Introduction

1 Motivation
   1.1 References ................................................................. 3
   1.2 Author ........................................................................... 4
   1.3 Acknowledgments ............................................................ 4

2 Citing NURBS-Python
   2.1 BibTeX .............................................................................. 5
   2.2 License .............................................................................. 5
   2.3 DOI Name .......................................................................... 5

3 Questions and Answers
   3.1 What is NURBS? ............................................................... 7
   3.2 Why NURBS-Python? ......................................................... 7
   3.3 Minimum Requirements ...................................................... 8
   3.4 Issues and Reporting .......................................................... 8
   3.5 API Changes ..................................................................... 8

4 Installation and Testing
   4.1 Install via Pip ................................................................. 9
   4.2 Install via Conda ............................................................... 9
   4.3 Manual Install ................................................................... 10
   4.4 Checking Installation ........................................................ 10
   4.5 Testing ............................................................................. 10

5 Examples Repository ................................................................ 11

6 Loading and Saving Data .......................................................... 13

7 Supported File Formats
   7.1 Text Files .......................................................................... 15
   7.2 Comma-Separated (CSV) ..................................................... 17
   7.3 VTK (Legacy) Format ........................................................ 18
   7.4 OBJ Format ....................................................................... 18
   7.5 STL Format ........................................................................ 19
   7.6 Object File Format (OFF) .................................................... 19
   7.7 Libconfig Format ............................................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Compatibility</td>
<td>21</td>
</tr>
<tr>
<td>9 Surface Generator</td>
<td>23</td>
</tr>
<tr>
<td>10 Visualization</td>
<td>25</td>
</tr>
<tr>
<td>11 Splitting and Decomposition</td>
<td>45</td>
</tr>
<tr>
<td>12 Exporting Plots as Image Files</td>
<td>55</td>
</tr>
<tr>
<td>13 Core Modules</td>
<td>57</td>
</tr>
<tr>
<td>14 Visualization Modules</td>
<td>131</td>
</tr>
<tr>
<td>15 Experimental Modules</td>
<td>141</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>143</td>
</tr>
</tbody>
</table>
Note: The version of the documentation that you are currently reading belongs to NURBS-Python v4.x series. Please see the following links, if you are using a different version of NURBS-Python:

- NURBS-Python v3.x
- NURBS-Python v2.x

Please note that there might be small API changes between the versions listed above.

Welcome to the NURBS-Python (geomdl) documentation! NURBS-Python contains native Python implementations of several The NURBS Book algorithms. These algorithms are used for generating Non-Uniform Rational B-Spline (NURBS) curves and surfaces.

NURBS-Python also provides a convenient and easy-to-use data structure for storing curve and surface descriptions. All elements of the provided data structures are documented under Modules.

This documentation is organized into a couple sections:

- Introduction
- Using the Library
- Modules
Motivation

NURBS-Python is an object-oriented Python library containing implementations of NURBS curve and surface generation and evaluation algorithms. It also provides a convenient and easy-to-use data structure for storing curve and surface descriptions.

Some significant features of NURBS-Python are:

- Fully object-oriented API with an abstract interface for extensions
- Data structures for storing surface and curve descriptions (in all dimensions)
- Helper functions, such as automatic uniform knot vector generator and many more
- Control points grid generator for surfaces
- Visualization module for direct plotting of curves and surfaces
- Shapes component for generation common surfaces and curves
- CSV export functionality with customizable meshing options
- Python 2.7.x and 3.x compatibility
- No external C/C++ library dependencies
- No compilation steps are necessary, everything is implemented in pure python
- Easy to install via pip: pip install geomdl
- Conda packages are also available for installation: conda install -c orbingol geomdl

1.1 References

NURBS-Python implements the following algorithms from The NURBS Book (2nd Edition) by Piegl & Tiller:

- A2.1 FindSpan (page 68)
- A2.2 BasisFuns (page 70)
1.2 Author

Onur Rauf B in gol

- E-mail: contact@onurbingol.net
- Twitter: https://twitter.com/orbingol
- LinkedIn: https://www.linkedin.com/in/onurraufbingol/

1.3 Acknowledgments

I would like to thank my PhD adviser, Dr. Adarsh Krishnamurthy, for his guidance and supervision throughout the course of this project.
CHAPTER 2

Citing NURBS-Python

2.1 BibTeX

@misc{bingol2016geomd1,
    author = {Onur Rauf Bingol},
    title = {NURBS-Python},
    year = 2016,
    doi = {10.5281/zenodo.815010},
    url = {https://doi.org/10.5281/zenodo.815010}
}

2.2 License

NURBS-Python is a free and open-source project released under the terms of the MIT License. Please see the LICENSE file for details.

2.3 DOI Name

The authors provide a Digital Object Identifier, i.e. DOI name obtained from Zenodo.org. The DOI name for NURBS-Python is 10.5281/zenodo.815010 and can also be found as a badge at the top, which links to NURBS-Python’s Zenodo page.
3.1 What is NURBS?

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

Please see the related Wikipedia article or The NURBS Book, a very nice and informative book written by Les A. Piegl and Wayne Tiller.

3.2 Why NURBS-Python?

The main purpose is implementing the well-known NURBS algorithms in native Python in an organized way and without using any converters or wrappers, like SWIG or Boost.Python.

Although these wrappers are lifesavers by means of converting C++ code to Python when there are too many deadlines, their support on the source language might be limited or you might need to learn the wrapper’s own language to get the things done in your way. Personally speaking, I had to learn a part of Python’s C API to understand how SWIG’s typemap system works. It takes so much time if you are not well-acquainted with the low-level programming or not willing to learn the inner details of the programming languages (and their interpreters, compilers, etc.).

On the other hand, NURBS-Python is designed to get the things done in a fast way. I used object-oriented approach throughout the library and tried to make the code look more pythonic and optimized. Since all the code is implemented in Python natively with no external dependencies, it is possible to use this library in every platform which core python programming language is supported or integrate into embedded systems/distributions. Using native implementation approach also allows users to debug and extend the library in a convenient way.
3.3 Minimum Requirements

NURBS-Python is tested on Python versions 2.7.x and 3.3.5+. 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 (or Miniconda).

3.4 Issues and Reporting

3.4.1 Contributions to NURBS-Python

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.

3.4.2 Bugs and Issues

Please use the issue tracker on GitHub for reporting bugs and other related issues.

I would be glad if you could provide as much detail as you can for pinpointing the problem. You don’t have to provide a solution for the problem that you encountered but it would be good if you would provide steps (preferably, as a list) to reproduce it. You may directly upload any data files required for testing to the issue tracker or email me if you feel that is more convenient for you.

Please note that the issue tracker is public and all written text and all uploaded files will be visible to everybody.

3.4.3 Comments, Questions and Feature Requests

You are encouraged to use the issue tracker on GitHub for your questions and comments. I would be glad if you could use the appropriate label (comment, question or feature request) to label your questions or comments on the issue tracker.

I also would like to leave the email communication open for NURBS-Python users. The issue tracker will stay as the preferred method for communication but I know some users don’t feel confident asking questions or commenting on a public system. I will try my best to reply back to your emails as soon as possible.

3.5 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.
CHAPTER 4

Installation and Testing

Note: You should remove NURBS-Python v2.x and v3.x before installing any version of v4.x. Uninstalling packages via pip is as easy as executing pip uninstall NURBS-Python. However, installation via setup.py requires manual removal of the packages from Python's site-packages directory. Directories to delete: nurbs and/or geomdl.

Installation via pip or conda is the recommended method for all users. Manual method is only recommended for advanced users.

4.1 Install via Pip

You may find NURBS-Python on Python Package Index and install via pip.

pip install geomdl

Upgrading to the latest version:

pip install geomdl --upgrade

4.2 Install via Conda

For your convenience, NURBS-Python has also been uploaded to Anaconda Cloud. You may use conda package manager to install and/or upgrade NURBS-Python.

To install:

conda install -c orbingol geomdl

To upgrade:

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 following command will copy NURBS-Python package to your Python distribution’s site-packages directory:

```bash
python setup.py install
```

If you don’t prefer copying for some reason (e.g. extension development, bug fixing and/or testing), you may run the following from the command line to generate a link to the directory where you cloned NURBS-Python package inside your Python distribution’s site-packages directory:

```bash
python setup.py develop
```

### 4.4 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:

```bash
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'
>>>
```

### 4.5 Testing

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

```bash
pytest or py.test
```

pytest will automatically find the tests under tests/ directory, execute them and show the results.
Using NURBS-Python is very easy: set degrees, control points and knot vectors, and you are all good. To give you a head-start on working with NURBS-Python, the authors also provide an Examples repository describing some basic usage of the library and its modules.

The Examples repository contains examples on the following:

- Bézier curves and surfaces
- B-Spline & NURBS curves and surfaces
- Control points grid generation for Surface classes
- Automatic knot vector generation for curves and surfaces
- Visualization components (Matplotlib and Plotly)
- Automatic NURBS curve and surface generation
- Curve & surface splitting and Bézier decomposition
- Exporting curves and surfaces using exchange module
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:

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

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

# Set the degree
curve.degree = 3

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

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

# Save the curve
curve.save("mycurve.pickle")
```
The saved curve can be loaded from the file with the following simple code segment:

```
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.

```
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.

7.1 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:

7.1.1 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:

\[
\begin{array}{cc}
\text{x} & \text{y} \\
\text{x}_1 & \text{y}_1 \\
\text{x}_2 & \text{y}_2 \\
\text{x}_3 & \text{y}_3 \\
\end{array}
\]
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:

<table>
<thead>
<tr>
<th>x(\ast)w</th>
<th>y(\ast)w</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1\ast w_1)</td>
<td>(y_1\ast w_1)</td>
<td>(w_1)</td>
</tr>
<tr>
<td>(x_2\ast w_2)</td>
<td>(y_2\ast w_2)</td>
<td>(w_2)</td>
</tr>
<tr>
<td>(x_3\ast w_3)</td>
<td>(y_3\ast w_3)</td>
<td>(w_3)</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1)</td>
<td>(y_1)</td>
<td>(z_1)</td>
</tr>
<tr>
<td>(x_2)</td>
<td>(y_2)</td>
<td>(z_2)</td>
</tr>
<tr>
<td>(x_3)</td>
<td>(y_3)</td>
<td>(z_3)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>x(\ast)w</th>
<th>y(\ast)w</th>
<th>z(\ast)w</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x_1\ast w_1)</td>
<td>(y_1\ast w_1)</td>
<td>(z_1\ast w_1)</td>
<td>(w_1)</td>
</tr>
<tr>
<td>(x_2\ast w_2)</td>
<td>(y_2\ast w_2)</td>
<td>(z_2\ast w_2)</td>
<td>(w_2)</td>
</tr>
<tr>
<td>(x_3\ast w_3)</td>
<td>(y_3\ast w_3)</td>
<td>(z_3\ast w_3)</td>
<td>(w_3)</td>
</tr>
</tbody>
</table>
Surfaces

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

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

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

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

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

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

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

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

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

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

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

7.2 Comma-Separated (CSV)

You may use `export_csv()` function to save control points and/or evaluated points as a CSV file. This function works with both curves and surfaces.
7.3 VTK (Legacy) Format

You may use `export_vtk()` function to save control points and/or evaluated points as a VTK file (legacy format). This function works with both curves and surfaces.

7.4 OBJ Format

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

7.4.1 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")
```

7.4.2 Example 2

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

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

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

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

# Evaluate the surface
surf.evaluate()

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

7.5 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.

