NURBS-Python Documentation

Onur Rauf Bingol

Feb 28, 2019
Introduction
Welcome to the NURBS-Python (geomdl) v5.x documentation! NURBS-Python is a 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 also provides a convenient and easy-to-use data structures for storing parametric shape descriptions. These are documented under Modules.

This documentation is organized into a couple sections:

- *Introduction*
- *Using the Library*
- *Modules*
CHAPTER 1

Motivation

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
- Customizable visualization and animation options with Matplotlib, Plotly and VTK modules
- Exporting curve, surface and volume data into various file formats, such as JSON, YAML, Libconfig, STL, OBJ and VTK
- Support for common algorithms: tessellation, voxelization, ray intersection, etc.
- Shapes component for generation common surfaces and curves
- Pure Python, no external C/C++ or FORTRAN library dependencies
- Python compatibility: 2.7.x, 3.4.x and later
- No compilation steps are necessary, everything is implemented in pure Python
- For higher performance, optional Compile with Cython options are also available
- Easy to install via pip: pip install geomdl or conda: conda install -c orbingol geomdl
- Docker images are available

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

• Dunn, Fletcher, and Ian Parberry. 3D Math Primer for Graphics and Game Development. CRC Press, 2015.
• Gamma, Erich. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education India, 1995.

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:

```
@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

- NURBS-Python source code is released under the terms of the MIT License. Please refer to the LICENSE file for details.
• NURBS-Python 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 (useful for non-Python and automated systems)
- Generating common spline shapes
• Rhino .3dm file import/export 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 is tested on Python versions 2.7.x and 3.4.x+. The core library does not depend on any additional packages or require any compilation steps; therefore, you can run it on a plain python installation as well as on a distribution, such as Anaconda.

Since Python 2.7.x and v3.4.x are reaching their end-of-life very soon, NURBS-Python project will only support Python 3.5+ starting with v6.0.0.

Please see the following page for more details: https://python3statement.org/

### 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.

### 3.6 Issues and Reporting

#### 3.6.1 Bugs and Feature Requests

Please use the issue tracker on GitHub for bug reporting and requesting a new feature.

#### 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.
3.7 API Changes

I try to keep the API (name and location of the functions, class fields and member functions) backward-compatible during minor version upgrades. During major version upgrades, the API change might not be backward-compatible. However, these changes will be kept minor and therefore, the users can update their code to the new version without much hassle. All of these changes, regardless of minor or major version upgrades, will be announced on the CHANGELOG file.

3.8 Installation Issues on Mac OS

Installation of six package from PyPI might cause some issues on some Mac OS versions. Six is a Python 2 to 3 compatibility library and allows NURBS-Python to run under both Python 2 and Python 3. If you are having issues regarding to installation of six package on Mac OS, please use the following command to install NURBS-Python (geomdl):

```bash
$ pip install geomdl --ignore-installed six
```

and to upgrade NURBS-Python (geomdl):

```bash
$ pip install geomdl --upgrade --ignore-installed six
```

or alternatively, you can use conda.

Please refer to the following issue on pip issue tracker for more details: https://github.com/pypa/pip/issues/3165

3.9 Conda Package on Python 2.7

You need to install functools_lru_cache module manually if you are using a Python 2.7 distribution managed via conda package manager (e.g. Anaconda, Miniconda).

The following command will install these packages in your active Python environment:

```bash
$ conda install functools_lru_cache
```
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.

### 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 geomdl
```

Upgrading to the latest version:

```
$ pip install geomdl --upgrade
```

Installing a specific version:

```
$ pip install 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 .

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 -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:

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

PS C:\> python
Python 3.6.2 (v3.6.2:5fd33b5, 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'
```

### 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 com-
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 . --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 . --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, the 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()

crv.degree = 2

# 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:
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 How to create a curve

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

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.

```
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)
```

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.

```
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

`delta` property will set the evaluation delta. It is also possible to use `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

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

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

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

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

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)

Please refer to the Examples Repository for more curve examples.

### 5.2 How to create a surface & a volume

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

Although using NURBS-Python is straight-forward, 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.
NURBS-Python provides the following methods for loading curve and surface data from a file:

- `BSpline.Curve.load()` and `NURBS.Curve.load()`
- `BSpline.Surface.load()` and `NURBS.Surface.load()`

Additionally, save functionality is provided via the following methods:

- `BSpline.Curve.save()` and `NURBS.Curve.save()`
- `BSpline.Surface.save()` and `NURBS.Surface.save()`

These functions implement Python’s `pickle` module to serialize the degree, knot vector and the control points data. The idea behind this system is only to provide users a basic data persistence capability, not to introduce a new file type. Since the data is pickled, it can be loaded with any compatible Python version even without using any special library.

The following example demonstrates the save functionality on a curve:
The saved curve can be loaded from the file with the following simple code segment:

```python
from geomdl import BSpline

# Create a B-Spline curve instance
curve2 = BSpline.Curve()

# Load the saved curve from a file
curve2.load("mycurve.pickle")
```

Since the load-save functionality implements Python’s `pickle` module, the saved file can also be loaded directly without using the NURBS-Python library.

```python
import pickle

# "data" variable will be a dictionary containing the curve information
data = pickle.load(open("mycurve.pickle"), "rb")
```

The `pickle` module has its own limitations by its design. Please see the Python documentation for more details.
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:

\[
\begin{array}{ccc}
    \text{x*w} & \text{y*w} & \text{w} \\
    x_1*w_1 & y_1*w_1 & w_1 \\
    x_2*w_2 & y_2*w_2 & w_2 \\
    x_3*w_3 & y_3*w_3 & w_3 \\
\end{array}
\]

**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:

\[
\begin{array}{ccc}
    \text{x} & \text{y} & \text{z} \\
    x_1 & y_1 & z_1 \\
    x_2 & y_2 & z_2 \\
    x_3 & y_3 & z_3 \\
\end{array}
\]

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:

\[
\begin{array}{cccc}
    \text{x*w} & \text{y*w} & \text{z*w} & \text{w} \\
    x_1*w_1 & y_1*w_1 & z_1*w_1 & w_1 \\
    x_2*w_2 & y_2*w_2 & z_2*w_2 & w_2 \\
    x_3*w_3 & y_3*w_3 & z_3*w_3 & w_3 \\
\end{array}
\]

Chapter 8. Supported File Formats
**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:

\[
\begin{array}{cccccc}
& v0 & v1 & v2 & v3 & v4 \\
\hline
u0 & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) \\
u1 & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) \\
u2 & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) & (x, y, z) \\
\end{array}
\]

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:

\[
\begin{array}{cccccc}
& v0 & v1 & v2 & v3 & v4 \\
\hline
u0 & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) \\
u1 & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) \\
u2 & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) & (x*w, y*w, z*w, w) \\
\end{array}
\]

**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)
surf.knotvector_u = utilities.generate_knot_vector(surf.degree_u, 4)
surf.knotvector_v = utilities.generate_knot_vector(surf.degree_v, 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 cylinndrical surface
surf = surface.cylinder(radius=5, height=12.5)

# Set evaluation delta
(continues on next page)
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 `libconf` 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.

```python
shape:
type: curve
data:
- 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, -1.0]
    - [-1.0, 0.0]
    - [-1.0, 1.0]
    - [0.0, 1.0]
    - [1.0, 1.0]
    - [1.0, 0.0]
    - [1.0, -1.0]
    - [0.0, -1.0]
  weights: # weights vector
  - 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 examples illustrates a NURBS surface:

```python
shape:
type: surface # define shape type
data:
- degree_u: 1 # degree of the u-direction
degree_v: 2 # degree of the v-direction
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
```

(continues on next page)
- [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, 0.0, 1.0]
- [1.0, 1.0, 1.0]
- [0.0, 1.0, 1.0]
- [-1.0, 1.0, 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]

weights:  # weights vector
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 0.7071
- 1.0
- 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

• **Shape section:** This section contains the single or multi NURBS data. The `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. The `weights` and `delta` sections are optional.

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

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

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

(continues on next page)
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()`
- `import_cfg()`
- `import_json()`
- `import_yaml()`

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_cylindertmpl.py` and `ex_cylindertmpl.cptw` files in the `Examples repository` for details on using Jinja2 templates with control point text files.
CHAPTER 9

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, 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
t_ctrlptsw = compat.combine_ctrlpts_weights(p_ctrlpts, p_weights)
```

# 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)
```

# Since we have no information on knot vectors, let's auto-generate them
```python
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)
```

# Import surface to NURBS-Python
```python
# Create a NURBS surface instance
surf = NURBS.Surface()
```

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

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

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

# Render the surface
```python
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.
CHAPTER 10

Surface Generator

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 BSpline.Surface and NURBS.Surface. It is capable of generating custom-sized 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 File Formats documentation.

The classes CPGen.Grid and CPGen.GridWeighted are responsible for generating surfaces and they are documented under Core Libraries.

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 VisPlotly

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

# Generate a grid of 25x30
surfgrid.generate(25, 30)

# Generate bumps on the grid
surfgrid.bumps(num_bumps=5, bump_height=20, base_extent=4)

# 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 = 30

# Generate the visualization component and its configuration
vis_config = VisPlotly.VisConfig(ctrlpts=False, legend=False)
vis_comp = VisPlotly.VisSurface(vis_config)

# Set visualization component
surf.vis = vis_comp

# Plot the surface
surf.render()

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 \(a\). **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.
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]

# 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.

```
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 = [
```
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
```

```python
# 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()
```
# 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, 0.2, 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, [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.

```python
from geomdl import NURBS
from geomdl import operations
from geomdl.visualization import VisMPL

# Control points
ctrlpts = [[25.0, -25.0, 0.0, 1.0], [15.0, -25.0, 0.0, 1.0], [5.0, -25.0, 0.0, 1.0],
           [-5.0, -25.0, 0.0, 1.0], [-15.0, -25.0, 0.0, 1.0], [-25.0, -25.0, 0.0, 1.0]],

          [[25.0, -15.0, 0.0, 1.0], [15.0, -15.0, 0.0, 1.0], [5.0, -15.0, 0.0, 1.0],
           [-5.0, -15.0, 0.0, 1.0], [-15.0, -15.0, 0.0, 1.0], [-25.0, -15.0, 0.0, 1.0]],

          [[25.0, -5.0, 5.0, 1.0], [15.0, -5.0, 5.0, 1.0], [5.0, -5.0, 5.0, 1.0],
           [-5.0, -5.0, 5.0, 1.0], [-15.0, -5.0, 5.0, 1.0], [-25.0, -5.0, 5.0, 1.0]],

          [[25.0, 5.0, 5.0, 1.0], [15.0, 5.0, 5.0, 1.0], [5.0, 5.0, 5.0, 1.0],
           [-5.0, 5.0, 5.0, 1.0], [-15.0, 5.0, 5.0, 1.0], [-25.0, 5.0, 5.0, 1.0]],

          [[25.0, 15.0, 0.0, 1.0], [15.0, 15.0, 0.0, 1.0], [5.0, 15.0, 5.0, 1.0],
           [-5.0, 15.0, 5.0, 1.0], [-15.0, 15.0, 0.0, 1.0], [-25.0, 15.0, 0.0, 1.0]],

          [[25.0, 25.0, 0.0, 1.0], [15.0, 25.0, 0.0, 1.0], [5.0, 25.0, 5.0, 1.0],
           [-5.0, 25.0, 5.0, 1.0], [-15.0, 25.0, 0.0, 1.0], [-25.0, 25.0, 0.0, 1.0]]

# Generate surface
surf = NURBS.Surface()

surf.degree_u = 3
surf.degree_v = 3
surf.ctrlpts2d = ctrlpts
surf.knotvector_u = [0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 3.0, 3.0, 3.0]
```

(continues on next page)
surf.knotvector_v = [0.0, 0.0, 0.0, 0.0, 1.0, 2.0, 3.0, 3.0, 3.0, 3.0]
surf.sample_size = 30

# Set visualization component
surf.vis = VisMPL.VisSurface(VisMPL.VisConfig(alpha=0.75))

# Refine knot vectors
operations.refine_knotvector(surf, [3, 0])

# Visualize
surf.render()
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.

12.1 Examples

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

12.1.1 Curves

12.1.2 Surfaces

12.1.3 Volumes

12.1.4 Advanced Visualization Examples

The following example scripts can be found in Examples repository under the visualization directory.
Chapter 12. Visualization
Chapter 12. Visualization
12.1. Examples
Chapter 12. Visualization
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.

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.
mpl_trisurf_vectors.py

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.

### 13.1 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.
13.2 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.
CHAPTER 14

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 (Bezier 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 modules are included in the core library:

15.1 Geometry Abstract 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.

15.1.1 Inheritance Diagram

15.1.2 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

    def evaluate_list(self, uv_list):
        # Implement this function
        pass

    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:**
- **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
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

data
Returns a dictionary containing all shape 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:

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

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

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

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
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

- **Getter** Gets the descriptor
- **Setter** Sets the descriptor

Type str

order
Order.

