geomdl.NURBS.Volume method, AbstractManager method), 186
set_ctrlpts() (geomdl.control_points.CurveManager method), 188
set_ctrlpts() (geomdl.control_points.SurfaceManager method), 189
set_ctrlpts() (geomdl.control_points.VolumeManager method), 191
ds() (geomdl.visualization.VisMPL.VisCurve2D method), 268
ds() (geomdl.visualization.VisMPL.VisCurve3D method), 269
ds() (geomdl.visualization.VisMPL.VisVolume method), 273
size() (geomdl.visualization.VisMPL.VisSurfScatter method), 270
size() (geomdl.visualization.VisMPL.VisSurfWireframe method), 271
size() (geomdl.visualization.VisMPL.VisVolume method), 273
size() (geomdl.visualization.VisVTK.VisVoxel method), 279
set_ctrlpts() (geomdl.control_points.AbstractManager method), 186
set_ctrlpts() (geomdl.control_points.CurveManager method), 188
set_ctrlpts() (geomdl.control_points.SurfaceManager method), 189
set_ctrlpts() (geomdl.control_points.VolumeManager method), 191
set_ctrlpts() (geomdl.abstract.Curve method), 200
set_ctrlpts() (geomdl.abstract.SplineGeometry method), 229
set_ctrlpts() (geomdl.abstract.Surface method), 210
set_ctrlpts() (geomdl.abstract.Volume method), 220
set_ctrlpts() (geomdl.BSpline.Curve method), 72
set_ctrlpts() (geomdl.BSpline.Surface method), 84
set_ctrlpts() (geomdl.BSpline.Volume method), 97
set_ctrlpts() (geomdl.NURBS.Curve method), 106
set_ctrlpts() (geomdl.NURBS.Surface method), 119
set_ctrlpts() (geomdl.NURBS.Volume method, AbstractManager method), 186
set_ctrlpts() (geomdl.NURBS.Volume method, CurveManager method), 188
set_ctrlpts() (geomdl.NURBS.Volume method, SurfaceManager method), 189
set_ctrlpts() (geomdl.NURBS.Volume method, VolumeManager method), 191
set_axes_equal() (geomdl.visualization.VisMPL.VisCurve2D method), 268
set_axes_equal() (geomdl.visualization.VisMPL.VisCurve3D method), 269
set_axes_equal() (geomdl.visualization.VisMPL.VisVolume method), 273
set_axes_equal() (geomdl.visualization.VisVTK.VisVolume method), 279
set_axes_equal() (geomdl.visualization.VisVTK.VisVoxel method), 279
SplineGeometry (class in geomdl.abstract), 225
split_curve() (in module geomdl.operations), 158
split_surface_u() (in module geomdl.operations), 159
split_surface_v() (in module geomdl.operations), 159
Surface (class in geomdl.abstract), 201
Surface (class in geomdl.BSpline), 73
Surface (class in geomdl.NURBS), 108
surface_deriv_cpts() (in module geomdl.helpers), 246
surface_tessellate() (in module geomdl.tessellate), 175
surface_tessellate() (in module geomdl.tessellate), 175
SurfaceContainer (class in geomdl.multi), 144
SurfaceEvaluator (class in geomdl.evaluators), 234
SurfaceEvaluator2 (class in geomdl.evaluators), 236
SurfaceEvaluatorRational (class in geomdl.evaluators), 235
SurfaceManager (class in geomdl.control_points),
vertex_ids (geomdl.elements.Triangle attribute), 258
vertices (geomdl.abstract.Surface attribute), 211
vertices (geomdl.BSpline.Surface attribute), 86
vertices (geomdl.elements.Quad attribute), 260
vertices (geomdl.elements.Triangle attribute), 259
vertices (geomdl.multi.SurfaceContainer attribute), 150
vertices (geomdl.NURBS.Surface attribute), 121
vertices (geomdl.tessellate.AbstractTessellate attribute), 171
vertices (geomdl.tessellate.QuadTessellate attribute), 173
vertices (geomdl.tessellate.TriangularTessellate attribute), 172
vertices (geomdl.tessellate.TrimTessellate attribute), 172
vertices_closed (geomdl.elements.Triangle attribute), 259
vis (geomdl.abstract.Curve attribute), 200
vis (geomdl.abstract.SplineGeometry attribute), 229
vis (geomdl.abstract.Surface attribute), 211
vis (geomdl.abstract.Volume attribute), 221
vis (geomdl.BSpline.Curve attribute), 72
vis (geomdl.BSpline.Surface attribute), 86
vis (geomdl.BSpline.Volume attribute), 97
vis (geomdl.multi.AbstractContainer attribute), 140
vis (geomdl.multi.CurveContainer attribute), 144
vis (geomdl.multi.SurfaceContainer attribute), 150
vis (geomdl.multi.VolumeContainer attribute), 155
vis (geomdl.NURBS.Curve attribute), 107
vis (geomdl.NURBS.Surface attribute), 121
vis (geomdl.NURBS.Volume attribute), 133
VisConfig (class in geomdl.visualization.VisMPL), 266
VisConfig (class in geomdl.visualization.VisVTK), 275
VisCurve2D (class in geomdl.visualization.VisMPL), 268
VisCurve2D (in module geomdl.visualization.VisVTK), 276
VisCurve3D (class in geomdl.visualization.VisMPL), 268
VisCurve3D (class in geomdl.visualization.VisVTK), 276
VisMPL (module), 266
VisSurface (class in geomdl.visualization.VisMPL), 271
VisSurface (class in geomdl.visualization.VisVTK), 276
VisSurfScatter (class in geomdl.visualization.VisMPL), 269
VisSurfTriangle (in module geomdl.visualization.VisMPL), 270