7.6 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.

7.7 Libconfig Format

`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.

Since any NURBS shape can be generated by defining degree, knot vector and (weighted) control points, NURBS-Python libconfig format defines the following elements:

- **type** is a string defining the type of the shape, `curve` or `surface`
- **degree** for curves, `degree_u` and `degree_v` for surfaces
- **knotvector** for curves, `knotvector_u` and `knotvector_v` for surfaces
- **control_points** is a 1-dimensional list of coordinates
- **weights** is an array for NURBS shapes, otherwise it is set to 0
- **control_points_size_u** and **control_points_size_v** for defining surface control points grid size on the parametric space

In addition, NURBS-Python also exports the shape name and sample size values but these are not required to import shape data.

7.7.1 Reading Exported .cfg Files with libconf Module

The following example illustrates reading the exported .cfg file with `libconfig` module as a reference for libconfig-based systems in different programming languages.
# Assuming that you have already installed 'libconf'

```python
import libconf
```

# Skipping export steps and assuming that we have already exported the data as 'my_nurbs.cfg'

```python
with open("my_nurbs.cfg", "r") as fp:
    # Open the file and parse using libconf module
    data = libconf.load(fp)
```

# 'count' attribute shows the number of shapes loaded from the file

```python
print(data.count)
```

# C-style for loop for processing loaded shapes

```python
for idx in range(data.count):
    # 'shape' variable contains the curve or surface data
    shape = data.shapes[idx]
```

# Python-style for loop

```python
for shape in data.shapes:
    # As an example, we access to the control points attribute
    ctrlpts = shape.control_points
```

# If you are sure that you have loaded a single shape from the file:

```python
my_shape = data.shapes[0]
```

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

The following example illustrates the usage of `compatibility` module:

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

# Surface exported from your CAD software

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

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

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

# Degrees
p_degree_u = 3
p_degree_v = 2
```

(continues on next page)
# Prepare data for import
#
# Combine weights vector with the control points list
t_ctrlptsw = compat.combine_ctrlpts_weights(p_ctrlpts, p_weights)

# Since NURBS-Python uses v-row order, we need to convert the exported ones
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
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
#
## Create a NURBS surface instance
surf = NURBS.Surface()

## Using __call__ method to fill the surface object
surf(p_degree_u, p_degree_v, p_size_u, p_size_v, n_ctrlptsw, n_knotvector_u, n_knotvector_v)

## Set evaluation delta
surf.delta = 0.05

## Set visualization component
vis_comp = VisMPL.VisSurfTriangle()
surf.vis = vis_comp

## Render the surface
surf.render()

Please see *Compatibility Module Documentation* for more details on manipulating and exporting control points.
NURBS-Python has some other options for exporting and importing data. Please see *File Formats* page for details.
NURBS-Python comes with a simple surface generator which is designed to generate a control points grid to be used as a randomized input to `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.
NURBS-Python comes with the following visualization modules for direct plotting evaluated curves and surfaces:

- VisMPL module for Matplotlib
- VisPlotly module for Plotly

Examples repository contains examples on how to use the visualization components with surfaces and curves. Please see Visualization Modules Documentation for more details.

### 10.1 Examples

The following figures illustrate some example 2D/3D curves and surfaces that can be generated and directly visualized using NURBS-Python.
10.1.1 Surfaces

ex_surface01.py
10.1. Examples
ex_surface02.py
ex_surface03.py
The following figure illustrates tangent and normal vectors on `ex_surface02.py` example. The example script can be found in Examples repository under the visualization directory.

**mpl_trisurf_vectors.py**
10.1.2 2D Curves

ex_curve01.py
ex_curve02.py
NURBS-Python Documentation

10.1. Examples
10.1.3 3D Curves

ex_curve3d01.py
ex_curve3d02.py
10.1.4 Advanced Visualization for 2D/3D Curves

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

**mpl_curve2d_tangents.py**

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

### 11.1 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 $u = 0.5$. 

11.1. Splitting
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.
11.2 Bézier Decomposition

The following figures illustrate Bézier cecomposition 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.
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
import os
from geomdl import BSpline
from geomdl import utilities
from geomdl.visualization import VisMPL

# Fix file path
os.chdir(os.path.dirname(os.path.realpath(__file__)))

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

# Set up the Bezier curve
curve.degree = 3
curve.cpts = [[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.cpts))

# Set sample size
curve.sample_size = 40

# Evaluate curve
curve.evaluate()
```

(continues on next page)
# Plot the control point polygon and the evaluated curve
vis_comp = VisMPL.VisCurve3D()
curve.vis = vis_comp

# Don't pop up the plot window, instead save it as a PDF file
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 NURBS-Python library is more than a simple NURBS library. It includes n-variate surface and curve classes which provide a convenient data structure implemented with Python properties as well as the evaluation, and export/exchange functionality.

Following modules are included in the library:

### 13.1 Abstract Module

Abstract module provides base classes for all curves and surfaces contained in this library and therefore, it provides an easy way to extend the library in the most proper way.

#### 13.1.1 Abstract Curve

```python
class geomdl.Abstract.Curve(**kwargs)
    Bases: object

Abstract base class (ABC) for all n-variate curves.

The 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.
bbox
Bounding box.
Evaluates the bounding box of the curve and returns the minimum and maximum coordinates.

Getter  Gets bounding box
Type    tuple

ctrlpts
Control points.

Getter  Gets the control points
Setter  Sets the control points

degree
Curve degree.

Getter  Gets the curve degree
Setter  Sets the curve degree
Type    integer

delta
Curve 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:

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

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

derivatives \((u, order, **kwargs)\)
Evaluates the derivatives of the curve at parameter \(u\).

Parameters
- \(u\) (float) – parameter value
- \(order\) (int) – derivative order
dimension
Dimension of the curve.
Dimension will be automatically estimated from the first element of the control points array.

Getter  Gets the dimension of the curve, e.g. 2D, 3D, etc.
Type    integer
evalpts
Evaluated curve points.

Getter  Gets the coordinates of the evaluated points

evaluate (**kwargs)
Evaluates the curve.
**evaluate_list** *(u_list)*

Evaluates the curve for an input range of parameters.

**Parameters**

- **u_list** – array of parameters

**evaluate_single** *(u)*

Evaluates the curve at the given parameter.

**Parameters**

- **u** – parameter

**evaluator**

Curve evaluator.

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

- **Getter** Gets the current `Evaluator` instance
- **Setter** Sets the evaluator

**knotvector**

Knot vector.

- **Getter** Gets the knot vector
- **Setter** Sets the knot vector

**name**

Curve descriptor (as a string or a number).

Descriptor field allows users to assign an identification to the curve object. The identification can be a string or a number.

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

**order**

Curve order.

Defined as order = degree + 1

- **Getter** Gets the curve order
- **Setter** Sets the curve order

- **Type** integer

**rational**

Returns True if the curve is rational.

**render** (**kwargs**)

Renders the curve using the loaded 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
- `filename`: saves the plot with the input name
- `plot`: a flag to control displaying the plot window. Default is True.
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.

```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

```python
sample_size
```

Sample size.

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

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

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

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

```python
set_ctrlpts(ctrlpts, **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.

It accepts a keyword argument `array_init` which defaults to a list of size \(\text{len}(\text{ctrlpts})\) where `ctrlpts` is the input list of control points. `array_init` keyword argument may be used to input other types of arrays to this method.

The following example illustrates a way to use a NumPy array with this method.

```python
# Import numpy
import numpy as np

# Assuming that "ctrlpts" is a NumPy array of a shape \((x,y)\) where \(x == len(ctrlpts)\) and \(y == len(ctrlpts[0])\)
curve.set_ctrlpts(ctrlpts, array_init=np.zeros(ctrlpts.shape, dtype=np.float32))
```

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

**vis**

Visualization component.

**Note:** The visualization component is completely optional to use.

```python
getter
```

Gets the visualization component
13.1.2 Abstract Surface

```python
class geomdl.Abstract.Surface(**kwargs):
    Bases: object
    
    Abstract base class (ABC) for all surfaces.
    
    The 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.

bbox
    Bounding box.
    
    Evaluates the bounding box of the surface and returns the minimum and maximum coordinates.
    
    Getter Gets bounding box

ctrlpts
    1-D control points.
    
    Getter Gets the control points
    
    Setter Sets the control points

ctrlpts2d
    2-D control points.
    
    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

ctrlpts_size_u
    Size of the control points array on the u-direction.
    
    Getter Gets number of control points on the u-direction
    
    Setter Sets number of control points on the u-direction

ctrlpts_size_v
    Size of the control points array on the v-direction.
    
    Getter Gets number of control points on the v-direction
    
    Setter Sets number of control points on the v-direction

degree_u
    Surface degree for u-direction.
    