Defined as $\text{order} = \text{degree} + 1$

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
**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
    )
}
```

**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.

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:

\[
\begin{vmatrix}
  u_{\text{start}}, \ldots, u_{\text{end}} \\
  \end{vmatrix}
\]

\[n_{\text{sample}}\]

Please refer to the wiki for details on using this class member.
**Getter** Gets sample size

**Setter** Sets sample size

**Type** int

```python
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

**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.

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

**Getter** Gets the weights

**Setter** Sets the weights

### 15.1.3 Abstract Surface

```python
class geomdl.abstract.Surface(**kwargs)
```

Faces: 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):
```

(continues on next page)
```python
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

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:**

- `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

**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

---

(continued from previous page)
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 dictionary containing all shape 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.

15.1. Geometry Abstract Base
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 \((u, v, order, **kwargs)\)
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
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

name
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

Getter  Gets the descriptor
Setter  Sets the descriptor
Type    str

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
[dict1, dict2, ...]
```

Each dictionary contains the following keys:

- `x`: an array of x-coordinates
- `y`: an array of y-coordinates
- `z`: an array of z-coordinates
- `color`: color of the line

For example:

```python
dict1 = {'x': [0, 1], 'y': [0, 1], 'z': [0, 1], 'color': 'red'}
```

This will add a red line between the points (0,0,0) and (1,1,1).
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.

```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:

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

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
```

Trim curves.

Trim curves are introduced to the surfaces on the parametric space. It should be an array (or list, tuple, etc.) and they are integrated to the existing visualization system.

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
```

```
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
```

---

15.1. Geometry Abstract Base
weights

Weights.

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

**Getter** Gets the weights

**Setter** Sets the weights

### 15.1.4 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.

Code segment below illustrates a possible implementation of Volume abstract base class:

```python
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, u,v,w):
        # 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:**

- **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

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 dictionary containing all shape 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:

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

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.

**Note:** This is an abstract method and it must be implemented in the subclass.

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

e 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

**name**
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

- **Getter** Gets the descriptor
- **Setter** Sets the descriptor
- **Type** str

**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`

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 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**: (16, 16, 16)
- `use_mp`: flag to activate multi-threaded voxelization. **Default**: False
- `num_procs`: number of concurrent processes for multi-threaded voxelization. **Default**: 4

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.

The `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:
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:

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
**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**(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

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

**Getter**
Gets the weights

**Setter**
Sets the weights

### 15.1.5 Low Level API

The following classes provide the low level API for the geometry abstract base.

- **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.Geometry(**kwargs)
    Bases: object

    Abstract base class for defining geometry elements.
```
This class provides the following properties:

- `name`
- `evalpts`

**Keyword Arguments:**

- `precision`: number of decimal places to round to. *Default: 18*

**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.

**name**

Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

**Getter**

Gets the descriptor

**Setter**

Sets the descriptor

**Type**

`str`

---

**class geomdl.abstract.SplineGeometry(**kwargs)**

**Bases:** `geomdl.abstract.Geometry`

Abstract base class for defining spline geometries.

This class provides the following properties:

- `name`
- `rational`
- `dimension`
- `pdimension`
- `degree`
- `knotvector`
- `ctrlpts`
- `ctrlpts_size`
- `weights` (for completeness with the rational spline implementations)
- `evalpts`
- `bbox`
- `evaluator`
- `vis`
Keyword Arguments:

- **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

**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

**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`

knotvector

Knot vector

**Getter**  Gets the knot vector

**Setter**  Sets the knot vector

**Type**  list

name

Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

**Getter**  Gets the descriptor

**Setter**  Sets the descriptor

**Type**  str

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

**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.

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

**Getter**  Gets the weights

**Setter**  Sets the weights

## 15.2 B-Spline Geometry

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

![Inheritance Diagram]

15.2.2 B-Spline Curve

```python
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:

- 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]]
```

(continues on next page)
# 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 *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 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** an array containing “point” and “vector” pairs

  **Return type** tuple

**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

**data**
Returns a dictionary containing all shape 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:

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

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

**Getter**  Gets the delta value

**Setter**  Sets the delta value

**Type**  float

**derivatives** *(u, order=0, **kwargs)*
Evaluates n-th order curve derivatives at the given parameter value.

**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.
NURBS-Python Documentation

**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

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.

  **Parameters**  file_name (str) – name of the file to be loaded
  **Raises**  IOError – an error occurred reading the file

name
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

Please refer to the wiki for details on using this class member.
  
  **Getter**  Gets the descriptor
  **Setter**  Sets the descriptor
  **Type**  str

next ()
normal (parpos, **kwargs)
Evaluates the normal vector of the curve 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 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** an array containing “point” and “vector” pairs

    **Return type** tuple

---

**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  
    )
]
```

*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.

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:

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

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.

---

**Parameters**

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

---

**Raises**:

- IOError – an error occurred writing the file

**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**(*param, **kwargs)*

Evaluates the tangent vector of the curve 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 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**

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

**Returns**

an array containing “point” and “vector” pairs

**Return type**

tuple

**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**

tuple

**weights**

Weights.

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

**Getter**

Gets the weights

**Setter**

Sets the weights

---

### 15.2.3 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:

- 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
- trims

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)
```

(continues on next page)
# 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.

**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

**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:
# 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

15.2. B-Spline Geometry
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 dictionary containing all shape 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:

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

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
**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 \([\text{start}_u, \text{stop}_u]\)[\text{start}_v, \text{stop}_v]\ i.e. the surface will also be evaluated at the \text{stop}_u and \text{stop}_v parameter values.
  The following examples illustrate the usage of the keyword arguments.

1  # Start evaluating in range u=[0, 0.7] and v=[0.1, 1]
2  surf.evaluate(stop_u=0.7, start_v=0.1)
3  
4  # Start evaluating in range u=[0, 1] and v=[0.1, 0.3]
5  surf.evaluate(start_v=0.1, stop_v=0.3)
6  
7  # Get the evaluated points
8  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`

**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)
Load the surface from a pickled file.

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

Raises IOError – an error occurred reading the file

name
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

Getter Gets the descriptor

Setter Sets the descriptor

Type str

next()

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

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

15.2. B-Spline Geometry
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.

`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:

\[
\begin{align*}
\mathbf{u}_{\text{start}}, \ldots, \mathbf{u}_{\text{end}} \\
\mathbf{n}_{\text{sample}}
\end{align*}
\]

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

**Getter** Gets sample size as a tuple of values corresponding to u- 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.

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

**Raises**
- IOError – an error occurred writing the file

**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
Trim curves.
Trim curves are introduced to the surfaces on the parametric space. It should be an array (or list, tuple, etc.) and they are integrated to the existing visualization system.
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

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

    Getter  Gets the weights
    Setter  Sets the weights

15.2.4 B-Spline Volume

New in version 5.0.

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:

- 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.

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

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 dictionary containing all shape 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:

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

  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 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`

`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

15.2. B-Spline Geometry
**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.

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

**Raises IOError** – an error occurred reading the file

**name**

Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

**Getter** Gets the descriptor

**Setter** Sets the descriptor

**Type** str

**next ()**
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.
  \textbf{Getter} Gets the surface order for u-direction
  \textbf{Setter} Sets the surface order for u-direction
  \textbf{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.
  \textbf{Getter} Gets the surface order for v-direction
  \textbf{Setter} Sets the surface order for v-direction
  \textbf{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.
  \textbf{Getter} Gets the surface order for v-direction
  \textbf{Setter} Sets the surface order for v-direction
  \textbf{Type} int

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

range
Domain range.
  \textbf{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.
  \textbf{Getter} Returns True if the B-spline object is rational (NURBS)
  \textbf{Type} bool

remove_knot \((u=\text{None}, v=\text{None}, w=\text{None}, **\text{kwargs})\)
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: (16, 16, 16)
- **use_mp**: flag to activate multi-threaded voxelization. Default: False
- **num_procs**: number of concurrent processes for multi-threaded voxelization. Default: 4

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
    )
}]
```

`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:

\[
\begin{bmatrix}
\ldots, \, u_{\text{end}} \end{bmatrix}
\]

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

**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.

**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.

**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.

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

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

**Raises**
- `IOError` – an error occurred writing the file
**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

**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.

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

- **Getter** Gets the weights
- **Setter** Sets the weights

### 15.3 NURBS Geometry

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

#### 15.3.1 Inheritance Diagram

![Inheritance Diagram](image)

#### 15.3.2 NURBS Curve

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
- `ctlpts`: 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`
- `ctlpts`
- `weights`
- `delta`
- `sample_size`
- `bbox`
- `vis`
- `name`
- `dimension`
- `evaluator`
- `rational`

The following code segment illustrates the usage of Curve class:

```python
from geomdl import NURBS

curve = NURBS.Curve()

curve.degree = 3

curve.ctrlpts = [[10, 5, 10], [10, 20, -30], [40, 10, 25], [-10, 5, 0]]

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

curve.delta = 0.05

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** *(param, **kwargs)*

Evaluates the binormal vector of the curve 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 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** `param` *(float, list or tuple)* – parametric position(s) where the evaluation will be executed

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

  **Return type** tuple

**ctrlpts**

Control points (P).

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

  **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
**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.

- **Getter** Gets the weighted control points
- **Setter** Sets the weighted control points

**data**
Returns a dictionary containing all shape 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:

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

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

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

**derivatives** *(u, order=0, **kwargs)*
Evaluates n-th order curve derivatives at the given parameter value.

**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.
**domain**

Domain.

Domain is determined using the knot vector(s).

**evalpts**

Evaluated points.

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

**evaluate(**

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)

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.

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

**Raises**
- IOError – an error occurred reading the file

name
Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

- **Getter** Gets the descriptor
- **Setter** Sets the descriptor
- **Type** str

next ()

normal (parpos, **kwargs)
Evaluates the normal vector of the curve 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 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**

An array containing “point” and “vector” pairs

**Return type**

tuple

**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 if 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*

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.

The `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
    )
]
```

**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.

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:

$$
\begin{bmatrix}
    u_{\text{start}}, \ldots, u_{\text{end}} \\
    n_{\text{sample}}
\end{bmatrix}
$$

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.
   Parameters file_name (str) – name of the file to be saved
   Raises IOError – an error occurred writing the file

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 (param, **kwargs)
   Evaluates the tangent vector of the curve 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 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 param (float, list or tuple) – parametric position(s) where the evaluation will be executed
   Returns an array containing “point” and “vector” pairs
   Return type tuple

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
15.3.3 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)
# Use curve.ctrlptsw is if you are using homogeneous points as Pw
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.

**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

**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.
**NURBS-Python Documentation**

**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

`ctrlptsuw`  
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 dictionary containing all shape 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:

\[ [u_0, u_{\text{start}} + \delta, (u_{\text{start}} + 2\delta), \ldots, u_{\text{end}}] \]

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

• \( u (\text{float}) \) – parameter on the u-direction

• \( v (\text{float}) \) – parameter on the v-direction

• \( \text{order} (\text{integer}) \) – derivative order

Returns A list \( \text{SKL} \), where \( \text{SKL}[k][l] \) 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 \( \text{evalpts} \) property.

Keyword arguments:

• \( \text{start\_u} \): start parameter on the u-direction

• \( \text{stop\_u} \): stop parameter on the u-direction

• \( \text{start\_v} \): start parameter on the v-direction

• \( \text{stop\_v} \): stop parameter on the v-direction

The \( \text{start\_u} \), \( \text{start\_v} \) and \( \text{stop\_u} \) and \( \text{stop\_v} \) parameters allow evaluation of a surface segment in the range \( [\text{start\_u}, \text{stop\_u}] / [\text{start\_v}, \text{stop\_v}] \) i.e. the surface will also be evaluated at the \( \text{stop\_u} \) and \( \text{stop\_v} \) 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)
```
(continues on next page)
# 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

**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](https://example.com) 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](https://example.com) 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.

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

**Raises** IOError – an error occurred reading the file

`name`

Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

Please refer to the [wiki](https://example.com) for details on using this class member.

**Getter** Gets the descriptor

**Setter** Sets the descriptor

**Type** str

`next()`

`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
**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 order for the u-direction
- **Setter** Sets order for the 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 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:

    ```
    [
        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.

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:

\[ [u_{start}, \ldots, u_{end}] \]

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 \( \text{delta}_u \) 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 \( \text{delta}_v \) 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

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

- **Parameters** file_name *(str)* – name of the file to be saved
- **Raises** IOError – an error occurred writing the file

**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

15.3. NURBS Geometry
**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**
Trim curves.

Trim curves are introduced to the surfaces on the parametric space. It should be an array (or list, tuple, etc.) and they are integrated to the existing visualization system.

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

**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
15.3.4 NURBS Volume

New in version 5.0.

```python
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.

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

ctrllsts
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

ctrllpts_size
Total number of control points.

  Getter Gets the total number of control points
  Type int

ctrllpts_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 dictionary containing all shape 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:

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

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 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`

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.

- **Parameters** file_name (str) – name of the file to be loaded
- **Raises** IOError – an error occurred reading the file

**name**

Descriptor field for storing the shape identification data, such as names, ID numbers, etc.

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

- **Getter** Gets the descriptor
- **Setter** Sets the descriptor

Type str

**next ()**

**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 \( \text{order} = \text{degree} + 1 \)
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 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: (16, 16, 16)
- `use_mp`: flag to activate multi-threaded voxelization. Default: False
- `num_procs`: number of concurrent processes for multi-threaded voxelization. Default: 4

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
  )
]
```

**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:
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

**save**(*file_name*)

Saves the volume as a pickled file.

**Parameters**

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

**Raises**  IOError – an error occurred writing the file

**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 

**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
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

15.4 Evaluators

Evaluators (or geometric evaluation strategies) allow users to change shape evaluation strategy, i.e. the algorithms that are used to evaluate curves, surfaces and volumes, take derivatives and more. Therefore, the user can switch between the evaluation algorithms at runtime, implement and use different algorithms or extend existing ones.

15.4.1 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.

```python
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
```
15.4.2 Inheritance Diagram

15.4.3 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(**kwargs)
    Abstract method for computation of derivatives at a single parameter.

    Note: This is an abstract method and it must be implemented in the subclass.

evaluate(**kwargs)
    Abstract method for computation of points over a range of parameters.

    Note: This is an abstract method and it must be implemented in the subclass.

name
    Evaluator name.

    Getter Gets the name of the evaluator

    Type str

15.4.4 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.

def derivatives(**kwargs)
    Evaluates the derivatives at the input parameter.

def evaluate(**kwargs)
    Evaluates the curve.

def name
    Evaluator name.
        Getter Gets the name of the evaluator
        Type str

class geomdl.evaluators.CurveEvaluator2(**kwargs)
    Bases: geomdl.evaluators.CurveEvaluator
    Sequential curve evaluation algorithms (alternative).

This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.1: CurvePoint
- Algorithm A3.4: CurveDerivsAlg2

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(**kwargs)
    Evaluates the derivatives at the input parameter.

def derivatives_ctrlpts(**kwargs)
    Computes the control points of all derivative curves up to and including the {degree}-th derivative.
        Implementation of Algorithm A3.3 from The NURBS Book by Piegl & Tiller.
        Output is PK[k][i], i-th control point of the k-th derivative curve where 0 <= k <= degree and r1 <= i <= r2-k.

def evaluate(**kwargs)
    Evaluates the curve.

def 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.

def derivatives (**kwargs)
    Evaluates the derivatives at the input parameter.

def evaluate (**kwargs)
    Evaluates the rational curve.

def name
    Evaluator name.

    Getter  Gets the name of the evaluator

    Type   str

15.4.5 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.

def derivatives (**kwargs)
    Evaluates the derivatives at the input parameter.

def evaluate (**kwargs)
    Evaluates the surface.

def 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.

    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.

**derivatives (**kwargs**)**
Evaluates the derivatives at the input parameter.

**static derivatives_ctrlpts (**kwargs**)**
Computes the control points of all derivative surfaces up to and including the \{degree\}-th derivative.

Output is PKL[k][l][i][j], i,j-th control point of the surface differentiated k times w.r.t to u and l times w.r.t v.

**evaluate (**kwargs**)**
Evaluates the surface.

**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 (**kwargs**)**
Evaluates the derivatives at the input parameter.

**evaluate (**kwargs**)**
Evaluates the rational surface.

**name**
Evaluator name.

**Getter** Gets the name of the evaluator

**Type** str

### 15.4.6 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(**kwargs)
Evaluates the volume.

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(**kwargs)
Evaluates the rational volume.

name
Evaluator name.

Getter  Gets the name of the evaluator

Type  str

15.5 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

15.5.1 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

```python
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

```python
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:
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 knot vector to be refined is \([0, 2, 4]\) with the superfluous knots from the start and end are removed:

- If *density* is 1, knot vector to be refined is \([0, 2, 4]\)
- If *density* is 2, knot vector to be refined is \([0, 1, 2, 3, 4]\)
- If *density* is 3, knot vector to be refined is \([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4]\)

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])
```

**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.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

```python
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

```python
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

```python
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

```python
geomdl.operations.add_dimension(obj, **kwargs)
```

Converts x-dimensional curve to a (x+1)-dimensional curve.

If you pass `inplace=True` keyword argument, the input shape will be updated. Otherwise, this function does not change the input shape but returns a new instance of the same shape with the updated data.

Useful when converting a 2-dimensional curve to a 3-dimensional curve. Please note that this function does not change the parametric dimension but only the spatial dimension.

15.5. Geometric Operations 161
Parameters `obj (abstract.Curve)` – Curve

Returns updated Curve

Return type `BSpline.Curve` or `NURBS.Curve`

`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.translate(obj, vec, **kwargs)
Translates curves, surface or volumes by the input vector.
If you pass inplace=True keyword argument, the input shape will be updated. Otherwise, this function does not change the input shape but returns a new instance of the same shape with the updated data.

Parameters
• obj (abstract.SplineGeometry or multi.AbstractContainer) – input geometry
• vec (list, tuple) – translation vector

Returns translated geometry object

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
• params (float, list or tuple) – parameters

Returns a list containing “point” and “vector” pairs

Return type tuple

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

gemdl.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

gemdl.operations.find_ctrlpts(obj, u, v=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

gemdl.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.
- inplace: if True, the input shape is modified. Default: False

Parameters
- obj (abstract.Curve, abstract.Surface or abstract.Volume) – input geometry
- 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 True, the input shape is modified. Default: False

Parameters
- obj (abstract.Curve, abstract.Surface or abstract.Volume) – input geometry
- multiplier (float) – scaling multiplier
Returns scaled geometry object

```python
geomdl.operations.transpose(surf, **kwargs)
```

Transposes the input surface by swapping u and v parametric directions.

If you pass `inplace=True` keyword argument, the input surface will be updated. Otherwise, this function does not change the input surface but returns a new instance of the same surface with the updated data.

**Parameters**
- `surf` (abstract.Surface) – input surface
- `**kwargs` (optional) – keyword arguments

**Returns**
- transposed surface (abstract.Surface)

---

### 15.6 Utility Functions

These modules contain common utility and helper functions for B-Spline / NURBS curve and surface evaluation operations.

#### 15.6.1 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) – sequence of points
- `size_u` (int) – size of u direction
- `size_v` (int) – size of v direction
• **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`

```python
geomdl.utilities.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** – number of points on the v-direction (row)

**Returns** a tuple containing lists of vertices and quads

**Return type** `tuple`

```python
geomdl.utilities.make_quadtree(points, size_u, size_v, **kwargs)
```

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`

```python
geomdl.utilities.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
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.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:

```
num cols
<--------------------->
--------->---------|
|--------<--------|
|-------->--------|
--------<--------|
```

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

`geomdl.utilities.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 (elements.Vertex)` – list of Vertex objects

**Returns** list of Triangle objects

**Return type** list
geomdl.utilities.triangle_center(tri, uv=False)
Computes the center of mass of the input triangle.

Parameters
• tri (elements.Triangle) – triangle object
• uv (bool) – if True, then finds parametric position of the center of mass

Returns center of mass of the triangle
Return type tuple

geomdl.utilities.triangle_normal(tri)
Computes the (approximate) normal vector of the input triangle.

Parameters tri (elements.Triangle) – triangle object

Returns normal vector of the triangle
Return type tuple

15.6.2 Helpers

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.

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.

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

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

geomdl.helpers.basis_functions\((degree, knot_vector, spans, knots)\)
Computes the non-vanishing basis functions for a list of parameters.

Parameters
- \(degree\ (int)\) – degree, \(p\)
- \(knot\_vector\ (list, tuple)\) – knot vector, \(U\)
NURBS-Python Documentation

- **spans** *(list, tuple)* – list of knot spans
- **knots** *(list, tuple)* – list of knots or parameters

**Returns** basis functions

**Return type** list

`geomdl.helpers.basis_functions_ders(degree, knot_vector, spans, knots, order)`

Computes derivatives of the basis functions for a list of parameters.

**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.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.findMultiplicity(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.findSpanBinsearch(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 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.findSpanLinear(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.findSpans(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, \( n + 1 \)
NURBS-Python Documentation

- **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`

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 knot vector to be refined is [0, 2, 4] with the superfluous knots from the start and end are removed:

- If `density` is 1, knot vector to be refined is [0, 2, 4]
- If `density` is 2, knot vector to be refined is [0, 1, 2, 3, 4]
- If `density` is 3, knot vector to be refined is [0, 0.5, 1, 1.5, 2, 2.5, 3, 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
### 15.6.3 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 $k$ choose $i$).

Please see the following website for details: [http://mathworld.wolfram.com/BinomialCoefficient.html](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.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](https://stackoverflow.com/a/36091634)

- **Parameters**
  - `start` (float) – start value
  - `stop` (float) – end value
  - `step` (float) – increment
Returns float

Return type generator

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.matrix_multiply(ml, m2)

Matrix multiplication (iterative algorithm).

The running time of the iterative matrix multiplication algorithm is $O(n^3)$.

Parameters
  • ml (list, tuple) – 1st matrix with dimensions $(n \times p)$
  • m2 (list, tuple) – 2nd matrix with dimensions $(p \times m)$

Returns resultant matrix with dimensions $(n \times m)$

Return type list

geomdl.linalg.matrix_transpose(m)

Transposes the input matrix.

The input matrix $m$ is a 2-dimensional array.

Parameters m (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
• \texttt{pt1(list, tuple)} – point 1
• \texttt{pt2(list, tuple)} – point 2

\textbf{Returns} distance between input points

\textbf{Return type} float

\texttt{geomdl.linalg.point_mid(pt1, pt2)}

Computes the midpoint of the input points.

\textbf{Parameters}

• \texttt{pt1(list, tuple)} – point 1
• \texttt{pt2(list, tuple)} – point 2

\textbf{Returns} midpoint

\textbf{Return type} list

\texttt{geomdl.linalg.point_translate(point_in, vector_in)}

Translates the input points using the input vector.

\textbf{Parameters}

• \texttt{point_in(list, tuple)} – input point
• \texttt{vector_in(list, tuple)} – input vector

\textbf{Returns} translated point

\textbf{Return type} list

\texttt{geomdl.linalg.vector_angle_between(vector1, vector2, **kwargs)}

Computes the angle between the two input vectors.

If the keyword argument \texttt{degrees} is set to \texttt{True}, then the angle will be in degrees. Otherwise, it will be in radians. By default, \texttt{degrees} is set to \texttt{True}.

\textbf{Parameters}

• \texttt{vector1(list, tuple)} – vector
• \texttt{vector2(list, tuple)} – vector

\textbf{Returns} angle between the vectors

\textbf{Return type} float

\texttt{geomdl.linalg.vector_cross(vector1, vector2)}

Computes the cross-product of the input vectors.

\textbf{Parameters}

• \texttt{vector1(list, tuple)} – input vector 1
• \texttt{vector2(list, tuple)} – input vector 2

\textbf{Returns} result of the cross product

\textbf{Return type} tuple

\texttt{geomdl.linalg.vector_dot(vector1, vector2)}

Computes the dot-product of the input vectors.

\textbf{Parameters}

• \texttt{vector1(list, tuple)} – input vector 1
NURBS-Python Documentation

• `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
from geomdl import 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 $\mathbf{v}_1 + c * \mathbf{v}_2$, 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

### 15.7 Knot Vector Generator

The knotvector module provides utility functions related to knot vector generation and validation.

#### 15.7.1 Function Reference

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

### 15.8 Geometry Converters

`convert` module provides functions for converting non-rational and rational geometries to each other.

#### 15.8.1 Function Reference

```
geomdl.convert.bspline_to_nurbs(obj)
```

Converts non-rational parametric shapes to rational ones.

**Parameters** `obj` *(BSpline.Curve, BSpline.Surface or BSpline.Volume)* – B-Spline shape

**Returns** NURBS shape

**Return type** `NURBS.Curve`, `NURBS.Surface` or `NURBS.Volume`

**Raises** TypeError
NURBS-Python Documentation

geomdl.convert.nurbs_to_bspline(obj, **kwargs)
Extracts the non-rational components from rational parametric shapes, if possible.
The possibility of converting a rational shape to a non-rational one depends on the weights vector.

Parameters obj (NURBS.Curve, NURBS.Surface or NURBS.Volume) – NURBS shape
Returns B-Spline shape
Return type BSpline.Curve, BSpline.Surface or BSpline.Volume
Raises TypeError

15.9 Geometry Constructors and Extractors

New in version 5.0.

construct module provides functions for constructing and extracting parametric shapes. A surface can be con-
structed 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.