    Getter Gets the surface degree for u-direction
    
    Setter Sets the surface degree for u-direction
```
**Type** integer

**degree_v**
Surface degree for v-direction.

**Getter** Gets the surface degree for v-direction

**Setter** Sets the surface degree for v-direction

**Type** integer

**delta**
Evaluation delta for both u- and v-directions.

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

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

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

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

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

**Setter** Sets the same delta value 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`.

**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* 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`.

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

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

**Type** float

**derivatives** *(u, v, order, **kwargs]*)

Evaluates the derivatives of the surface at parameter *(u,v).*
Parameters

- \( u (\text{float}) \) – parameter on the u-direction
- \( v (\text{float}) \) – parameter on the v-direction
- \( \text{order} (\text{int}) \) – derivative order

**dimension**

Dimension of the surface.

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

**Getter** Gets the dimension of the surface

**Type** integer

**evalpts**

Evaluated surface points.

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

**evaluate\(*\text{kwargs}\)**

Evaluates the surface.

**evaluate_list\((uv\text{\_list})\)**

Evaluates the surface for an input range of \((u,v)\) parameter pairs.

**Parameters** \(uv\text{\_list}\) – array of parameter pairs \((u, v)\)

**evaluate\_single\((uv)\)**

Evaluates the surface at the given \((u,v)\) parameter.

**Parameters** \(uv\) – parameter pair \((u, v)\)

**evaluator**

Curve evaluator.

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

**Getter** Prints the name of the evaluator and returns the current Evaluator instance

**Setter** Sets the evaluator

**knotvector\_u**

Knot vector for u-direction.

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

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

**knotvector\_v**

Knot vector for v-direction.

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

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

**name**

Surface descriptor (as a string or a number).

Descriptor field allows users to assign an identification to the surface object. The identification can be a string or a number.

**Getter** Gets the descriptor

**Setter** Sets the descriptor
order_u
Surface order for u-direction.
Follows the following equality: order = degree + 1

Getter Gets the surface order for u-direction
Setter Sets the surface order for u-direction
Type integer

order_v
Surface order for v-direction.
Follows the following equality: order = degree + 1

Getter Gets the surface order for v-direction
Setter Sets the surface order for v-direction
Type integer

rational
Returns True if the surface is rational.

render(**kwargs)
Renders the surface using the loaded 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: a flag to control displaying the plot window. Default is True.
- 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.

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:

\[
\begin{bmatrix} u_{\text{start}}, \ldots, u_{\text{end}} \end{bmatrix}_n = \text{sample size}
\]

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

**Setter** Sets the same sample size value 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` property.

**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` property.

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

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

**Type** int

### set_ctrlpts(ctrlpts, size_u, size_v, **kwargs)

Sets 1-dimensional 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`.

You may initialize the 1-dimensional and 2-dimensional arrays via `array_init` and `array_init2d` keyword arguments. Please see `Curve.set_ctrlpts()` for details.

**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
- `size_u` (int) – size of the control points grid on the u-direction
- `size_v` (int) – size of the control points grid on the v-direction

**Returns** None

### tessellate(**kwargs)

Tessellates the surface.

Keyword arguments are directly passed to the tessellation component.
tessellator
Tessellation component.

- **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.

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

vis
Visualization component.

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

---

### 13.2 B-Spline Module

The **BSpline** module provides data storage and evaluation functions for B-Spline (NUBS) curves and surfaces.

#### 13.2.1 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.

Notes:

- Please see the `Abstract.Curve()` documentation for details.
- This class sets the `FindSpan` implementation to Linear Search by default.

bbox
Bounding box.
Evaluates the bounding box of the curve and returns the minimum and maximum coordinates.

- **Getter** Gets 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.

Getter Gets the control points

Setter Sets the control points

Type list

`curvept(u)`
Evaluates the curve at the given parameter.

Parameters `u(float)` – parameter

Returns evaluated curve point

Return type list

`degree`
Curve degree.

Getter Gets the curve degree

Setter Sets the curve degree

Type integer

`delta`
Curve evaluation delta.

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

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

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

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 \(\{\text{order}\}\)-th derivative of the curve

Return type list
**dimension**

Dimension of the curve.

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

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

**Type** integer

**evalpts**

Evaluated curve points.

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

**evaluate(**

Evaluates the curve.

The evaluated curve points are stored in :py:attr:`-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**(*u_list*)

Evaluates the curve for an input range of parameters.

**Parameters** u_list (list, tuple) – list of parameters

**Returns** evaluated surface points at the input parameters

**Return type** tuple

**evaluate_single**(*u*)

Evaluates the curve at the given parameter.

**Parameters** u (float) – parameter

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

**Return type** list

**evaluator**

Curve evaluator.

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

Getter
Gets the current Evaluator instance

Setter
Sets the evaluator

insert_knot \( (u, r=1, \text{check } r=True) \)
Inserts the given knot and updates the control points array and the knot vector.

Parameters
- \( u (\text{float}) \) – knot to be inserted
- \( r (\text{int}) \) – number of knot insertions
- \( \text{check } r (\text{bool}) \) – enables/disables number of knot insertions check

knotvector
Knot vector.

Getter
Gets the knot vector

Setter
Sets the knot vector

Type
list

load \( (\text{file } \text{name}) \)
Loads the curve from a pickled file.

Parameters
- \( \text{file } \text{name} (\text{str}) \) – name of the file to be loaded

Raises
IOError – an error occurred reading the file

name
Curve descriptor (as a string or a number).

Descriptor field allows users to assign an identification to the curve object. The identification can be a string or a number.

Getter
Gets the descriptor

Setter
Sets the descriptor

normal \( (\text{parpos}, **\text{kwargs}) \)
Evaluates the normal vector of the curve at the given parametric position(s).

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

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

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

Return type
tuple

order
Curve order.

Defined as \( \text{order} = \text{degree} + 1 \)
Getter  Gets the curve order
Setter  Sets the curve order
Type  integer

**rational**
Returns True if the curve is rational.

**render(**kwargs)**
Renders the curve using the loaded 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
- filename: saves the plot with the input name
- plot: a flag to control displaying the plot window. Default is True.

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

**reset(**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 curve points to generate. It also sets the delta property.

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

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

\[ \frac{1}{N_{\text{sample}}} \]

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, **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.

It accepts a keyword argument `array_init` which defaults to a list of size \(\text{len(ctrlpts)}\) where `ctrlpts` is the input list of control points. `array_init` keyword argument may be used to input other types of arrays to this method.

The following example illustrates a way to use a NumPy array with this method.

```python
# Import numpy
import numpy as np

# Assuming that "ctrlpts" is a NumPy array of a shape (x,y) where x == \rightarrow\text{len(ctrlpts)} and y == \rightarrow\text{len(ctrlpts[0])}
curve.set_ctrlpts(ctrlpts, array_init=np.zeros(ctrlpts.shape, dtype=np.float32))
```

### 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`

---

**Note:** The visualization component is completely optional to use.

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

---

### 13.2.2 B-Spline Surface

**class** `geomdl.BSpline.Surface(**kwargs)`

**Bases:** `geomdl.Abstract.Surface`

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

**Notes:**
• Please see the Abstract.Surface() documentation for details.
• This class sets the FindSpan implementation to Linear Search by default.

bbox
Bounding box.

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

Getter Gets bounding box

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.

Getter Gets the control points
Setter Sets the control points
Type list
ctrlpts2d
2-dimensional array of control points.

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

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

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

# Do degree, control points and knot vector assignments here

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

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

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

# Each u includes a row of v values
for u in surf_nb.ctrlpts2d:
    # Each row contains the coordinates of the weighted control points
    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.

- **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_u
Size of the control points array on the u-direction.
- **Getter** Gets number of control points on the u-direction
- **Setter** Sets number of control points on the u-direction

### ctrlpts_size_v
Size of the control points array on the v-direction.
- **Getter** Gets number of control points on the v-direction
- **Setter** Sets number of control points on the v-direction

### degree_u
Surface degree for u-direction.
- **Getter** Gets the surface degree for u-direction
- **Setter** Sets the surface degree for u-direction
- **Type** integer

### degree_v
Surface degree for v-direction.
- **Getter** Gets the surface degree for v-direction
- **Setter** Sets the surface degree for v-direction
- **Type** integer

### 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:

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

- **Getter** Gets the delta values as a tuple of values corresponding to u- and v-directions
- **Setter** Sets the same delta value 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`.

- **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* 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`.

- **Getter** Gets the delta value for the v-direction
- **Setter** Sets the delta value 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**

Dimension of the surface.

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

- **Getter** Gets the dimension of the surface
- **Type** integer
evalpts
   Evaluated surface points.
   
   **Getter** Gets the coordinates of the evaluated points

**evaluate(****kwargs**)**
   Evaluates the surface.
   
   *The evaluated surface points are stored in :py:attr:`evalpts` property.*

**Keyword arguments:**

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

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

The following examples illustrate the usage of the keyword arguments.

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

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

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

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

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

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

**Return type** tuple

**evaluate_single** *(uv)*
   Evaluates the surface at the given (u,v) parameter pair.

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

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

**Return type** list

**evaluator**
   Curve evaluator.

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

   **Getter** Prints the name of the evaluator and returns the current Evaluator instance

   **Setter** Sets the evaluator

**insert_knot** *(u=None, v=None, ru=1, rv=1, check_r=True)*
   Inserts the knot in single dimension, with only u or v input, or multi-dimensions, with a (u,v) pair input.
Parameters

- \( u (\text{float}) \) – Knot to be inserted on the \( u \)-direction
- \( v (\text{float}) \) – Knot to be inserted on the \( v \)-direction
- \( \text{ru} (\text{int}) \) – Number of knot insertions on the \( u \)-direction
- \( \text{rv} (\text{int}) \) – Number of knot insertions on the \( v \)-direction
- \( \text{check}_r (\text{bool}) \) – enables/disables number of knot insertions check

**knotvector_u**
Knot vector for \( u \)-direction.

- **Getter** Gets the knot vector for \( u \)-direction
- **Setter** Sets the knot vector for \( u \)-direction
- **Type** list

**knotvector_v**
Knot vector for \( v \)-direction.

- **Getter** Gets the knot vector for \( v \)-direction
- **Setter** Sets the knot vector for \( v \)-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**
Surface descriptor (as a string or a number).
Descriptor field allows users to assign an identification to the surface object. The identification can be a string or a number.

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

**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
Surface order for u-direction.
Follows the following equality: order = degree + 1

Getter  Gets the surface order for u-direction
Setter  Sets the surface order for u-direction
Type  integer

order_v
Surface order for v-direction.
Follows the following equality: order = degree + 1

Getter  Gets the surface order for v-direction
Setter  Sets the surface order for v-direction
Type  integer

rational
Returns True if the surface is rational.

render (**kwargs)
Renders the surface using the loaded visualization component.
The visualization component must be set using \textit{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: a flag to control displaying the plot window. Default is True.
  • colormap: sets the colormap of the surface

The \textit{plot} argument is useful when you would like to work on the command line without any window context. If \textit{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 \textit{colormap} argument can only work with visualization classes that support colormaps. As an example, please see \texttt{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:

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

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

**Setter** Sets the same sample size value 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` property.

**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` property.

**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, size_u, size_v, **kwargs)
Sets 1-dimensional 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`.

You may initialize the 1-dimensional and 2-dimensional arrays via `array_init` and `array_init2d` keyword arguments. Please see `Curve.set_ctrlpts()` for details.

**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
- `size_u` *(int)* – size of the control points grid on the u-direction
• size_v \text{(int)} – size of the control points grid on the v-direction

Returns None

surfpt(u, v)
Evaluates the surface at the given (u,v) parameter pair.

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

Returns evaluated surface point at the given parameter pair

Return type list

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

The \text{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 \text{True}.

Parameters \text{parpos} \text{(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.

Getter Gets the tessellation component

Setter Sets the tessellation component

transpose()
Transposes the surface by swapping u- and v-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.

Getter Gets the array of trim curves

Setter Sets the array of trim curves
13.3 NURBS Module

NURBS module provides data storage and evaluation functions for NURBS curves and surfaces.

13.3.1 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
• 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.

Notes:

• Please see the Abstract.Surface() documentation for details.
• This class sets the FindSpan implementation to Linear Search by default.

bbox
    Bounding box.

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

    Getter Gets 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
Unweighted control points (P).

Getter Gets unweighted control points. Use weights to get weights vector.
Setter Sets unweighted control points
Type list

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.

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

curvept(u)
Evaluates the curve at the given parameter.

Parameters u (float) – parameter
Returns evaluated curve point
Return type list

degree
Curve degree.

Getter Gets the curve degree
Setter Sets the curve degree
Type integer

delta
Curve evaluation delta.

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

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

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

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 \( \text{order} \)-th derivative of the curve

**Return type** list

dimension
Dimension of the curve.

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

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

**Type** integer

evalpts
Evaluated curve points.

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

evaluate(**kwargs)
Evaluates the curve.

The evaluated curve points are stored in :py:attr:`~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(u_list)
Evaluates the curve for an input range of parameters.

**Parameters** u_list (list, tuple) – list of parameters

**Returns** evaluated surface points at the input parameters

**Return type** tuple

evaluate_single(u)
Evaluates the curve at the given parameter.

**Parameters** u (float) – parameter

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

**Return type** list
**evaluator**

Curve evaluator.

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

- **Getter** Gets the current Evaluator instance
- **Setter** Sets the evaluator

**insert_knot** \((u, r=1, check_r=True)\)

Inserts the given knot and updates the control points array and the knot vector.

**Parameters**

- \(u\) (**float**) – knot to be inserted
- \(r\) (**int**) – number of knot insertions
- \(check_r\) (**bool**) – enables/disables number of knot insertions check

**knotvector**

Knot vector.

- **Getter** Gets the knot vector
- **Setter** Sets the knot vector

**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**

Curve descriptor (as a string or a number).

Descriptor field allows users to assign an identification to the curve object. The identification can be a string or a number.

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

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

**Getter**  Gets the curve order
**Setter**  Sets the curve order

**Type**  integer

rational
Returns True if the curve is rational.

render(**kwargs)
Renders the curve using the loaded visualization component

The visualization component must be set using \( \text{vis} \) property before calling this method.

**Keyword Arguments:**
- \( \text{cpcolor} \): sets the color of the control points polygon
- \( \text{evalcolor} \): sets the color of the curve
- \( \text{filename} \): saves the plot with the input name
- \( \text{plot} \): a flag to control displaying the plot window. Default is True.

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

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

**Keyword Arguments:**
- \( \text{evalpts} \): if True, then resets evaluated points
- \( \text{ctrlpts} \): if True, then resets control points

sample_size
Sample size.
Sample size defines the number of curve points to generate. It also sets the \( \text{delta} \) property.

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

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

**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, **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.

It accepts a keyword argument `array_init` which defaults to a list of size `len(ctrlpts)` where `ctrlpts` is the input list of control points. `array_init` keyword argument may be used to input other types of arrays to this method.

The following example illustrates a way to use a NumPy array with this method.