15.9.1 Function Reference

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

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
**NURBS-Python Documentation**

```python
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`

```python
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`

```python
geomdl.construct.extract_surfaces(pvol)
```

Extracts surfaces from a volume.

**Parameters**

- `pvol (abstract.Volume)`: input volume

**Returns**

- `extracted surface`

**Return type**

- `dict`

### 15.10 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

### 15.10.1 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:

![Heart Valve with 3 Leaflets](image)

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.
15.10.2 Inheritance Diagram

```
geomdl.multi.AbstractContainer
geomdl.multi.CurveContainer
geomdl.multi.SurfaceContainer
geomdl.multi.VolumeContainer
```

15.10.3 Abstract Container

```python
class geomdl.multi.AbstractContainer(*args, **kwargs):
    Bases: object

    Abstract class for curve and surface 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:
    • dimension
    • evalpts
    • bbox
    • vis
    • delta
    • sample_size

    add(element)
    Adds shapes to the container.
    The input can be a single shape, a list of shapes or a container object.

    Parameters element -- shape to be added

    append(element)
    Adds shapes to the container.
    The input can be a single shape, a list of shapes or a container object.

    Parameters element -- shape to be added

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

    Getter Gets the bounding box of all contained shapes

    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**
Shape dimension.

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

**Getter** Gets the dimension of the shape

**evalpts**
Evaluated points.

Since there are multiple shapes 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 shapes 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 shapes

**render** (**kwargs**)
Renders plots using the visualization component.

**Note:** This is an abstract method and it must be implemented in the subclass.

**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:

\[ [u_{start}, \ldots, u_{end}] \]

\[ n_{sample} \]

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

**Getter** Gets sample size

**Setter** Sets sample size
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

### 15.10.4 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:

- `dimension`
- `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 shapes to the container.

The input can be a single shape, a list of shapes or a container object.

**Parameters**
- `element` – shape to be added

**append(element)**

Adds shapes to the container.

The input can be a single shape, a list of shapes or a container object.

**Parameters**
- `element` – shape to be added

**bbox**

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

**Getter** Gets the bounding box of all contained shapes

### 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
Shape dimension.

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

**Getter** Gets the dimension of the shape

### evalpts
Evaluated points.

Since there are multiple shapes 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 shapes 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 shapes

### 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 Multi object will be used. **Default: True**

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.

**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:

\[
\begin{align*}
&\left[ u_{\text{start}}, \ldots, u_{\text{end}} \right] \\
&n_{\text{sample}}
\end{align*}
\]

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

- **Getter** Gets sample size
- **Setter** Sets sample size

**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

### 15.10.5 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:

- `dimension`
- `evalpts`
- `bbox`
- `vis`
- `delta`
- `delta_u`
- `delta_v`
- `sample_size`
• `sample_size_u`
• `sample_size_v`

The following code example illustrates the usage of these Python properties:

```python
# 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
```

### `add(element)`

Adds shapes to the container.

- The input can be a single shape, a list of shapes or a container object.
- **Parameters**
  - `element` – shape to be added

### `append(element)`

Adds shapes to the container.

- The input can be a single shape, a list of shapes or a container object.
- **Parameters**
  - `element` – shape to be added

### `bbox`

Bounding box.

- Please refer to the wiki for details on using this class member.
- **Getter** Gets the bounding box of all contained shapes

### `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

`dimension`

Shape dimension.

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

**Getter**  Gets the dimension of the shape

`evalpts`

Evaluated points.

Since there are multiple shapes 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 shapes 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 shapes

`render(**kwargs)`

Renders the surfaces.

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

**Keyword Arguments:**

- `cpcolor`: sets the color of the control points grids
- `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*

• **colormap**: sets the colormap of the surfaces

• **delta**: if True, the evaluation delta of the Multi object will be used. *Default: True*

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.

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 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.

### 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:

\[
\begin{align*}
\mathbf{u}_{\text{start}}, \ldots, \mathbf{u}_{\text{end}} \\
\mathbf{n}_{\text{sample}}
\end{align*}
\]

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
**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

### 15.10.6 Volume Container

class geomdl.multi.VolumeContainer(*args, **kwargs)

Bases: geomdl.multi.SurfaceContainer

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:

- **dimension**
- **evalpts**
- **bbox**
- **vis**
- **delta**
- **delta_u**
- **delta_v**
- **delta_w**
- **sample_size**
- **sample_size_u**
- **sample_size_v**
- **sample_size_w**

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 shapes to the container.

The input can be a single shape, a list of shapes or a container object.

**Parameters** element – shape to be added
**append** *(element)*

Adds shapes to the container.

The input can be a single shape, a list of shapes or a container object.

**Parameters**

- **element** – shape to be added

**bbox**

Bounding box.

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

**Getter**

Gets the bounding box of all contained shapes

**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**
Shape dimension.

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

**Getter** Gets the dimension of the shape

**evalpts**
Evaluated points.

Since there are multiple shapes 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 shapes 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 shapes

**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 Multi object will be used. *Default: True*
- `grid_size`: grid size for voxelization. *Default: (16, 16, 16)*
• **use_mp**: flag to activate multi-threaded voxelization. *Default: False*

• **num_procs**: number of concurrent processes for multi-threaded voxelization. *Default: 4*

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.

**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

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

15.11 Compatibility and Conversion

This module contains conversion operations related to control points, such as flipping arrays and adding weights.

15.11.1 Function Reference

`geomdl.compatibility.combine_ctrlpts_weights(ctrlpts, weights=None)`
Multiplies 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

`geomdl.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

`geomdl.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

gemdl.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

gemdl.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

```python
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 \((P_w)\)

**Return type**
- list

```python
geomdl.compatibility.generate_ctrlptsw(ctrlpts)
```
Generates weighted control points from unweighted ones in 1-D.

This function
1. Takes in a 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 \((P_w)\)

**Return type**
- list

```python
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 \((P_w)\)

**Return type**
- list

```python
geomdl.compatibility.generate_ctrlptsw2d_file(file_in='', 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**

### 15.12 Curve and Surface Fitting

New in version 5.0.

The `fitting` module provides functions for interpolating and approximating B-spline curves and surfaces from points. It provides functions for global interpolation and approximation to a list of data points. Please see the following functions for details:

- **interpolate_curve()**
- **interpolate_surface()**
- **approximate_curve()**
- **approximate_surface()**

#### 15.12.1 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`

gemdl.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`

gemdl.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`

### 15.13 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.

#### 15.13.1 Grid

```python
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 \((\text{num}_u, \text{num}_v)\)
Generates grid using the input division parameters.

Parameters
- \(\text{num}_u\) (int) – number of divisions in x-direction
- \(\text{num}_v\) (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.

15.13.2 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
- \(\text{size}_x\) (float) – width of the grid
- \(\text{size}_y\) (float) – height of the grid

bumps \((\text{num}_bumps, **\text{kwargs})\)
Generates arbitrary bumps (i.e. hills) on the 2-dimensional grid.
This method generates hills on the grid defined by the \(\text{num}_bumps\) argument. It is possible to control the \(z\)-value using \(\text{bump\_height}\) argument. \(\text{bump\_height}\) 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 \(\text{num}_bumps\) number of bumps. Therefore, this function uses a brute-force algorithm to determine whether the bumps can be generated or not. For instance:

```
<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_grid = Grid(5, 10)</td>
<td># generates a 5x10 rectangle</td>
</tr>
<tr>
<td>test_grid.generate(4, 4)</td>
<td># splits the rectangle into 2x2 pieces</td>
</tr>
<tr>
<td>test_grid.bumps(100)</td>
<td># impossible, it will return an error message</td>
</tr>
<tr>
<td>test_grid.bumps(1)</td>
<td># You will get a bump at the center of the generated grid</td>
</tr>
</tbody>
</table>
```

This method accepts the following keyword arguments:
- \(\text{bump\_height}\): \(z\)-value of the generated bumps on the grid. \textit{Default: 5.0}
- \(\text{base\_extent}\): extension of the hill base from its center in terms of grid points. \textit{Default: 2}
- \(\text{base\_adjust}\): padding between the bases of the hills. \textit{Default: 0}

Parameters \(\text{num}_bumps\) (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

### 15.14 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()`

### 15.14.1 Function Reference

```python
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.

```python
# Import curve control points from a text file
curve_ctrlpts = exchange.import_txt(file_name="control_points.txt")

# Import surface control points from a text file (1-dimensional file)
surf_ctrlpts = exchange.import_txt(file_name="control_points.txt")

# Import surface control points from a text file (2-dimensional file)
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): \sqrt{x}`
- `cubert(x): 3\sqrt{x}`
- `pow(x, y): 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.

```python
# Import curve control points from a text file delimited with space
curve_ctrlpts = exchange.import_txt(file_name="control_points.txt", separator=" ")

# Import surface control points from a text file (2-dimensional file) w/ space and comma delimiters
surf_ctrlpts, size_u, size_v = exchange.import_txt(file_name="control_points.txt", two_dimensional=True, separator=" ", col_separator=" , ")
```

(continues on next page)
surf_ctrlpts, size_u, size_v = exchange.import_txt(file_name="control_points.txt",
                                           two dimensional=True,
                                           separator=" ", coll separator=" ",
                                           )

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 IOError – 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-D control point output file using two dimensional flag. Please see the supported file formats for more details on the text file format.

Please see exchange.import_txt() for detailed description of the keyword arguments.

Parameters
• obj (abstract.Curve, abstract.Surface) – a curve or a surface 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 IOError – 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.

# By default, import_csv uses 'comma' as the value separator
ctrlpts = exchange.import_csv("control_points.csv")

# Alternatively, it is possible to import a file containing tab-separated values
ctrlpts = exchange.import_csv("control_points.csv", separator="	")

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 IOError – 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
### geomdl.exchange.import_cfg(file_name, **kwargs)
Imports curves and surfaces from files in libconfig format.

**Parameters**
- **file_name (str)** — name of the input file

**Returns**
a list of NURBS curve(s) or surface(s)

**Return type**
list

**Raises**
`IOError` – an error occurred writing the file

**Note:** Requires libconf package.

Use `jinja2=True` to activate Jinja2 template processing. Please refer to the documentation for details.

### geomdl.exchange.export_cfg(obj, file_name)
Exports curves and surfaces in libconfig format.

**Parameters**
- **obj (abstract.Curve, abstract.Surface, multi.CurveContainer or multi.SurfaceContainer)** — input curve(s) or surface(s)
- **file_name (str)** — name of the output file

**Raises**
`IOError` – an error occurred writing the file

**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.

### geomdl.exchange.import_yaml(file_name, **kwargs)
Imports curves and surfaces from files in YAML format.

**Parameters**
- **file_name (str)** — name of the input file

**Returns**
a list of NURBS curve(s) or surface(s)

**Return type**
list

**Raises**
`IOError` – an error occurred reading the file

**Note:** Requires ruamel.yaml package.

Use `jinja2=True` to activate Jinja2 template processing. Please refer to the documentation for details.

### geomdl.exchange.export_yaml(obj, file_name)
Exports curves and surfaces in YAML format.

**Parameters**
- **obj (abstract.Curve, abstract.Surface, multi.CurveContainer or multi.SurfaceContainer)** — input curve(s) or surface(s)
- **file_name (str)** — name of the output file

**Raises**
`IOError` – an error occurred reading the file

**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.Curve], [abstract.Surface], [multi.CurveContainer] or [multi.SurfaceContainer]) – input curve(s) or surface(s)
- `file_name` (str) – name of the output file

**Raises** `IOError` – 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 NURBS curve(s) or surface(s)

**Return type** list

**Raises** `IOError` – 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.Curve], [abstract.Surface], [multi.CurveContainer] or [multi.SurfaceContainer]) – input curve(s) or surface(s)
- `file_name` (str) – name of the output file

**Raises** `IOError` – 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
• update_delta: use multi-surface evaluation delta for all surfaces. Default: False

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises IOError – an error occurred writing the file

gemdl.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: 2
• update_delta: use multi-surface evaluation delta for all surfaces. Default: False

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises IOError – an error occurred writing the file

gemdl.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: 2
• update_delta: use multi-surface evaluation delta for all surfaces. Default: False

Parameters

- surface (abstract.Surface or multi.SurfaceContainer) – surface or surfaces to be saved
- file_name (str) – name of the output file

Raises IOError – an error occurred writing the file

gemdl.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** `IOError` – an error occurred reading the file

`geomdl.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** `IOError` – an error occurred writing the file

`geomdl.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** `IOError` – an error occurred reading the file

`geomdl.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** `IOError` – an error occurred writing the file

`geomdl.exchange.import_3dm(file_name, **kwargs)`
Imports Rhinoceros/OpenNURBS .3dm file format.

**Parameters**
- `file_name (str)` – input file name

**Requires** `rw3dm` module: https://github.com/orbingol/rw3dm

`geomdl.exchange.export_3dm(obj, file_name, **kwargs)`
Exports NURBS curves and surfaces in Rhinoceros/OpenNURBS .3dm format.

**Parameters**

**Note:** Requires `rw3dm` module: https://github.com/orbingol/rw3dm
• **obj** *(abstract.Curve, abstract.Surface, multi.CurveContainer, multi.SurfaceContainer)* – curves/surfaces to be exported

• **file_name**(str) – file name

### 15.14.2 VTK Support

The following functions export control points and evaluated points as VTK files (in legacy format).