```python
# Import numpy
import numpy as np
# Assuming that "ctrlpts" is a NumPy array of a shape (x,y) where x == len(ctrlpts) and y == len(ctrlpts[0])
curve.set_ctrlpts(ctrlpts, array_init=np.zeros(ctrlpts.shape, dtype=np.float32))
```

**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.

**Note:** The visualization component is completely optional to use.

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

**weights**

Weights vector.

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

Type list

13.3.2 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.

Notes:

• Please see the Abstract.Surface() documentation for details.
• This class sets the FindSpan implementation to Linear Search by default.

bbox

Bounding box.

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

Getter Gets bounding box

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:
```

(continues on next page)
# 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.

**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_u**
Size of the control points array on the u-direction.

**Getter** Gets number of control points on the u-direction

**Setter** Sets number of control points on the u-direction

**ctrlpts_size_v**
Size of the control points array on the v-direction.

**Getter** Gets number of control points on the v-direction

**Setter** Sets number of control points on the v-direction

**ctrlptsw**
1-dimensional array of weighted control points (Pw).

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

This property sets and gets the control points in 1-D.

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

**degree_u**
Surface degree for u-direction.

**Getter** Gets the surface degree for u-direction

**Setter** Sets the surface degree for u-direction

Type  integer

**degree_v**
Surface degree for v-direction.

**Getter** Gets the surface degree for v-direction

**Setter** Sets the surface degree for v-direction

Type  integer

**delta**
Evaluation delta for both u- and v-directions.

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

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

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

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

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

**Setter** Sets the same delta value 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.

**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 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.
**Getter**  Gets the delta value for the v-direction

**Setter**  Sets the delta value 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**
Dimension of the surface.
Dimension will be automatically estimated from the first element of the control points array.

**Getter**  Gets the dimension of the surface

**Type**  integer

**evalpts**
Evaluated surface points.

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

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

The evaluated surface points are stored in :py:attr:`evalpts` property.

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

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

The following examples illustrate the usage of the keyword arguments.
# Start evaluating in range u=[0, 0.7] and v=[0.1, 1]
surf.evaluate(stop_u=0.7, start_v=0.1)

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

# Get the evaluated points
surface_points = surf.evalpts

evaluate_list (uv_list)
   Evaluates the surface for a given list of (u,v) parameters.
   Parameters uv_list (list, tuple) – list of parameter pairs (u, v)
   Returns evaluated surface point at the input parameter pairs
   Return type tuple

evaluate_single (uv)
   Evaluates the surface at the given (u,v) parameter pair.
   Parameters uv (list, tuple) – parameter pair (u, v)
   Returns evaluated surface point at the given parameter pair
   Return type list

evaluator
   Curve evaluator.
   Evaluators allow users to use different algorithms for B-Spline and NURBS evaluations. Please see the documentation on Evaluator classes.
   Getter Prints the name of the evaluator and returns the current Evaluator instance
   Setter Sets the evaluator

insert_knot (u=None, v=None, ru=1, rv=1, check_r=True)
   Inserts the knot in single dimension, with only u or v input, or multi-dimensions, with a (u,v) pair input.
   Parameters
      • u (float) – Knot to be inserted on the u-direction
      • v (float) – Knot to be inserted on the v-direction
      • ru (int) – Number of knot insertions on the u-direction
      • rv (int) – Number of knot insertions on the v-direction
      • check_r (bool) – enables/disables number of knot insertions check

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

knotvector_v
   Knot vector for v-direction.
   Getter Gets the knot vector for v-direction
Setter  Sets the knot vector for 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
Surface descriptor (as a string or a number).

Descriptor field allows users to assign an identification to the surface object. The identification can be a
string or a number.

Getter  Gets the descriptor

Setter  Sets the descriptor

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
Surface order for u-direction.

Follows the following equality: order = degree + 1

Getter  Gets the surface order for u-direction

Setter  Sets the surface order for u-direction

Type  integer

order_v
Surface order for v-direction.

Follows the following equality: order = degree + 1

Getter  Gets the surface order for v-direction

Setter  Sets the surface order for v-direction

Type  integer
rational
Returns True if the surface is rational.

render(**kwargs)
Renders the surface using the loaded 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: a flag to control displaying the plot window. Default is True.
• 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.
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:

\[
\begin{bmatrix}
\mathbf{u}_{\text{start}} & \ldots & \mathbf{u}_{\text{end}} \\
\end{bmatrix}
\]

Getter Gets sample size values as a tuple of values corresponding to u- and v-directions
Setter Sets the same sample size value 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 property.

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` property.

**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, size_u, size_v, **kwargs)
Sets 1-dimensional 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`.

You may initialize the 1-dimensional and 2-dimensional arrays via `array_init` and `array_init2d` keyword arguments. Please see `Curve.set_ctrlpts()` for details.

**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
- `size_u`(int) – size of the control points grid on the u-direction
- `size_v`(int) – size of the control points grid on the v-direction

**Returns**
None

**surfpt**(u, v)
Evaluates the surface at the given (u,v) parameter pair.

**Parameters**
- `u`(float) – parameter on the u-direction
- `v`(float) – parameter on the v-direction

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

**Return type**
list

**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.

**Getter**

Gets the tessellation component

**Setter**

Sets the tessellation component

transpose ()

Transposes the surface by swapping u- and v-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.

**Getter**

Gets the array of trim curves

**Setter**

Sets the array of trim curves

vis

Visualization component.

**Getter**

Gets the visualization component

**Setter**

Sets the visualization component

weights

Weights vector.

**Getter**

Gets the weights vector

**Setter**

Sets the weights vector

**Type**

list

### 13.4 Evaluators

Evaluators (or evaluation strategies) allow users to change curve and/or surface evaluation strategy, i.e. the algorithms that are used to evaluate the curve & surface, take derivatives and more. Therefore, the user can switch between the evaluation algorithms at runtime, implement and use different algorithms or improve existing ones.
13.4.1 Abstract Evaluators

This class provides an abstract base for all evaluator classes.

class geomdl.Abstract.Evaluator(**kwargs)
    Bases: object
    
    Evaluator abstract base class.
    
    The methods evaluate and derivative is intended to be used for computation over a range of values. The suggested usage of evaluate_single and derivative_single methods are computation of a single value.
    
    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 over a range of parameters.

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

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

evaluate_single (**kwargs)
    Abstract method for computation of a single point at a single parameter.

name
    Evaluator name (as a string).

    Getter Gets the name of the evaluator
    
    Type str

The following classes provide curve and surface customizations for the abstract base.

class geomdl.Abstract.CurveEvaluator(**kwargs)
    Bases: object
    
    Curve customizations for Evaluator abstract base class.

    insert_knot (**kwargs)
        Abstract method for implementation of knot insertion algorithm.

class geomdl.Abstract.SurfaceEvaluator(**kwargs)
    Bases: object
    
    Surface customizations for the Evaluator abstract base class.

    insert_knot_u (**kwargs)
        Abstract method for implementation of knot insertion algorithm on the u-direction.

    insert_knot_v (**kwargs)
        Abstract method for implementation of knot insertion algorithm on the v-direction.