```
geomdl.exchange_vtk.export_polydata(obj, file_name, point_type='evalpts', **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](http://www.vtk.org/VTK/img/file-formats.pdf)

**Parameters**

• **obj**(abstract.Curve, abstract.Surface) – a curve or a surface object

• **file_name**(str) – output file name

• **point_type**(str) – ctrlpts for control points or evalpts for evaluated points

**Raises** IOError – an error occurred writing the file

### 15.15 Tessellation

The `tessellate` module provides tessellation algorithms for surfaces. The following example illustrates the usage scenario of the tessellation algorithms with surfaces.

```
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.

### 15.15.1 Abstract Tessellator

```
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
```

---

210 Chapter 15. Core Modules
Setter Sets the tessellation arguments

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

15.15.2 Triangular Tessellator

class geomdl.tessellate.TriangularTessellate(**kwargs)
Bases: geomdl.tessellate.AbstractTessellate
Triangular tessellation algorithm for surfaces.
This class provides the default 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
Setter Sets the tessellation arguments

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

15.15.3 Trim Tessellator

New in version 5.0.
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
   Setter  Sets the tessellation arguments
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

---

**15.16 Voxelization**

New in version 5.0.

voxelize module provides functions for voxelizing NURBS volumes. voxelize() also supports multi-threaded operations via multiprocessing module.

**15.16.1 Function Reference**

**geomdl.voxelize.** **voxelize**(obj, **kwargs)  
Generates binary voxel representation of the surfaces and volumes.

**Keyword Arguments:**
- grid_size: size of the voxel grid. Default: (16, 16, 16)
- padding: voxel padding for in-outs finding. Default: 10e-8
- use_mp: flag to activate multi-threaded voxelization. Default: False
- num_procs: number of concurrent processes for multi-threaded voxelization. Default: 4

**Parameters**  
obj (abstract.Surface or abstract.Volume) – input surface(s) or volume(s)

**Returns**  
voxel grid and filled information

**Return type**  
tuple

**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
15.17 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**

15.17.1 Class Reference

```python
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
    Identifier for the geometric entity.
    It must be an integer number, otherwise the setter will raise a ValueError.
    Getter Gets the identifier
    Setter Sets the identifier
    Type int

    inside
    Inside-outside flag
    Getter Gets the flag
    Setter Sets the flag
    Type bool

    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

class geomdl.elements.Triangle(*args, **kwargs)
Bases: geomdl.elements.AbstractEntity
Triangle entity which represents a triangle composed of vertices.

add_vertex(*args)
   Adds vertices to the Triangle object.
   This method takes a single or a list of vertices as its function arguments.

edges
   Edges of the triangle
       Getter  Gets the list of vertices that generates the edges of the triangle
       Type    list

id
   Identifier for the geometric entity.
   It must be an integer number, otherwise the setter will raise a ValueError.
       Getter  Gets the identifier
       Setter  Sets the identifier
       Type    int
inside
   Inside-outside flag
    Getter  Gets the flag
    Setter  Sets the flag
    Type    bool

vertex_ids
   Vertex indices
   Vertex numbering starts from 1.
    Getter  Gets the vertex indices
    Type    list

vertex_ids_zero
   Zero-indexed vertex indices
   Vertex numbering starts from 0.
    Getter  Gets the vertex indices
    Type    list

vertices
   Vertices of the triangle
    Getter  Gets the list of vertices
    Type    tuple

vertices_raw
   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

vertices_uv
   Parametric coordinates of the triangle vertices
    Getter  Gets the parametric coordinates of the vertices
    Type    list

class geomdl.elements.Quad(*args, **kwargs)
   Bases: geomdl.elements.AbstractEntity
Quad entity which represents a quadrilateral structure composed of vertices.

add_vertex(*args)
   Adds vertices to the Quad object.
   This method takes a single or a list of vertices as its function arguments.

data
   Vertex indices.
    Getter  Gets the vertex indices
    Setter  Sets the vertex indices
id
Identifier for the geometric entity.

It must be an integer number, otherwise the setter will raise a `ValueError`.

   **Getter**  Gets the identifier
   **Setter**  Sets the identifier
   **Type**    int

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
Identifier for the geometric entity.

It must be an integer number, otherwise the setter will raise a `ValueError`.

   **Getter**  Gets the identifier
   **Setter**  Sets the identifier
   **Type**    int
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
Identifier for the geometric entity.

It must be an integer number, otherwise the setter will raise a `ValueError`.

   **Getter**  Gets the identifier
   **Setter**  Sets the identifier
   **Type**    int
15.18 Ray Module

The ray module provides utilities for ray operations. A ray (half-line) is defined by two distinct points represented by the `Ray` class. This module also provides a function to compute the intersection of 2 rays.

15.18.1 Function and Class Reference

class `geomdl.ray.Ray(point1, point2)`

Representation of a n-dimensional ray generated from 2 points.

A ray is defined by \( r(t) = p_1 + t \times \vec{d} \) where \( t \) is the parameter value, \( \vec{d} = p_2 - p_1 \) is the vector component of the ray, \( p_1 \) is the origin point and \( p_2 \) 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](https://example.com/wikipedia) 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](https://example.com/wikipedia) 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](https://example.com/wikipedia) 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](https://example.com/wikipedia) for details on using this class member.

**Getter** Gets the points

class `geomdl.ray.RayIntersection`

The status of the ray intersection operation

**COLINEAR = 2**
INTERSECT = 1
SKEW = 3

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 an 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 (t_ray1, t_ray2, RayIntersection.INTERSECT)
- Colinear: The rays can be parallel or coincident. Returns (-1, -1, RayIntersection.COLINEAR)
- Skew: The rays are neither parallel nor intersecting. Returns (t_ray1, t_ray2, RayIntersection.SKEW)

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 takes *The NURBS Book 2nd Edition by Piegl & Tiller* as the main reference for the evaluation algorithms. The users may want to use different algorithms and Evaluators serve directly to this purpose by allowing users to switch evaluation algorithms (i.e. evaluation strategy) at runtime. Please see evaluator property documentation for more details.
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:

### 16.1 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.

#### 16.1.1 Class Reference

Abstract base class for visualization

Defines an abstract base for NURBS-Python 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`

Visualization module internal configuration directives
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 (\textit{ctrlpts}) and the surface points (\textit{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 (\textit{ctrlpts}): points
- For evaluated points (\textit{evalpts}): points

\textbf{Surface}:

- For control points (\textit{ctrlpts}): points, quads, quadmesh
- For evaluated points (\textit{evalpts}): points, quads, triangles

\textbf{Volume}:

- For control points (\textit{ctrlpts}): points
- For evaluated points (\textit{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 visualization configuration.

\section{16.2 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.

\subsection{16.2.1 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): Axes 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: [10.67, 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

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.

`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()`.

```python
class geomdl.visualization.VisMPL.VisCurve2D(config=<geomdl.visualization.VisMPL.VisConfig object>)
```

**Bases:** `geomdl.vis.VisAbstract`

Matplotlib visualization module for 2D curves

**add**(`ptsarr`, `plot_type`, `name=", color="`)  
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` (`str`) – plot color

```python
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.

```python
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

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

Plots the 2D curve and the control points polygon.

```python
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.VisCurve3D(config=<geomdl.visualization.VisMPL.VisConfig object>)
```

**Bases:** `geomdl.vis.VisAbstract`

Matplotlib visualization module for 3D curves.

**add**(`ptsarr`, `plot_type`, `name=", color="`)  
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 (str)` – plot color

`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>)**

Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for surfaces.
Wireframe plot for the control points and scatter plot for the surface points.

`add (ptsarr, plot_type, name=", color=")`
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 (str)` – plot color
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

geomdl.visualization.VisMPL.VisSurfTriangle
alias of geomdl.visualization.VisMPL.VisSurface

class geomdl.visualization.VisMPL.VisSurfWireframe(config=<geomdl.visualization.VisMPL.VisConfig object>)
Bases: geomdl.vis.VisAbstract

Matplotlib visualization module for surfaces.
Scatter plot for the control points and wireframe plot for the surface points.

add(ptsarr, plot_type, name='', color='')
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(str) – plot color

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.VisMPL.VisSurface(config=<geomdl.visualization.VisMPL.VisConfig object>)

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=")  
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 (str) – plot color

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)

```python
clear()
```
Clears the points, colors and names lists.

```python
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`

```python
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)

```python
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`

```python
vconf
```
User configuration class for visualization

- **Getter** Gets the user configuration class
- **Type** `vis.VisConfigAbstract`

```python
class geomdl.visualization.VisMPL.VisVolume (config=<geomdl.visualization.VisMPL.VisConfig object>)
```
Matplotlib visualization module for volumes.

```python
add(ptsarr, plot_type, name=", color=")
```
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` *(str)* – plot color
**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>)*

**Bases:** `geomdl.vis.VisAbstract`

Matplotlib visualization module for voxel representation of the volumes.

**add** (**ptsarr, plot_type, name=", color="**)  
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** (str) – plot color

**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

### 16.3 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.

#### 16.3.1 Class Reference

**class**  geomdl.visualization.VisPlotly.VisConfig(**kwargs**)

**Bases:**  geomdl.vis.VisConfigAbstract

Configuration class for Plotly 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 VisPlotly 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
- **trims**  (bool): Trim curves visibility.  Default: True
- **axes_equal**  (bool): Enables or disables equal aspect ratio for the axes.  Default: True
• **line_width** (int): Thickness of the lines on the figure. Default: 2
• **figure_size** (list): Size of the figure in (x, y). Default: [800, 600]
• **trim_size** (int): Size of the trim curves. Default: 20

The following example illustrates the usage of the configuration class.

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

# Skipping degree, knot vector and control points assignments

# Create a visualization configuration instance with no legend, no axes and no control points grid
vis_config = VisPlotly.VisConfig(legend=False, axes=False, ctrlpts=False)

# Create a visualization method instance using the configuration above
vis_obj = VisPlotly.VisSurface(vis_config)

# Set the visualization method of the surface object
surf.vis = vis_obj

# Plot the surface
surf.render()
```

Please refer to the [Examples Repository](#) for more details.

```python
class geomdl.visualization.VisPlotly.VisCurve2D(config=<geomdl.visualization.VisPlotly.VisConfig object>)

Bases: geomdl.vis.VisAbstract
```

Plotly visualization module for 2D curves.

**add**(*ptsarr, plot_type, name=", color="*)

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** (str) – plot color

**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

```python
render(**kwargs)
```
Plots the curve and the control points polygon.

```python
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

```python
vconf
```
User configuration class for visualization

**Getter**
Gets the user configuration class

**Type**
vis.VisConfigAbstract

```python
class geomdl.visualization.VisPlotly.VisCurve3D(config=<geomdl.visualization.VisPlotly.VisConfig object>)
```
Bases: geomdl.vis.VisAbstract

Plotly visualization module for 3D curves.

```python
add(ptsarr, plot_type, name='', color='')
```
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** (str) – plot color

```python
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.

```python
clear()
```
Clears the points, colors and names lists.

```python
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

```python
render(**kwargs)
```
Plots the curve and the control points polygon.

```python
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.VisPlotly.VisSurface (config=<geomdl.visualization.VisPlotly.VisConfig object>)

Bases: geomdl.vis.VisAbstract

Plotly visualization module for surfaces.

Triangular mesh plot for the surface and wireframe plot for the control points grid.

add (ptsarr, plot_type, name=",", color=")

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 (str) – plot color

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
```python
class geomdl.visualization.VisPlotly.VisVolume(config=geomdl.visualization.VisPlotly.VisConfig object)

Bases: geomdl.vis.VisAbstract

Plotly visualization module for volumes.

add(ptsarr, plot_type, name=None, color=None)

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** (str) – plot color

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 evaluated 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
```

## 16.4 VTK Implementation

New in version 5.0.

This module provides VTK visualization implementation for NURBS-Python.
Note: Please make sure that you have installed \texttt{vtk} package before using this visualization module.

### 16.4.1 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 \texttt{VisVTK} module has the following configuration variables:

- `ctrlpts` (bool): Control points polygon/grid visibility. \textit{Default: True}
- `evalpts` (bool): Curve/surface points visibility. \textit{Default: True}
- `figure_size` (list): Size of the figure in (x, y). \textit{Default: (800, 600)}
- `line_width` (int): Thickness of the lines on the figure. \textit{Default: 0.5}

**keypress_callback**(\texttt{obj, ev})

VTK callback for keypress events.

Available custom keypress events:

- `b`: change background color

**class** `geomdl.visualization.VisVTK.VisCurve2D(config=<geomdl.visualization.VisVTK.VisConfig \texttt{object}>)`

Bases: `geomdl.vis.VisAbstract`

VTK visualization module for curves.

**add**\texttt{(ptsarr, plot\_type, name=\textquoteright , color=\textquoteright )}

Adds points sets to the visualization instance for plotting.

Parameters

- `ptsarr` (\texttt{list, tuple}) – control or evaluated points
- `plot\_type` (\texttt{str}) – type of the plot, e.g. \texttt{ctrlpts}, \texttt{evalpts}, \texttt{bbox}, etc.
- `name` (\texttt{str}) – name of the plot displayed on the legend
- `color` (\texttt{str}) – plot color

**animate**\texttt{(**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

geomdl.visualization.VisVTK.VisCurve3D
alias of geomdl.visualization.VisVTK.VisCurve2D

**class** geomdl.visualization.VisVTK.VisSurface (config=<geomdl.visualization.VisVTK.VisConfig object>)

**Bases:** geomdl.vis.VisAbstract

VTK visualization module for surfaces.

**add** (*ptsarr, plot_type*, *name=", color="*)
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** (*str*) – plot color

**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>)

Bases: geomdl.vis.VisAbstract

VTK visualization module for volumes.

`add (ptsarr, plot_type, name=", color=")`

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 (str)` – plot color

`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

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.

### 17.1 Installation

The easiest method to install is via `pip`. It will install all the required modules.

```
$ pip install geomdl-cli
```

Please refer to `geomdl-cli` documentation for more installation options.

### 17.2 Documentation

`geomdl-cli` has a very detailed online documentation which describes the usage and customization options of the command-line application.

### 17.3 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.

### 18.1 Installation

The easiest method to install is via pip.

```
$ pip install geomdl.shapes
```

Please refer to geomdl-shapes documentation for more installation options.

### 18.2 Documentation

You can find the class and function references in the geomdl-shapes documentation.

### 18.3 References

- PyPI: https://pypi.org/project/geomdl.shapes
- Documentation: https://geomdl-shapes.readthedocs.io
- Development: https://github.com/orbingol/geomdl-shapes
The **Rhino importer/exporter** module `rw3dm` is a `pybind11`-wrapped Python package for OpenNURBS, only focused on reading and writing `.3dm` files. `exchange.import_3dm()` function uses `rw3dm` module to extract curve and surface data from `.3dm` files. Similarly, `exchange.export_3dm()` function uses `rw3dm` module to save NURBS data as `.3dm` files.

### 19.1 Installation

Please refer to the `rw3dm` repository for installation options.

### 19.2 References

- **Development**: https://github.com/orbingol/rw3dm
<table>
<thead>
<tr>
<th>Module</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compatibility</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>construct</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>convert</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>elements</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>exchange</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>exchange_vtk</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>geomdl.compatibility</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.construct</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.convert</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.elements</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.exchange</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.exchange_vtk</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.fitting</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.helpers</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.knotvector</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.linalg</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.operations</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.ray</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.utilities</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.vis.VisAbstract</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.vis.VisConfigAbstract</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.visualization.VisMPL</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.visualization.VisPlotly</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.visualization.VisVTK</code></td>
<td></td>
</tr>
<tr>
<td><code>geomdl.voxelize</code></td>
<td></td>
</tr>
<tr>
<td><code>interpolate</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>knotvector</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>linalg</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>operations</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>ray</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>utilities</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>VisMPL</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>VisPlotly</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>VisVTK</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
<tr>
<td><code>voxelize</code></td>
<td><code>Unix</code>, <code>Windows</code></td>
</tr>
</tbody>
</table>
AbstractContainer (class in geomdl.multi), 184
AbstractEvaluator (class in geomdl.evaluators), 154
AbstractTessellate (class in geomdl.tessellate), 210
add() (geomdl.multi.AbstractContainer method), 184
add() (geomdl.multi.CurveContainer method), 186
add() (geomdl.multi.SurfaceContainer method), 189
add() (geomdl.multi.VolumeContainer method), 192
add() (geomdl.visualization.VisMPL.VisCurve2D method), 224
add() (geomdl.visualization.VisMPL.VisCurve3D method), 224
add() (geomdl.visualization.VisMPL.VisSurface method), 227
add() (geomdl.visualization.VisMPL.VisSurfScatter method), 225
add() (geomdl.visualization.VisMPL.VisSurfWireframe method), 226
add() (geomdl.visualization.VisMPL.VisVolume method), 228
add() (geomdl.visualization.VisMPL.VisVoxel method), 229
add() (geomdl.visualization.VisPlotly.VisCurve2D method), 231
add() (geomdl.visualization.VisPlotly.VisCurve3D method), 232
add() (geomdl.visualization.VisPlotly.VisSurface method), 233
add() (geomdl.visualization.VisPlotly.VisVolume method), 234
add() (geomdl.visualization.VisVTK.VisCurve2D method), 235
add() (geomdl.visualization.VisVTK.VisCurve3D method), 236
add() (geomdl.visualization.VisVTK.VisSurface method), 237
add() (geomdl.visualization.VisVTK.VisVolume method), 237
add_face() (geomdl.elements.Body method), 217
add_triangle() (geomdl.elements.Face method), 217
add_vertex() (geomdl.elements.Quad method), 216
add_vertex() (geomdl.elements.Triangle method), 215
animate() (geomdl.visualization.VisMPL.VisCurve2D method), 224
animate() (geomdl.visualization.VisMPL.VisCurve3D method), 225
animate() (geomdl.visualization.VisMPL.VisSurface method), 227
animate() (geomdl.visualization.VisMPL.VisSurfScatter method), 226
animate() (geomdl.visualization.VisMPL.VisSurfWireframe method), 226
animate() (geomdl.visualization.VisMPL.VisVolume method), 228
animate() (geomdl.visualization.VisMPL.VisVoxel method), 229
animate() (geomdl.visualization.VisPlotly.VisCurve2D method), 231
animate() (geomdl.visualization.VisPlotly.VisCurve3D method), 232
animate() (geomdl.visualization.VisPlotly.VisSurface method), 233
animate() (geomdl.visualization.VisPlotly.VisVolume method), 234
animate() (geomdl.visualization.VisVTK.VisCurve2D method), 235
animate() (geomdl.visualization.VisVTK.VisCurve3D method), 236
animate() (geomdl.visualization.VisVTK.VisSurface method), 237
append() (geomdl.multi.AbstractContainer method), 184
append() (geomdl.multi.CurveContainer method), 186
append() (geomdl.multi.SurfaceContainer method), 189
append() (geomdl.visualization.VisMPL.VisCurve2D method), 224
append() (geomdl.visualization.VisMPL.VisCurve3D method), 224
append() (geomdl.visualization.VisMPL.VisSurface method), 227
append() (geomdl.visualization.VisMPL.VisSurfScatter method), 226
append() (geomdl.visualization.VisMPL.VisSurfWireframe method), 226
append() (geomdl.visualization.VisMPL.VisVolume method), 228
append() (geomdl.visualization.VisMPL.VisVoxel method), 229
append() (geomdl.visualization.VisPlotly.VisCurve2D method), 231
append() (geomdl.visualization.VisPlotly.VisCurve3D method), 232
append() (geomdl.visualization.VisPlotly.VisSurface method), 233
append() (geomdl.visualization.VisPlotly.VisVolume method), 234
append() (geomdl.visualization.VisVTK.VisCurve2D method), 235
append() (geomdl.visualization.VisVTK.VisCurve3D method), 236
append() (geomdl.visualization.VisVTK.VisSurface method), 237
append() (geomdl.visualization.VisVTK.VisVolume method), 237
add() (geomdl.operations.add_dimension() method), 161
add() (geomdl.visualization.VisMPL.VisCurve2D method), 224
add() (geomdl.visualization.VisMPL.VisCurve3D method), 224
add() (geomdl.visualization.VisMPL.VisSurface method), 227
add() (geomdl.visualization.VisMPL.VisSurfScatter method), 225
add() (geomdl.visualization.VisMPL.VisSurfWireframe method), 226
add() (geomdl.visualization.VisMPL.VisVolume method), 228
add() (geomdl.visualization.VisMPL.VisVoxel method), 229
add() (geomdl.visualization.VisPlotly.VisCurve2D method), 231
add() (geomdl.visualization.VisPlotly.VisCurve3D method), 232
add() (geomdl.visualization.VisPlotly.VisSurface method), 233
add() (geomdl.visualization.VisPlotly.VisVolume method), 234
add() (geomdl.visualization.VisVTK.VisCurve2D method), 235
add() (geomdl.visualization.VisVTK.VisCurve3D method), 236
add() (geomdl.visualization.VisVTK.VisSurface method), 237
add() (geomdl.visualization.VisVTK.VisVolume method), 237
add() (in module geomdl.operations), 247
append() (geomdl.multi.VolumeContainer method), 193
approximate_curve() (in module geomdl.fitting), 200
approximate_surface() (in module geomdl.fitting), 200
arguments (geomdl.tessellate.AbstractTessellate attribute), 210
arguments (geomdl.tessellate.TriangularTessellate attribute), 211
arguments (geomdl.tessellate.TrimTessellate attribute), 212
backward_substitution() (in module geomdl.linalg), 175
basis_function() (in module geomdl.helpers), 168
basis_function_all() (in module geomdl.helpers), 168
basis_function_ders() (in module geomdl.helpers), 168
basis_function_ders_one() (in module geomdl.helpers), 169
basis_function_one() (in module geomdl.helpers), 169
basis_functions() (in module geomdl.helpers), 169
basis_functions_ders() (in module geomdl.helpers), 170
bbox (geomdl.abstract.Curve attribute), 66
bbox (geomdl.abstract.SplineGeometry attribute), 90
bbox (geomdl.abstract.Surface attribute), 72
bbox (geomdl.abstract.Volume attribute), 80
bbox (geomdl.BSpline.Curve attribute), 94
bbox (geomdl.BSpline.Surface attribute), 102
bbox (geomdl.BSpline.Volume attribute), 113
bbox (geomdl.multi.AbstractContainer attribute), 184
bbox (geomdl.multi.CurveContainer attribute), 186
bbox (geomdl.multi.VolumeContainer attribute), 189
bbox (geomdl.NURBS.Curve attribute), 124
bbox (geomdl.NURBS.Surface attribute), 132
bbox (geomdl.NURBS.Volume attribute), 144
binomial_coefficient (in module geomdl.linalg), 175
binormal() (geomdl.BSpline.Curve method), 94
binormal() (geomdl.NURBS.Curve method), 124
binormal() (in module geomdl.operations), 164
Body (class in geomdl.elements), 217
bspline_to_nurbs() (in module geomdl.convert), 180
bumps() (geomdl.CPGen.Grid method), 201
bumps() (geomdl.CPGen.GridWeighted method), 202
ctrlpts2d (geomdl.BSpline.Surface attribute), 102
ctrlpts2d (geomdl.NURBS.Surface attribute), 133
ctrlpts_offset
geomdl.visualization.VisMPL.VisCurve2D attribute), 224
ctrlpts_offset
geomdl.visualization.VisMPL.VisCurve3D attribute), 225
ctrlpts_offset
geomdl.visualization.VisMPL.VisSurface attribute), 228
ctrlpts_offset
geomdl.visualization.VisMPL.VisSurfScatter attribute), 226
ctrlpts_offset
geomdl.visualization.VisMPL.VisSurfWireframe attribute), 227
ctrlpts_offset
geomdl.visualization.VisMPL.VisVolume attribute), 229
ctrlpts_offset
geomdl.visualization.VisPlotly.VisCurve2D attribute), 231
ctrlpts_offset
geomdl.visualization.VisPlotly.VisCurve3D attribute), 232
ctrlpts_offset
geomdl.visualization.VisPlotly.VisSurface attribute), 233
ctrlpts_offset
geomdl.visualization.VisPlotly.VisSurfScatter attribute), 234
ctrlpts_offset
geomdl.visualization.VisPlotly.VisSurfWireframe attribute), 235
ctrlpts_offset
geomdl.visualization.VisVTK.VisCurve2D attribute), 236
ctrlpts_offset
geomdl.visualization.VisVTK.VisSurface attribute), 237
ctrlpts_offset (in module geomdl.vis.VisAbstract), 221
ctrlpts_size (geomdl.abstract.Curve attribute), 67
ctrlpts_size (geomdl.abstract.SplineGeometry attribute), 90
ctrlpts_size (geomdl.abstract.Surface attribute), 72
ctrlpts_size (geomdl.abstract.Volume attribute), 81
ctrlpts_size (geomdl.BSpline.Curve attribute), 95
ctrlpts_size (geomdl.BSpline.Surface attribute), 103
data (geomdl.BSpline.Curve attribute), 95
data (geomdl.BSpline.Surface attribute), 104
data (geomdl.BSpline.Volume attribute), 114
data (geomdl.elements.Quad attribute), 216
data (geomdl.elements.Vertex attribute), 214
data (geomdl.NURBS.Curve attribute), 125
data (geomdl.NURBS.Surface attribute), 134
data (geomdl.NURBS.Volume attribute), 145
decompose_curve () (in module geomdl.operations), 161
decompose_surface () (in module geomdl.operations), 162
degree (geomdl.abstract.Curve attribute), 67
degree (geomdl.abstract.SplineGeometry attribute), 90
degree (geomdl.abstract.Surface attribute), 73
degree (geomdl.abstract.Volume attribute), 81
degree (geomdl.BSpline.Curve attribute), 95
degree (geomdl.BSpline.Surface attribute), 104
degree (geomdl.BSpline.Volume attribute), 114
degree (geomdl.NURBS.Curve attribute), 125
degree (geomdl.NURBS.Surface attribute), 134
degree (geomdl.NURBS.Volume attribute), 145
degree_elevation () (in module geomdl.helpers), 170
degree_reduction () (in module geomdl.helpers), 170
degree_u (geomdl.abstract.Surface attribute), 73
degree_u (geomdl.abstract.Volume attribute), 81
degree_u (geomdl.BSpline.Surface attribute), 104
degree_u (geomdl.BSpline.Volume attribute), 114
degree_u (geomdl.NURBS.Surface attribute), 134
degree_u (geomdl.NURBS.Volume attribute), 145
degree_v (geomdl.abstract.Surface attribute), 73
degree_v (geomdl.abstract.Volume attribute), 82
degree_v (geomdl.BSpline.Surface attribute), 104
degree_v (geomdl.BSpline.Volume attribute), 115
degree_v (geomdl.NURBS.Surface attribute), 134
degree_v (geomdl.NURBS.Volume attribute), 145
degree_w (geomdl.abstract.Volume attribute), 82
degree_w (geomdl.BSpline.Volume attribute), 115
degree_w (geomdl.NURBS.Volume attribute), 145
delta (geomdl.abstract.Curve attribute), 67
delta (geomdl.abstract.Surface attribute), 73
delta (geomdl.abstract.Volume attribute), 82
delta (geomdl.BSpline.Curve attribute), 95
delta (geomdl.BSpline.Surface attribute), 104
delta (geomdl.BSpline.Volume attribute), 115
delta (geomdl.multi.AbstractContainer attribute), 184
delta (geomdl.multi.CurveContainer attribute), 187
delta (geomdl.multi.SurfaceContainer attribute), 189
delta (geomdl.multi.VolumeContainer attribute), 193
delta (geomdl.NURBS.Curve attribute), 125
delta (geomdl.NURBS.Surface attribute), 134
delta (geomdl.NURBS.Volume attribute), 146
delta_u (geomdl.abstract.Surface attribute), 74
delta_u (geomdl.abstract.Volume attribute), 82
delta_u (geomdl.BSpline.Curve attribute), 104
delta_u (geomdl.BSpline.Volume attribute), 115
delta_u (geomdl.multi.SurfaceContainer attribute), 189
delta_u (geomdl.multi.VolumeContainer attribute), 193
delta_u (geomdl.NURBS.Surface attribute), 135
delta_u (geomdl.NURBS.Volume attribute), 146
delta_v (geomdl.abstract.Surface attribute), 74
delta_v (geomdl.abstract.Volume attribute), 83
delta_v (geomdl.BSpline.Surface attribute), 105
delta_v (geomdl.BSpline.Volume attribute), 115
delta_v (geomdl.multi.SurfaceContainer attribute), 190
delta_v (geomdl.multi.VolumeContainer attribute), 193
delta_w (geomdl.NURBS.Surface attribute), 135
delta_w (geomdl.NURBS.Volume attribute), 146
delta_w (geomdl.BSpline.Volume attribute), 116
delta_w (geomdl.multi.VolumeContainer attribute), 193
derivative_curve () (in module geomdl.operations), 147
derivative_surface () (in module geomdl.operations), 161
derivatives () (geomdl.abstract.Curve method), 67
derivatives () (geomdl.abstract.Surface method), 74
derivatives () (geomdl.BSpline.Curve method), 95
derivatives () (geomdl.BSpline.Surface method), 105
derivatives () (geomdl.evaluators.AbstractEvaluator method), 154
derivatives () (geomdl.evaluators.CurveEvaluator method), 155
derivatives () (geomdl.evaluators.CurveEvaluator2 method), 155
derivatives () (geomdl.evaluators.CurveEvaluatorRational method), 156
derivatives () (geomdl.evaluators.SurfaceEvaluator method), 156
derivatives () (geomdl.evaluators.SurfaceEvaluator2 method), 157
derivatives () (geomdl.evaluators.SurfaceEvaluatorRational method), 157
derivatives() (geomdl.evaluators.VolumeEvaluator method), 157
derivatives() (geomdl.evaluators.VolumeEvaluatorRational method), 158
derivatives() (geomdl.NURBS.Curve method), 125
derivatives() (geomdl.NURBS.Surface method), 135
derivatives_ctrlpts() (geomdl.evaluators.CurveEvaluator2 static method), 155
derivatives_ctrlpts() (geomdl.evaluators.SurfaceEvaluator2 static method), 157
dimension (geomdl.abstract.Curve attribute), 67
dimension (geomdl.abstract.SplineGeometry attribute), 90
dimension (geomdl.abstract.Surface attribute), 74
dimension (geomdl.abstract.Volume attribute), 83
dimension (geomdl.BSpline.Curve attribute), 95
dimension (geomdl.BSpline.Surface attribute), 105
dimension (geomdl.BSpline.Volume attribute), 116
dimension (geomdl.multi.AbstractContainer attribute), 185
dimension (geomdl.multi.CurveContainer attribute), 187
dimension (geomdl.multi.SurfaceContainer attribute), 190
dimension (geomdl.multi.VolumeContainer attribute), 194
dimension (geomdl.NURBS.Curve attribute), 125
dimension (geomdl.NURBS.Surface attribute), 136
dimension (geomdl.NURBS.Volume attribute), 147
dimension (geomdl.BSpline.Surface attribute), 88
domain (geomdl.abstract.Curve attribute), 68
d domain (geomdl.abstract.SplineGeometry attribute), 90
domain (geomdl.abstract.Volume attribute), 83
domain (geomdl.BSpline.Curve attribute), 96
domain (geomdl.BSpline.Surface attribute), 106
domain (geomdl.BSpline.Volume attribute), 116
domain (geomdl.NURBS.Curve attribute), 126
domain (geomdl.NURBS.Surface attribute), 136
domain (geomdl.NURBS.Volume attribute), 147