13.4.2 Included Evaluators

These evaluators are implementations of the above abstract base classes and all are included in the NURBS-Python package.
class geomdl.evaluators.CurveEvaluator(**kwargs)

Sequential B-Spline curve evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.1: CurvePoint
- Algorithm A3.2: CurveDerivsAlg1
- Algorithm A5.1: CurveKnotIns

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 n-th order curve derivatives over a range of parameters.

derivatives_single (**kwargs)
    Evaluates n-th order curve derivatives at a single parameter.

evaluate (**kwargs)
    Evaluates the curve.

evaluate_single (**kwargs)
    Evaluates a single curve point.

insert_knot (**kwargs)
    Insert knot multiple times at a single parameter.

name
    Evaluator name (as a string).

    Getter  Gets the name of the evaluator

    Type  str

class geomdl.evaluators.CurveEvaluator2(**kwargs)
    Bases: geomdl.evaluators.CurveEvaluator

Sequential B-Spline curve evaluation algorithms (alternative).

This evaluator implements the following algorithms from The NURBS Book:

- Algorithm A3.1: CurvePoint
- Algorithm A3.4: CurveDerivsAlg2
- Algorithm A5.1: CurveKnotIns

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 n-th order curve derivatives over a range of parameters.

static 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.
derivatives_single(**kwargs)
   Evaluates n-th order curve derivatives at a single parameter.

evaluate(**kwargs)
   Evaluates the curve.

evaluate_single(**kwargs)
   Evaluates a single curve point.

insert_knot(**kwargs)
   Insert knot multiple times at a single parameter.

ame
   Evaluator name (as a string).
      Getter Gets the name of the evaluator
      Type   str

class geomdl.evaluators.NURBSCurveEvaluator(**kwargs)
   Bases: geomdl.evaluators.CurveEvaluator

Sequential NURBS curve evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:
   • Algorithm A3.1: CurvePoint
   • Algorithm A4.2: RatCurveDerivs
   • Algorithm A5.1: CurveKnotIns

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 n-th order curve derivatives over a range of parameters.

derivatives_single(**kwargs)
   Evaluates n-th order curve derivatives at a single parameter.

evaluate(**kwargs)
   Evaluates the curve.

evaluate_single(**kwargs)
   Evaluates a single curve point.

insert_knot(**kwargs)
   Insert knot multiple times at a single parameter.

name
   Evaluator name (as a string).
      Getter Gets the name of the evaluator
      Type   str

class geomdl.evaluators.NURBSSurfaceEvaluator(**kwargs)
   Bases: geomdl.evaluators.SurfaceEvaluator

Sequential NURBS surface evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:
   • Algorithm A4.3: SurfacePoint
• Algorithm A4.4: RatSurfaceDerivs
• Algorithm A5.3: SurfaceKnotIns

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)
def derivatives_single(**kwargs)

def evaluate(**kwargs)
def evaluate_single(**kwargs)

def insert_knot_u(**kwargs)
def insert_knot_v(**kwargs)

name

Getter  Gets the name of the evaluator
Type   str

class geomdl.evaluators.SurfaceEvaluator(**kwargs)


Sequential B-Spline surface evaluation algorithms.

This evaluator implements the following algorithms from The NURBS Book:

• Algorithm A3.5: SurfacePoint
• Algorithm A3.6: SurfaceDerivsAlg1
• Algorithm A5.3: SurfaceKnotIns

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)
def derivatives_single(**kwargs)

def evaluate(**kwargs)
def evaluate_single(**kwargs)

def insert_knot_u(**kwargs)
**insert_knot_v(**kwargs**)
Inserts knot(s) in v-direction.

**name**
Evaluator name (as a string).

**Getter** Gets the name of the evaluator

**Type** *str*

**class** `geomdl.evaluators.SurfaceEvaluator2(**kwargs**)`
**Bases:** `geomdl.evaluators.SurfaceEvaluator`
Sequential B-Spline 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
- Algorithm A5.3: SurfaceKnotIns

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 n-th order surface derivatives over a range of (u,v) parameters.

**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.

**derivatives_single(**kwargs**)
Evaluates the n-th order surface derivatives at (u,v) parameters.

Output is SKL[k][l], derivative of the surface k times with respect to U and l times with respect to V

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

**evaluate_single(**kwargs**)
Evaluates a single surface point.

**insert_knot_u(**kwargs**)
Inserts knot(s) in u-direction.

**insert_knot_v(**kwargs**)
Inserts knot(s) in v-direction.

**name**
Evaluator name (as a string).

**Getter** Gets the name of the evaluator

**Type** *str*
13.5 Operations Module

This module provides common geometric operations for curves and surfaces.

```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 the updated shape.

Useful when converting a 2-dimensional curve to a 3-dimensional curve.

Parameters

- **obj** (BSpline.Curve or NURBS.Curve) – Curve

Returns

updated Curve

Return type

BSpline.Curve or NURBS.Curve

```python
geomdl.operations.binormal(obj, params, **kwargs)
```

Evaluates the binormal vector of the curves or surfaces at the input parameter values.

This function is designed to evaluate binormal vectors of the B-Spline and NURBS shapes at single or multiple parameter positions.

Parameters

- **obj** (Abstract.Curve or Abstract.Surface) – input shape
- **params** (float, list or tuple) – parameters

Returns

a list containing “point” and “vector” pairs

Return type

tuple

```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.

Parameters

- **obj** (BSpline.Curve or NURBS.Curve) – Curve to be decomposed

Returns

a list of curve objects arranged in Bezier curve segments

Return type

Multi.MultiCurve

```python
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.

Parameters

- **obj** (BSpline.Surface or NURBS.Surface) – Surface

Returns

a list of surface objects arranged as Bezier surface patches

Return type

Multi.MultiSurface

```python
geomdl.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
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 shape
• params(float, list or tuple) – parameters

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

Return type tuple

geomdl.operations.split_curve(obj, u, **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.

Parameters

• obj(BSpline.Curve or NURBS.Curve) – Curve to be split
• u(float) – parametric coordinate

Returns a list of curves as the split pieces of the initial curve

Return type Multi.MultiCurve

geomdl.operations.split_surface_u(obj, t, **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.

Parameters

• obj(BSpline.Surface or NURBS.Surface) – Surface
• t(float) – parametric coordinate on the u-direction

Returns a list of surface as the split pieces of the initial surface

Return type Multi.MultiSurface

geomdl.operations.split_surface_v(obj, t, **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.

Parameters

• obj(BSpline.Surface or NURBS.Surface) – Surface
• t(float) – parametric coordinate on the v-direction

Returns a list of surface as the split pieces of the initial surface

Return type Multi.MultiSurface
Geomdl documentation includes several utility and helper functions that help computation of several common linear algebra and geometry operations.

### 13.6 Utilities and Helpers

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

#### 13.6.1 Utilities

The **Utilities** module contains several utility functions that help computation of several common linear algebra and geometry 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.utilities.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.utilities.check_knot_vector(degree, knot_vector, num_ctrlpts)**

Checks if the input knot vector follows the mathematical rules.

**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

geomdl.utilities.check_uv(*u=None, v=None*)
Checks if the parameter values are valid.

Parameters
• **u** (*float*) – u parameter
• **v** (*float*) – v parameter

Raises ValueError – u and/or v is not in the interval [0, 1]

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*)
Evaluates the bounding box of a curve or a surface.

Parameters
**ctrlpts** (*list, tuple*) – control points

Returns bounding box

Return type list

geomdl.utilities.frange(*start, stop, step=1.0*)
Implementation of Python’s range() function which works with floats.
Reference to this implementation: https://stackoverflow.com/a/36091634

Parameters
• **start** (*float*) – start value
• **stop** (*float*) – end value
• **step** (*float*) – increment

Returns float

Return type generator

geomdl.utilities.generate_knot_vector(*degree, num_ctrlpts, **kwargs*)
Generates a uniformly-spaced knot vector using the degree and the number of control points.
It uses the following equation 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 (integer)` – degree
- `num_ctrlpts (integer)` – number of control points

Returns uniform knot vector

Return type: list

```python
geomdl.utilities.linspace(start, stop, num, decimals=6)
```

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

```python
geomdl.utilities.make_quad_mesh(points, size_u, size_v)
```

Generates a quad mesh from linearly ordered list of points.

Parameters
- `points (list, tuple)` – list of points to be ordered
- `size_v (int)` – number of elements in a row
- `size_u (int)` – number of elements in a column

Returns re-ordered points

Return type: list

```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

`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
- **trims**: List of trim curves passed to the tessellation function
- **tessellate_func**: Function called for tessellation (default is `triangular_tessellation`)
- **tessellate_args**: Arguments passed to the tessellation function

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)`

Changes linearly ordered list 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
   <-----------><---><--><----------->
   ---------<-------><-------<-------->
```

(continues on next page)
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.normalize_knot_vector(knot_vector, decimals=4)`

Normalizes the input knot vector between 0 and 1.

**Parameters**

- `knot_vector(list, tuple)` – knot vector to be normalized
- `decimals(int)` – rounding number

**Returns**
normalized knot vector

**Return type** list

`geomdl.utilities.point_distance(pt1, pt2)`

Computes distance between two points.

**Parameters**

- `pt1(list, tuple)` – point 1
- `pt2(list, tuple)` – point 2

**Returns**
distance between input points

**Return type** float

`geomdl.utilities.point_mid(pt1, pt2)`

Computes the midpoint of the two points.

**Parameters**

- `pt1(list, tuple)` – point 1
- `pt2(list, tuple)` – point 2

**Returns**
midpoint

**Return type** tuple

`geomdl.utilities.point_translate(point_in, vector_in)`

Translates the input points using the input vector.

**Parameters**

- `point_in(list, tuple)` – input point
- `vector_in(list, tuple)` – input vector

**Returns**
translated point

**Return type** tuple
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** (tuple) – list of Vertex objects

Returns list of Triangle objects

Return type list

tuple 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

geomdl.utilities.vector_angle_between(vector1, vector2, **kwargs)

Computes the angle between the two input vectors.

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

Parameters

- **vector1** (list, tuple) – vector
- **vector2** (list, tuple) – vector

Returns angle between the vectors

Return type float

geomdl.utilities.vector_cross(vector1, vector2)

Computes the cross-product of the input vectors.

Parameters

- **vector1** (list, tuple) – input vector 1
- **vector2** (list, tuple) – input vector 2

Returns result of the cross product

Return type tuple

geomdl.utilities.vector_dot(vector1, vector2)

Computes the dot-product of the input vectors.
Parameters

• vector1(list, tuple) – input vector 1
• vector2(list, tuple) – input vector 2

Returns result of the dot product

Return type float

gemdl.utilities.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 tuple

gemdl.utilities.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

gemdl.utilities.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

gemdl.utilities.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  tuple

`geomdl.utilities.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.utilities.vector_normalize(vector_in, decimals=6)`
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  tuple

13.6.2 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 knot.

Implementation of Algorithm A2.2 from The NURBS Book by Piegl & Tiller.

Parameters
- `degree(int)` – degree
- `knot_vector(list, tuple)` – knot vector
- `span(int)` – span of the knot
- `knot(float)` – knot

Returns  basis functions
Return type  list

`geomdl.helpers.basis_function_all(degree, knot_vector, span, knot)`
Finds all non-zero basis functions of all degrees from 0 up to the input degree for a single knot.

A slightly modified version of Algorithm A2.2 from The NURBS Book by Piegl & Tiller.

Parameters
- `degree(int)` – degree
- `knot_vector(list, tuple)` – knot vector
NURBS-Python Documentation

- **span** *(int)* – span of the knot
- **knot** *(float)* – knot

**Returns** basis functions

**Return type** *list*

```python
geomdl.helpers.basis_function_ders(degree, knot_vector, span, knot, order)
```

Finds derivatives of the basis functions for a single knot.

**Parameters**

- **degree** *(int)* – degree
- **knot_vector** *(list, tuple)* – knot vector
- **span** *(int)* – span of the knot
- **knot** *(float)* – knot
- **order** *(int)* – order of the derivative

**Returns** basis function derivatives

**Return type** *list*

```python
geomdl.helpers.basis_function_ders_one(degree, knot_vector, span, knot, order)
```

Finds the derivative of one basis functions for a single knot.

**Parameters**

- **degree** *(int)* – degree
- **knot_vector** *(list, tuple)* – knot vector
- **span** *(int)* – span of the knot
- **knot** *(float)* – knot
- **order** *(int)* – order of the derivative

**Returns** basis function derivatives values

**Return type** *list*

```python
geomdl.helpers.basis_function_one(degree, knot_vector, span, knot)
```

Computes the value of a basis function for a knot.

**Parameters**

- **degree** *(int)* – degree
- **knot_vector** *(list, tuple)* – knot vector
- **span** *(int)* – span of the knot
- **knot** *(float)* – knot

**Returns** basis function value

**Return type** *float*
geomdl.helpers.basis_functions (degree, knot_vector, spans, knots)
Computes the non-vanishing basis functions for a list of knots.

Parameters
- **degree** (int) – degree
- **knot_vector** (list, tuple) – knot vector
- **spans** (list, tuple) – spans
- **knots** (list, tuple) – knots

Returns basis functions
Return type list

geomdl.helpers.find_multiplicity (knot, knot_vector, **kwargs)
Finds knot multiplicity over the knot vector.

Parameters
- **knot** (float) – knot
- **knot_vector** (list, tuple) – knot vector

Returns multiplicity of the knot
Return type int

geomdl.helpers.find_span_binsearch (degree, knot_vector, num_ctrlpts, knot, **kwargs)
Finds the span of the knot over the input knot vector using binary search.

Implementation of Algorithm A2.1 from The NURBS Book by Piegl & Tiller.

The NURBS Book states that the knot span index always starts from zero, i.e. for a knot vector [0, 0, 1, 1]; if FindSpan returns 1, then the knot is between the interval [0, 1).

Parameters
- **degree** (int) – degree
- **knot_vector** (list, tuple) – knot vector
- **num_ctrlpts** (int) – number of control points
- **knot** (float) – knot

Returns span of the knot over the knot vector
Return type int

geomdl.helpers.find_span_linear (degree, knot_vector, num_ctrlpts, knot, **kwargs)
Finds the span of a single knot over the knot vector using linear search.

Alternative implementation for the Algorithm A2.1 from The NURBS Book by Piegl & Tiller.

Parameters
- **degree** (int) – degree
- **knot_vector** (list, tuple) – knot vector
- **num_ctrlpts** (int) – number of control points
- **knot** (float) – knot

Returns span of the knot over the knot vector
Return type int
geomdl.helpers.find_spans\(degree, \text{knot\_vector}, \text{num\_ctrlpts}, \text{knots}, \text{func}=<function \text{find\_span\_linear}>)

Finds spans of a list of knots over the knot vector.

**Parameters**

- \text{degree (int)} – degree
- \text{knot\_vector (list, tuple)} – knot vector
- \text{num\_ctrlpts (int)} – number of control points
- \text{knots (list, tuple)} – list of knots
- \text{func} – function to evaluate span finding operation

**Returns** list of spans

**Return type** list

### 13.7 Converters

geomdl.convert.bspline\_to\_nurbs\(obj\)

Converts B-Spline objects to NURBS objects.

The intended functionality is converting B-Spline curves and surfaces to NURBS curves and surfaces, respectively. Therefore, the inputs should be \text{BSpline.Curve} or \text{BSpline.Surface}. Otherwise, the function will raise a TypeError.

**Parameters** \text{obj} (BSpline.Curve, BSpline.Surface) – B-Spline object

**Returns** NURBS object

**Return type** \text{NURBS.Curve, NURBS.Surface}

**Raises** TypeError

### 13.8 Compatibility Module

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

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 \text{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**

- \text{ctrlpts (list, tuple)} – unweighted control points
- \text{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 (row length)
- `size_v (int)` – size in v-direction (column length)

**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

geomdl.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

geomdl.compatibility.flip_ctrlpts_u(ctrlpts, size_u, size_v)
Flips a list of 1-dimensional control points from u-row order to v-row order.

**u-row order**: each row corresponds to a list of u values

**v-row order**: each row corresponds to a list of v values

**Parameters**
- `ctrlpts (list, tuple)` – control points in u-row order
- `size_u (int)` – size in u-direction
- `size_v (int)` – size in v-direction

**Returns** control points in v-row order

**Return type** list

geomdl.compatibility.generate_ctrlpts2d_weights(ctrlpts2d)
Generates unweighted control points from weighted ones in 2-D.

This function

1. Takes in 2-D control points list whose coordinates are organized like \((x*w, y*w, z*w, w)\)
2. Converts the input control points list into (x, y, z, w) format
3. Returns the result

**Parameters**

`ctrlpts2d(list)` – 2-D control points (P)

**Returns**

2-D weighted control points (Pw)

**Return type**

list

### geomdl.compatibility.generate_ctrlpts2d_weights_file

Generates unweighted control points from weighted ones in 2-D.

1. Takes in 2-D control points list whose coordinates are organized like (x*w, y*w, z*w, w)
2. Converts the input control points list into (x, y, z, w) format
3. Saves the result to a file

**Parameters**

- `file_in (str)` – name of the input file (to be read)
- `file_out (str)` – name of the output file (to be saved)

**Raises**

`IOError` – an error occurred reading or writing the file

### geomdl.compatibility.generate_ctrlpts_weights

Generates unweighted control points from weighted ones in 1-D.

This function

1. Takes in 1-D control points list whose coordinates are organized in (x*w, y*w, z*w, w) format
2. Converts the input control points list into (x, y, z, w) format
3. Returns the result

**Parameters**

`ctrlpts (list)` – 1-D control points (P)

**Returns**

1-D weighted control points (Pw)

**Return type**

list

### geomdl.compatibility.generate_ctrlptsw

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 (Pw)

**Return type**

list
geomdl.compatibility.generate_ctrlptsw2d(ctrlpts2d)
Generates weighted control points from unweighted ones in 2-D.

This function
1. Takes in a 2D control points list whose coordinates are organized in (x, y, z, w) format
2. converts into (x*w, y*w, z*w, w) format
3. Returns the result

Therefore, the returned list could be a direct input of the NURBS.Surface class.

Parameters ctrlpts2d (list) – 2-D control points (P)
Returns 2-D weighted control points (Pw)
Return type list

geomdl.compatibility.generate_ctrlptsw2d_file(file_in=", 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

13.9 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.
13.9.1 Grid

class geomdl.CPGen.Grid(size_x, size_y)
    Bases: object
    
    Simple grid generator to use with B-Spline surfaces.
    This class stores grid points in \([x, y, z]\) format.

    Note: Additional details on the file formats can be found in the documentation.

    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 in-
    stance:

    ```python
    testgrid = Grid(5, 10)  # generates a 5x10 rectangle
    testgrid.generate(4, 4)  # splits the rectangle into 2x2 pieces
    testgrid.bumps(100)  # impossible, it will return an error message
    testgrid.bumps(1)  # You will get a bump at the center of the generated grid
    ```

    This method accepts the following keyword arguments:
    • bump_height: z-value of the generated bumps on the grid. Default: 5.0
    • base_extent: extension of the hill base from its center in terms of grid points. Default: 2
    • base_adjust: padding between the bases of the hills. Default: 0

    Parameters num_bumps(int) – number of bumps (i.e. hills) to be generated on the 2D grid

generate(num_u, num_v)
    Generates grid using the input division parameters.

    Parameters
    • num_u(int) – number of divisions in x-direction
    • num_v(int) – number of divisions in y-direction

grid
    The generated grid.

    Getter Gets the 2-dimensional list of points in \([u][v]\) format

reset()
    Resets the grid to its initial state.
rotate_x (angle=0.0)
Rotates the grid about the x-axis.

Parameters
angle (float) – angle of rotation about the x-axis

rotate_y (angle=0.0)
Rotates the grid about the y-axis.

Parameters
angle (float) – angle of rotation about the y-axis

rotate_z (angle=0.0)
Rotates the grid about the z-axis.

Parameters
angle (float) – angle of rotation about the z-axis

save (filename='grid.txt')
Saves the generated grid to a text file.

Parameters
filename (str) – File name to be saved

Raises
IOError – an error occurred writing the file

translate (pos=(0.0, 0.0, 0.0))
Translates the grid origin to the input position.

The origin is initially (0, 0, 0) and always represents the bottom left corner of the 2-dimensional grid.

Parameters
pos (list) – new origin point

13.9.2 Weighted Grid

class geomdl.CPGen.GridWeighted(size_x, size_y)
Bases: geomdl.CPGen.Grid
Simple grid generator to use with NURBS surfaces.
This class stores grid points in [x*w, y*w, z*w, w] format.

Note: Additional details for the file formats can be found in the documentation.

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
testgrid = Grid(5, 10)  # generates a 5x10 rectangle
testgrid.generate(4, 4)  # splits the rectangle into 2x2 pieces
testgrid.bumps(100)  # impossible, it will return an error message
testgrid.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

```python
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**
The generated grid with weighted points.

- **Getter** Gets the 2-dimensional list of weighted points in $[u][v]$ format

```python
reset()
```
Resets the grid to its initial state.