e
edges (geomdl.elements.Triangle attribute), 215
elements (module), 214
eval() (geomdl.BSpline.Curve method), 218
evalpts (geomdl.BSpline.Curve attribute), 68
evalpts (geomdl.abstract.Curve attribute), 69
evalpts (geomdl.abstract.Geometry attribute), 89
evalpts (geomdl.abstract.SplineGeometry attribute), 90
evalpts (geomdl.abstract.Surface attribute), 75
evalpts (geomdl.abstract.Volume attribute), 83
evalpts (geomdl.BSpline.Curve attribute), 96
evalpts (geomdl.BSpline.Surface attribute), 106
evalpts (geomdl.BSpline.Volume attribute), 116
evalpts (geomdl.multi.AbstractContainer attribute), 185
evalpts (geomdl.multi.CurveContainer attribute), 187
evalpts (geomdl.multi.SurfaceContainer attribute), 190
evalpts (geomdl.multi.VolumeContainer attribute), 194
evalpts (geomdl.NURBS.Curve attribute), 126
evalpts (geomdl.NURBS.Surface attribute), 136
evalpts (geomdl.NURBS.Volume attribute), 147
evaluate() (geomdl.abstract.Curve method), 68
evaluate() (geomdl.abstract.Surface method), 75
evaluate() (geomdl.abstract.Volume method), 84
evaluate_list() (geomdl.BSpline.Curve method), 96

Index 251
evaluate_list (geomdl.BSpline.Surface method), 106
evaluate_list (geomdl.BSpline.Volume method), 117
evaluate_list (geomdl.NURBS.Curve method), 126
evaluate_list (geomdl.NURBS.Surface method), 137
evaluate_list (geomdl.NURBS.Volume method), 147
evaluate_single (geomdl.abstract.Curve method), 68
evaluate_single (geomdl.abstract.Surface method), 75
evaluate_single (geomdl.abstract.Volume method), 84
evaluate_single (geomdl.BSpline.Curve method), 96
evaluate_single (geomdl.BSpline.Surface method), 106
evaluate_single (geomdl.BSpline.Volume method), 117
evaluate_single (geomdl.NURBS.Curve method), 126
evaluate_single (geomdl.NURBS.Surface method), 137
evaluate_single (geomdl.NURBS.Volume method), 148
evaluator (geomdl.abstract.Curve attribute), 68
evaluator (geomdl.abstract.SplineGeometry attribute), 91
evaluator (geomdl.abstract.Surface attribute), 75
evaluator (geomdl.abstract.Volume attribute), 84
evaluator (geomdl.BSpline.Curve attribute), 96
evaluator (geomdl.BSpline.Surface attribute), 106
evaluator (geomdl.BSpline.Volume attribute), 117
evaluator (geomdl.NURBS.Curve attribute), 126
evaluator (geomdl.NURBS.Surface attribute), 137
evaluator (geomdl.NURBS.Volume attribute), 148
exchange (module), 204
exchange_vtk (module), 210
export_3dm () (in module geomdl.exchange), 209
export_cfg () (in module geomdl.exchange), 206
export_csv () (in module geomdl.exchange), 205
export_json () (in module geomdl.exchange), 210
export_obj () (in module geomdl.exchange), 208
export_solid () (in module geomdl.exchange), 208
export_txt () (in module geomdl.exchange), 208
export_vmesh () (in module geomdl.exchange), 205
export_yaml () (in module geomdl.exchange), 206
extract_curves () (in module geomdl.construct), 181
extract_isosurface () (in module geomdl.construct), 182
extract_surfaces () (in module geomdl.construct), 182
F
Face (class in geomdl.elements), 217
test (geomdl.elements.Body attribute), 217
test (geomdl.tessellate.AbstractTessellate attribute), 211
test (geomdl.tessellate.TriangularTessellate attribute), 211
test (geomdl.tessellate.TrimTessellate attribute), 212
find_cripts () (in module geomdl.operations), 164
find_multiplicity () (in module geomdl.helpers), 170
find_span_binsearch () (in module geomdl.helpers), 171
find_span_linear () (in module geomdl.helpers), 171
find_spans () (in module geomdl.helpers), 171
flip_cripts () (in module geomdl.compatibility), 196
flip_cripts2d () (in module geomdl.compatibility), 196
flip_cripts2d_file () (in module geomdl.compatibility), 197
flip_cripts_u () (in module geomdl.compatibility), 197
forward_substitution () (in module geomdl.linalg), 175
frange () (in module geomdl.linalg), 175
G
generate (geomdl.CPGen.Grid method), 201
generate (geomdl.CPGen.GridWeighted method), 202
generate () (in module geomdl.knotvector), 179
generate_cripts2d_weights () (in module geomdl.compatibility), 197
generate_cripts2d_weights_file () (in module geomdl.compatibility), 197
generate_cripts_weights () (in module geomdl.compatibility), 198
generate_cripts2w () (in module geomdl.compatibility), 198
generate_cripts2w2d () (in module geomdl.compatibility), 198
generate_cripts2w2d_file () (in module geomdl.compatibility), 198
geomdl.compatibility (module), 196
geomdl.construct (module), 181
make_quad() (geomdl.NURBS.Surface attribute), 137
knotvector_u (geomdl.NURBS.Volume attribute), 148
knotvector_u (geomdl.NURBS.Volume attribute), 137
knotvector_v (geomdl.abstract.Surface attribute), 76
knotvector_v (geomdl.abstract.Volume attribute), 84
knotvector_v (geomdl.BSpline.Surface attribute), 107
knotvector_v (geomdl.BSpline.Volume attribute), 118
knotvector_v (geomdl.NURBS.Surface attribute), 138
knotvector_v (geomdl.NURBS.Volume attribute), 149
knotvector_w (geomdl.abstract.Volume attribute), 85
knotvector_w (geomdl.BSpline.Volume attribute), 118
knotvector_w (geomdl.NURBS.Volume attribute), 149