```python
rotate_x(angle=0.0)
```
Rotates the grid about the x-axis.

**Parameters**

- **angle** (**float**) – angle of rotation about the x-axis

```python
rotate_y(angle=0.0)
```
Rotates the grid about the y-axis.

**Parameters**

- **angle** (**float**) – angle of rotation about the y-axis

```python
rotate_z(angle=0.0)
```
Rotates the grid about the z-axis.

**Parameters**

- **angle** (**float**) – angle of rotation about the z-axis

```python
save(filename='grid.txt')
```
Saves the generated grid to a text file.

**Parameters**

- **filename** (**str**) – File name to be saved

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

```python
translate(pos=(0.0, 0.0, 0.0))
```
Translates the grid origin to the input position.

The origin is initially $(0, 0, 0)$ and always represents the bottom left corner of the 2-dimensional grid.

**Parameters**

- **pos** (**list**) – new origin point

```python
weight
```
Weight $(w)$ component of the points.

- **Getter** Gets the weight
13.10 Surface and Curve Containers

This module provides curve and surface containers which could be created

- As a result of a geometric operation, such as splitting
- As a result of file import, e.g. reading a file or a set of files containing multiple surfaces
- For advanced post-processing, such as visualization or file export

This module works with BSpline and NURBS modules and it contains the following classes:

- Multi abstract base class for all containers
- MultiCurve curve container class
- MultiSurface surface container class

```python
class geomdl.Abstract.Multi(*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.

    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

    render()
        Abstract method for rendering plots using the visualization component.

    vis
        Visualization component.

        Getter
        -------
        Gets the visualization component

        Setter
        -------
        Sets the visualization component

        Type
        -----
        float
```

```python
class geomdl.Multi.MultiCurve(*args, **kwargs)
    Bases: geomdl.Abstract.Multi

    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.

    Rendering depends on the visualization instance, e.g. if you are using VisMPL module, you can visualize a 3D
curve using a VisCurve2D instance but you cannot visualize a 2D curve with a VisCurve3D instance.

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

Renders the curve the 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
- `filename`: saves the plot with the input name
- `plot`: a flag to control displaying the plot window. Default is 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.

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

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

### vis

Visualization component.

- **Getter** Gets the visualization component
- **Setter** Sets the visualization component
- **Type** float

**class** geomdl.Multi.MultiSurface(**args**, **kwargs**)

**Bases:** geomdl.Abstract.Multi

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.

**add**(element)

Add 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

**render(**kwargs**)

Renders the surface the 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 grids
• **evalcolor**: sets the color of the surface
• **filename**: saves the plot with the input name
• **plot**: a flag to control displaying the plot window. Default is True.
• **colormap**: sets the colormap of the surfaces

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.
Sample size defines the number of evaluated points to generate on u- and v-direction.

  Getter  Gets sample size
  Setter  Sets sample size
  Type    int

**sample_size_u**
Sample size for the u-direction.
Sample size defines the number of evaluated points to generate on the defined direction.

  Getter  Gets sample size
  Setter  Sets sample size
  Type    int

**sample_size_v**
Sample size for the v-direction.
Sample size defines the number of evaluated points to generate on the defined direction.

  Getter  Gets sample size
  Setter  Sets sample size
  Type    int

**vis**
Visualization component.

  Getter  Gets the visualization component
  Setter  Sets the visualization component
  Type    float
13.11 Exchange Module

This module allows users to export NURBS surfaces in common CAD exchange formats.

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

Parameters

• `obj` (Abstract.Curve or Abstract.Surface) – input curve or surface
• `file_name` (str) – name of the output file

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

```python
geomdl.exchange.export_csv(obj, file_name, point_type='evalpts')
Exports control points or evaluated points as a CSV file.

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
```

```python
geomdl.exchange.export_obj(surf_in, file_name, **kwargs)
Exports surface(s) as a .obj file.

Keyword Arguments:

• `vertex_spacing`: size of the triangle edge in terms of points sampled on the surface. Default: 2

Parameters

• `surf_in` (Abstract.Surface or Multi.MultiSurface) – surface or surfaces to be saved
• `file_name` (str) – name of the output file

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

```python
geomdl.exchange.export_off(surf_in, 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

Parameters

• `surf_in` (Abstract.Surface or Multi.MultiSurface) – surface or surfaces to be saved
• `file_name` (str) – name of the output file

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

```python
geomdl.exchange.export_smesh(surf_in, file_name, **kwargs)
Exports surface(s) as .smesh files.

Parameters
```
**surf_in** (Abstract.Surface or Multi.MultiSurface) – surface or surfaces to be saved

**file_name** (str) – name of the output file

Raises IOError – an error occurred writing the file

**geomdl.exchange.export_stl** *(surf_in, 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

**Parameters**

- **surf_in** (Abstract.Surface or Multi.MultiSurface) – surface or surfaces to be saved
- **file_name** (str) – name of the output file

Raises IOError – an error occurred writing the file

**geomdl.exchange.export_txt** *(obj, file_name, two_dimensional=False, **kwargs)*

Saves control points to 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.export_vtk** *(obj, file_name, point_type='evalpts' )*  

Exports control points or evaluated points as a VTK file (legacy format).

Please see the following document for details: 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

**geomdl.exchange.import_cfg** *(file_name)*

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

• ImportError – cannot find ‘libconf’ module
• IOError – an error occurred writing the file

`geomdl.exchange.import_smesh(file)`
Generates NURBS surface(s) from 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 smesh files or a single smesh file

Returns NURBS surface(s)

Return type NURBS.Surface or Multi.MultiSurface

Raises IOError – an error occurred reading the file

`geomdl.exchange.import_txt(file_name, two_dimensional=False, **kwargs)`
Reads control points from a text file and generates a 1-D list of control points.

The following code examples illustrate importing different types of text files for curves and surfaces:

```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)
```

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=",
                                                  ")
```

Chapter 13. Core Modules
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

### 13.12 Tessellation

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

```python
from geomdl import NURBS
from geomdl import tessellate

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

# Set tessellation algorithm (you can use another algorithm)
surf.tessellator = tessellate.TriangularTessellate()

# Tessellate surface
surf.tessellate()
```

NURBS-Python uses `TriangularTessellate` class for surface tessellation by default.

#### 13.12.1 Abstract Tessellation

```python
class geomdl.Abstract.Tessellate(**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
    Setter Sets the tessellation arguments

    reset()
    Clears stored vertices and triangles.

    tessellate(points, size_u, size_v, **kwargs)
    Abstract method for the implementation of the tessellation algorithm.

    This algorithm should update `vertices` and `triangles` properties.

    Parameters
```
• **points** – 1-dimensional array of surface points
• **size_u** – number of surface points on the u-direction
• **size_v** – number of surface points on the v-direction

**triangles**
Triangle objects generated after tessellation.

**Getter** Gets the triangles

**vertices**
Vertex objects generated after tessellation.

**Getter** Gets the vertices

### 13.12.2 Included Tessellation Options

The following tessellation classes are included in NURBS-Python.

```python
class geomdl.tessellate.TriangularTessellate(**kwargs)
    Bases: geomdl.Abstract.Tessellate

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

**reset**()
Clears stored vertices and triangles.

**tessellate**(points, size_u, size_v, **kwargs)
Applies triangular tessellation.

**Parameters**
• **points**(list, tuple) – points to be triangulated
• **size_u**(int) – number of points on the u-direction
• **size_v**(int) – number of points on the v-direction

**triangles**
Triangle objects generated after tessellation.

**Getter** Gets the triangles

**vertices**
Vertex objects generated after tessellation.

**Getter** Gets the vertices
13.13 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`
- `Face`
- `Body`

```python
class geomdl.elements.AbstractEntity
    Bases: object

    Abstract base class for all geometric entities.

    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.Body
    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

class geomdl.elements.Face
    Bases: geomdl.elements.AbstractEntity

    Representation of Face entity which is composed from triangles.

    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

```python
class geomdl.elements.Triangle
    Bases: geomdl.elements.AbstractEntity
```
Triangle entity which represents a triangle composed from 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.Vertex
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
NURBS-Python Documentation

**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

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.

The **Operations** module contains specialized geometrical operations that can be directly applied to the B-Spline and NURBS shapes.
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:

### 14.1 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. `VisAbstract` and `VisAbstractSurf` classes provide the required abstract base classes for all visualization components.

#### 14.1.1 Abstract Visualization

```python
class geomdl.Abstract.VisAbstract(config=None)
    Bases: object

    Visualization abstract class
    Uses Python's Abstract Base Class implementation to define a base for all common visualization options in NURBS-Python package.

    add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
    Adds points sets to the visualization instance for plotting.

    Parameters
    * ptsarr(list, tuple) – control, curve or surface points
    * size(int, tuple, list) – size in all directions, e.g. in u- and v-directions
    * name(str) – name of the point on the legend
    * color(str) – color of the point on the legend
```
• **plot_type** (int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

`clear()`

Clears the points, colors and names lists.

`render(**kwargs)`

Abstract method for rendering plots of the point sets.

This method must be implemented in all subclasses of VisAbstract class.

### 14.1.2 Abstract Visualization for Surfaces

```python
class geomdl.Abstract.VisAbstractSurf (config=None)
```

Visualization abstract class for surfaces

`Bases: geomdl.Abstract.VisAbstract`

Implements VisAbstract class and also uses Python’s Abstract Base Class implementation to define a base for surface visualization options in NURBS-Python package.

`add(ptsarr=(), size=0, name=None, color=None, plot_type=0)`

Adds points sets to the visualization instance for plotting.

**Parameters**

- **ptsarr** (list, tuple) – control, curve or surface points
- **size** (int, tuple, list) – size in all directions, e.g. in u- and v-directions
- **name** (str) – name of the point on the legend
- **color** (str) – color of the point on the legend
- **plot_type** (int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

`clear()`

Clears the points, colors and names lists.

`plot_types`

Plot types

**Getter** Gets the plot types

**Type** tuple

`render(**kwargs)`

Abstract method for rendering plots of the point sets.

This method must be implemented in all subclasses of VisAbstractSurf class.

```python
set_ctrlpts_offset (offset_value)
```

Sets an offset for the control points grid plot.

**Parameters** `offset_value` (float) – offset value

### 14.1.3 Abstract Visualization Configuration

```python
class geomdl.Abstract.VisConfigAbstract (**kwargs)
```

Visualization configuration abstract class
Uses Python’s *Abstract Base Class* implementation to define a base for all visualization configurations in NURBS-Python package.

## 14.2 Matplotlib Implementation

This module provides Matplotlib visualization implementation.