length_curve() (in module geomdl.operations), 161
linalg (module), 175
linspace() (in module geomdl.linalg), 176
load() (geomdl.BSpline.Curve method), 97
load() (geomdl.BSpline.Surface method), 107
load() (geomdl.BSpline.Volume method), 118
load() (geomdl.NURBS.Curve method), 127
load() (geomdl.NURBS.Surface method), 138
load() (geomdl.NURBS.Volume method), 149
lu_decomposition() (in module geomdl.linalg), 176

make_quad() (in module geomdl.utilities), 165
make_quadtree() (in module geomdl.utilities), 166
make_triangle_mesh() (in module geomdl.utilities), 166
make_zigzag() (in module geomdl.utilities), 167
matrix_multiply() (in module geomdl.linalg), 176
matrix_transpose() (in module geomdl.linalg), 176
mconf (in module geomdl.vis.VisAbstract), 221

name (geomdl.abstract.Curve attribute), 69
name (geomdl.abstract.Geometry attribute), 89
name (geomdl.abstract.SplineGeometry attribute), 91
name (geomdl.abstract.Surface attribute), 76
name (geomdl.abstract.Volume attribute), 85
name (geomdl.BSpline.Curve attribute), 97
name (geomdl.BSpline.Surface attribute), 108
name (geomdl.BSpline.Volume attribute), 118
name (geomdl.evaluators.AbstractEvaluator attribute), 154
name (geomdl.evaluators.CurveEvaluator attribute), 155
name (geomdl.evaluators.CurveEvaluator2 attribute), 155
name (geomdl.evaluators.CurveEvaluatorRational attribute), 156
name (geomdl.evaluators.SurfaceEvaluator attribute), 156
name (geomdl.evaluators.SurfaceEvaluator2 attribute), 157
name (geomdl.evaluators.SurfaceEvaluatorRational attribute), 157
name (geomdl.evaluators.VolumeEvaluator attribute), 158
name (geomdl.NURBS.Curve attribute), 127
name (geomdl.NURBS.Surface attribute), 138
name (geomdl.NURBS.Volume attribute), 149
next() (geomdl.BSpline.Curve method), 97
next() (geomdl.BSpline.Surface method), 108
next() (geomdl.BSpline.Volume method), 118
next() (geomdl.NURBS.Curve method), 127
next() (geomdl.NURBS.Surface method), 138
next() (geomdl.NURBS.Volume method), 149
normal() (geomdl.BSpline.Curve method), 97
normal() (geomdl.BSpline.Surface method), 108
normal() (geomdl.BSpline.Volume method), 118
normal() (geomdl.NURBS.Curve method), 127
normal() (geomdl.NURBS.Surface method), 138
normal() (geomdl.NURBS.Volume method), 149
normalize() (in module geomdl.operations), 163
nurbs_to_bspline() (in module geomdl.convert), 180

operations (module), 158
order (geomdl.abstract.Curve attribute), 69
order (geomdl.BSpline.Curve attribute), 98
order (geomdl.NURBS.Curve attribute), 128
order_u (geomdl.abstract.Surface attribute), 76
order_u (geomdl.abstract.Volume attribute), 85
order_u (geomdl.BSpline.Surface attribute), 108
order_u (geomdl.BSpline.Volume attribute), 118
order_u (geomdl.NURBS.Surface attribute), 138
order_u (geomdl.NURBS.Volume attribute), 149
order_v (geomdl.abstract.Surface attribute), 76
order_v (geomdl.abstract.Volume attribute), 85
order_v (geomdl.BSpline.Surface attribute), 108
order_v (geomdl.BSpline.Volume attribute), 119
order_v (geomdl.NURBS.Surface attribute), 139
order_v (geomdl.NURBS.Volume attribute), 149
order_w (geomdl.abstract.Volume attribute), 85
order_w (geomdl.BSpline.Volume attribute), 119
order_w (geomdl.NURBS.Volume attribute), 150
P
p (geomdl.ray.Ray attribute), 218
pdimension (geomdl.abstract.Curve attribute), 69
pdimension (geomdl.abstract.CurveGeometry attribute), 91
pdimension (geomdl.abstract.Surface attribute), 77
pdimension (geomdl.abstract.Volume attribute), 86
pdimension (geomdl.BSpline.Curve attribute), 98
pdimension (geomdl.BSpline.Surface attribute), 108
pdimension (geomdl.BSpline.Volume attribute), 119
pdimension (geomdl.NURBS.Curve attribute), 128
pdimension (geomdl.NURBS.Surface attribute), 139
pdimension (geomdl.NURBS.Volume attribute), 150
point_distance() (in module geomdl.linalg), 176
point_mid() (in module geomdl.linalg), 177
point_translate() (in module geomdl.linalg), 177
points (geomdl.ray.Ray attribute), 218
polygon_triangulate() (in module geomdl.utilities), 167
Q
Quad (class in geomdl.elements), 216
R
range (geomdl.abstract.Curve attribute), 69
range (geomdl.abstract.CurveGeometry attribute), 91
range (geomdl.abstract.Surface attribute), 77
range (geomdl.abstract.Volume attribute), 86
range (geomdl.BSpline.Curve attribute), 98
range (geomdl.BSpline.Surface attribute), 109
range (geomdl.BSpline.Volume attribute), 119
range (geomdl.NURBS.Curve attribute), 128
range (geomdl.NURBS.Surface attribute), 139
range (geomdl.NURBS.Volume attribute), 150
rational (geomdl.abstract.Curve attribute), 69
rational (geomdl.abstract.CurveGeometry attribute), 91
rational (geomdl.abstract.Surface attribute), 77
rational (geomdl.abstract.Volume attribute), 86
rational (geomdl.BSpline.Curve attribute), 98
rational (geomdl.BSpline.Surface attribute), 109
rational (geomdl.BSpline.Volume attribute), 119
rational (geomdl.NURBS.Curve attribute), 128
rational (geomdl.NURBS.Surface attribute), 139
rational (geomdl.NURBS.Volume attribute), 150
Ray (class in geomdl.ray), 218
Ray (module), 218
RayIntersection (class in geomdl.ray), 218
refine_knotvector() (in module geomdl.operations), 159
remove_knot() (geomdl.BSpline.Curve method), 98
remove_knot() (geomdl.BSpline.Surface method), 109
remove_knot() (geomdl.BSpline.Volume method), 119
remove_knot() (geomdl.NURBS.Curve method), 128
remove_knot() (geomdl.NURBS.Surface method), 139
remove_knot() (geomdl.NURBS.Volume method), 149
remove_knot() (in module geomdl.operations), 159
render() (geomdl.abstract.Curve method), 69
render() (geomdl.abstract.CurveGeometry method), 92
render() (geomdl.abstract.Surface method), 77
render() (geomdl.abstract.Volume method), 86
render() (geomdl.BSpline.Curve method), 98
render() (geomdl.BSpline.Surface method), 109
render() (geomdl.BSpline.Volume method), 120
render() (geomdl.multi.AbstractContainer method), 185
render() (geomdl.multi.CurveContainer method), 187
render() (geomdl.multi.SurfaceContainer method), 190
render() (geomdl.multi.VolumeContainer method), 194
render() (geomdl.NURBS.Curve method), 128
render() (geomdl.NURBS.Surface method), 139
render() (geomdl.NURBS.Volume method), 150
render() (geomdl.visualization.VisMPL.VisCurve2D method), 224
render() (geomdl.visualization.VisMPL.VisCurve3D method), 225
render() (geomdl.visualization.VisMPL.VisSurface method), 228
render() (geomdl.visualization.VisMPL.VisSurfScatter method), 226
render() (geomdl.visualization.VisMPL.VisSurfWireframe method), 227
render() (geomdl.visualization.VisMPL.VisVolume method), 229
render() (geomdl.visualization.VisMPL.VisVoxel method), 230
render() (geomdl.visualization.VisPlotly.VisCurve2D method), 232
render() (geomdl.visualization.VisPlotly.VisCurve3D method), 232
render() (geomdl.visualization.VisPlotly.VisSurface method), 233
render() (geomdl.visualization.VisPlotly.VisVolume method), 234
render() (geomdl.visualization.VisVTK.VisCurve2D method), 236
render() (geomdl.visualization.VisVTK.VisSurface method), 236

Index
render() (geomdl.visualization.VisVTK.VisVolume method), 237
reset() (geomdl.abstract.Curve method), 70
reset() (geomdl.abstract.Spline geometry), 78
reset() (geomdl.abstract.Volume method), 87
reset() (geomdl.BSpline.Curve method), 99
reset() (geomdl.BSpline.Surface method), 110
reset() (geomdl.BSpline.Volume method), 120
reset() (geomdl.CPGen.Grid method), 202
reset() (geomdl.CPGen.GridWeighted method), 203
reset() (geomdl.NURBS.Curve method), 129
reset() (geomdl.NURBS.Surface method), 140
reset() (geomdl.NURBS.Volume method), 151
reset() (geomdl.tessellate.AbstractTessellate method), 211
reset() (geomdl.tessellate.TriangularTessellate method), 212
reset() (geomdl.tessellate.TrimTessellate method), 212
rotate() (in module geomdl.operations), 164

S
sample_size (geomdl.abstract.Curve attribute), 70
sample_size (geomdl.abstract.Surface attribute), 78
sample_size (geomdl.abstract.Volume attribute), 87
sample_size (geomdl.BSpline.Curve attribute), 99
sample_size (geomdl.BSpline.Surface attribute), 110
sample_size (geomdl.BSpline.Volume attribute), 121
sample_size (geomdl.multi.AbstractContainer attribute), 185
sample_size (geomdl.multi.CurveContainer attribute), 188
sample_size (geomdl.multi.SurfaceContainer attribute), 191
sample_size (geomdl.multi.VolumeContainer attribute), 195
sample_size (geomdl.NURBS.Curve attribute), 129
sample_size (geomdl.NURBS.Surface attribute), 140
sample_size (geomdl.NURBS.Volume attribute), 151
sample_size_u (geomdl.abstract.Surface attribute), 78
sample_size_u (geomdl.BSpline.Volume attribute), 87
sample_size_u (geomdl.BSpline.Surface attribute), 110
sample_size_u (geomdl.BSpline.Volume attribute), 121
sample_size_u (geomdl.multi.SurfaceContainer attribute), 191
sample_size_u (geomdl.multi.VolumeContainer attribute), 195
sample_size_u (geomdl.NURBS.Surface attribute), 141
sample_size_v (geomdl.abstract.Surface attribute), 78
sample_size_v (geomdl.abstract.Volume attribute), 87
sample_size_v (geomdl.BSpline.Surface attribute), 111
sample_size_v (geomdl.BSpline.Volume attribute), 121
sample_size_v (geomdl.multi.SurfaceContainer attribute), 191
sample_size_v (geomdl.multi.VolumeContainer attribute), 195
sample_size_v (geomdl.NURBS.Surface attribute), 141
sample_size_v (geomdl.NURBS.Volume attribute), 152
sample_size_w (geomdl.abstract.Surface attribute), 88
sample_size_w (geomdl.BSpline.Volume attribute), 121
sample_size_w (geomdl.multi.SurfaceContainer attribute), 195
sample_size_w (geomdl.NURBS.Volume attribute), 152
save() (geomdl.BSpline.Curve method), 100
save() (geomdl.BSpline.Surface method), 111
save() (geomdl.BSpline.Volume method), 121
save() (geomdl.NURBS.Curve method), 130
save() (geomdl.NURBS.Surface method), 141
save() (geomdl.NURBS.Volume method), 152
save_figure_as() (geomdl.visualization.VisMPL.VisConfig static method), 223
save_voxel_grid() (in module geomdl.voxelize), 213
scale() (in module geomdl.visualization), 164
separate_ctrlpts_weights() (in module geomdl.compatibility), 199
set_axes_equal() (geomdl.visualization.VisMPL.VisConfig static method), 223
set_ctrlpts() (geomdl.abstract.Curve method), 71
set_ctrlpts() (geomdl.abstract.SplineGeometry method), 92
set_ctrlpts() (geomdl.abstract.Surface method), 79
set_ctrlpts() (geomdl.abstract.Volume method), 88
set_ctrlpts() (geomdl.BSpline.Curve method), 100
set_ctrlpts() (geomdl.BSpline.Surface method), 111
set_ctrlpts() (geomdl.BSpline.Volume method), 122
weights (geomdl.abstract.SplineGeometry attribute), 92
weights (geomdl.abstract.Surface attribute), 79
weights (geomdl.abstract.Volume attribute), 88
weights (geomdl.BSpline.Curve attribute), 100
weights (geomdl.BSpline.Surface attribute), 112
weights (geomdl.BSpline.Volume attribute), 122
weights (geomdl.NURBS.Curve attribute), 130
weights (geomdl.NURBS.Surface attribute), 142
weights (geomdl.NURBS.Volume attribute), 153

X
x (geomdl.elements.Vertex attribute), 215

Y
y (geomdl.elements.Vertex attribute), 215

Z
z (geomdl.elements.Vertex attribute), 215