```python
class geomdl.visualization.VisMPL.VisConfig(**kwargs)
Bases: geomdl.Abstract.VisConfigAbstract
```

Configuration class for Matplotlib visualization module.

This class is only required when you prefer to change the default plotting behavior, such as hiding control points plot or legend. By default, the following variables and their default values are used in all `VisMPL` visualization classes.

- **ctrlpts** (True or False, default: True): Enables/Disables control points polygon/grid plot in the figure
- **legend** (True or False): Enables/Disables legend in the figure
- **axes** (True or False): Enables/Disables axes and grid in the figure
- **trims** (True or False): Enables/Disables trim curves display in the figure
- **figure_size** (list, default: [10.67, 8]): Size of the figure in (x, y)
- **figure_dpi** (int, default: 96): Resolution of the figure in DPI
- **trim_size** (int, default: 20): Size of the trim curves

The following example illustrates the usage of the configuration class.

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

# Skipping degree, knot vector and control points assignments

# Create a visualization configuration instance with no legend, no axes and set the resolution to 120 dpi
vis_config = VisMPL.VisConfig(legend=False, axes=False, figure_dpi=120)

# Create a visualization method instance using the configuration above
vis_obj = VisMPL.VisCurve2D(vis_config)

# Set the visualization method of the curve object
curve.vis = vis_obj

# Plot the curve
curve.render()
```

Please refer to the Examples Repository for more details.

```python
static save_figure_as(fig, filename)
```

Saves the figure as a file.

Parameters

- **fig** – a Matplotlib figure instance
- **filename** – file name to save

## 14.2. Matplotlib Implementation
**static set_axes_equal**(*ax*)
Sets equal aspect ratio across the three axes of a 3D plot.
Contributed by Xuefeng Zhao.

**Parameters**

- `ax` – a Matplotlib axis, e.g., as output from `plt.gca()`.

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

* Bases: `geomdl.Abstract.VisAbstract`

Matplotlib visualization module for 2D curves

*add*(*ptsarr=(), size=0, name=None, color=None, plot_type=0*)

Adds points sets to the visualization instance for plotting.

**Parameters**

- `ptsarr` (*list, tuple*) – control, curve or surface points
- `size` (*int, tuple, list*) – size in all directions, e.g. in u- and v-directions
- `name` (*str*) – name of the point on the legend
- `color` (*str*) – color of the point on the legend
- `plot_type` (*int*) – type of the plot, control points (type = 1) or evaluated points (type = 0)

*clear*()

Clears the points, colors and names lists.

*render*(***kwargs*)

Plots the 2D curve and the control points polygon.

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

* Bases: `geomdl.Abstract.VisAbstract`

Matplotlib visualization module for 3D curves.

*add*(*ptsarr=(), size=0, name=None, color=None, plot_type=0*)

Adds points sets to the visualization instance for plotting.

**Parameters**

- `ptsarr` (*list, tuple*) – control, curve or surface points
- `size` (*int, tuple, list*) – size in all directions, e.g. in u- and v-directions
- `name` (*str*) – name of the point on the legend
- `color` (*str*) – color of the point on the legend
- `plot_type` (*int*) – type of the plot, control points (type = 1) or evaluated points (type = 0)

*clear*()

Clears the points, colors and names lists.

*render*(***kwargs*)

Plots the 3D curve and the control points polygon.

**class** `geomdl.visualization.VisMPL.VisSurfScatter`(*config=<geomdl.visualization.VisMPL.VisConfig object>*)


134 Chapter 14. Visualization Modules
Matplotlib visualization module for surfaces.

Wireframe plot for the control points and scatter plot for the surface points.

```python
add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
```

Adds points sets to the visualization instance for plotting.

**Parameters**

- `ptsarr (list, tuple)` – control, curve or surface points
- `size (int, tuple, list)` – size in all directions, e.g. in u- and v-directions
- `name (str)` – name of the point on the legend
- `color (str)` – color of the point on the legend
- `plot_type (int)` – type of the plot, control points (type = 1) or evaluated points (type = 0)

```python
clear()
```

Clears the points, colors and names lists.

```python
plot_types
```

Plot types

**Getter** Gets the plot types

**Type** tuple

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

Plots the surface and the control points grid.

```python
set_ctrlpts_offset(offset_value)
```

Sets an offset for the control points grid plot.

**Parameters**

- `offset_value (float)` – offset value

**class** `geomdl.visualization.VisMPL.VisSurfTriangle(config=<geomdl.visualization.VisMPL.VisConfig object>)(...)`

**Bases**: `geomdl.Abstract.VisAbstractSurf`

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.

```python
add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
```

Adds points sets to the visualization instance for plotting.

**Parameters**

- `ptsarr (list, tuple)` – control, curve or surface points
- `size (int, tuple, list)` – size in all directions, e.g. in u- and v-directions
- `name (str)` – name of the point on the legend
- `color (str)` – color of the point on the legend
- `plot_type (int)` – type of the plot, control points (type = 1) or evaluated points (type = 0)

```python
clear()
```

Clears the points, colors and names lists.

```python
plot_types
```

Plot types
Getter  Gets the plot types

Type  tuple

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

set_ctrlpts_offset(offset_value)
Sets an offset for the control points grid plot.

Parameters offset_value (float) – offset value

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


Matplotlib visualization module for surfaces.

Scatter plot for the control points and wireframe plot for the surface points.

add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
Adds points sets to the visualization instance for plotting.

Parameters

• ptsarr (list, tuple) – control, curve or surface points

• size (int, tuple, list) – size in all directions, e.g. in u- and v-directions

• name (str) – name of the point on the legend

• color (str) – color of the point on the legend

• plot_type (int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

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

plot_types
Plot types

Getter  Gets the plot types

Type  tuple

render(**kwargs)
Plots the surface and the control points grid.

set_ctrlpts_offset(offset_value)
Sets an offset for the control points grid plot.

Parameters offset_value (float) – offset value

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


Matplotlib visualization module for surfaces.

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

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

```
add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
```

Adds points sets to the visualization instance for plotting.

### Parameters
- `ptsarr` *(list, tuple)* – control, curve or surface points
- `size` *(int, tuple, list)* – size in all directions, e.g. in u- and v-directions
- `name` *(str)* – name of the point on the legend
- `color` *(str)* – color of the point on the legend
- `plot_type` *(int)* – type of the plot, control points (type = 1) or evaluated points (type = 0)

```
clear()
```

Clears the points, colors and names lists.

```
plot_types
```

Plot types

**Getter** Gets the plot types

**Type** tuple

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

Plots the surface and the control points grid.

```
set_ctrlpts_offset(offset_value)
```

Sets an offset for the control points grid plot.

**Parameters** `offset_value` *(float)* – offset value

### 14.3 Plotly Implementation

This module provides Plotly visualization implementation.

```
class geomdl.visualization.VisPlotly.VisConfig(**kwargs)
Bases: geomdl.Abstract.VisConfigAbstract
```

Configuration class for Plotly visualization module.

This class is only required when you prefer to change the default plotting behavior, such as hiding control points plot or legend. By default, the following variables and their default values are used in all `VisPlotly` visualization classes.

- `ctrlpts` *(True or False, default: True)*: Enables/Disables control points polygon/grid plot on the figure
- `legend` *(True or False)*: Enables/Disables legend on the figure
- `axes` *(True or False)*: Enables/Disables axes and grid on the figure
- `trims` *(True or False)*: Enables/Disables trim curves display in the figure
- `figure_size` *(list, default: [800, 600])*: Size of the figure in (x, y)
- `trim_size` *(int, default: 20)*: Size of the trim curves
- `linewidth` *(int, default: 2)*: thickness of the lines on the figure
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.Abstract.VisAbstract
Plotly visualization module for 2D curves.
add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
   Adds points sets to the visualization instance for plotting.
   Parameters
   • ptsarr(list, tuple) – control, curve or surface points
   • size(int, tuple, list) – size in all directions, e.g. in u- and v-directions
   • name(str) – name of the point on the legend
   • color(str) – color of the point on the legend
   • plot_type(int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

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

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

class geomdl.visualization.VisPlotly.VisCurve3D(config=<geomdl.visualization.VisPlotly.VisConfig object>)
Bases: geomdl.Abstract.VisAbstract
Plotly visualization module for 3D curves.
add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
   Adds points sets to the visualization instance for plotting.
   Parameters
   • ptsarr(list, tuple) – control, curve or surface points
   • size(int, tuple, list) – size in all directions, e.g. in u- and v-directions
```
• **name** (str) – name of the point on the legend
• **color** (str) – color of the point on the legend
• **plot_type** (int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

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

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

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

Plotly visualization module for surfaces.
Triangular mesh plot for the surface and wireframe plot for the control points grid.

add(ptsarr=(), size=0, name=None, color=None, plot_type=0)
Adds points sets to the visualization instance for plotting.

Parameters
• **ptsarr** (list, tuple) – control, curve or surface points
• **size** (int, tuple, list) – size in all directions, e.g. in u- and v-directions
• **name** (str) – name of the point on the legend
• **color** (str) – color of the point on the legend
• **plot_type** (int) – type of the plot, control points (type = 1) or evaluated points (type = 0)

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

plot_types
Plot types

Getter Gets the plot types

Type tuple

render(**kwargs)
Plots the surface and the control points grid.

set_ctrlpts_offset(offset_value)
Sets an offset for the control points grid plot.

Parameters offset_value (float) – offset value

The users are not limited with these visualization backends. For instance, control points and evaluated points can be exported via `export_csv()` or `export_vtk()` functions to plot with COTS software.
NURBS-Python comes with several optional modules. These modules might require installation of additional packages, might come with extra requirements or their API might change between NURBS-Python releases. Therefore, even though they are distributed with the package, due to these reasons they are considered as experimental. However, they are mature enough to be used in production environments.

15.1 Generating Common Shapes

NURBS-Python provides an experimental module for automatic generation of the most commonly used curves and surfaces.

15.1.1 Shapes Module

The shapes module provides functionality for automatic generation of the most commonly used geometric shapes.

2D Curves

geomdl.shapes.curve2d.full_circle(radius=1)
Generates a NURBS full circle from 9 control points.

- **Parameters** radius (int, float) — radius of the circle
- **Returns** a NURBS curve
- **Return type** NURBS.Curve

geomdl.shapes.curve2d.full_circle2(radius=1)
Generates a NURBS full circle from 7 control points.

- **Parameters** radius (int, float) — radius of the circle
- **Returns** a NURBS curve
Return type  *NURBS.Curve*

**Surfaces**

```python
geomdl.shapes.surface.cylinder(radius=1, height=1)
```

Generates a NURBS cylindrical surface.

The cylindrical surface example is kindly contributed by John-Eric Dufour.

**Parameters**

- `radius (int, float)` – radius of the cylinder
- `height (int, float)` – height of the cylinder

**Returns** a NURBS surface

Return type  *NURBS.Surface*

NURBS-Python is developed by Onur Rauf Bingol and all the code released under the MIT License.
Python Module Index

C
compatibility (Unix, Windows), 112
convert (Unix, Windows), 112
curve2d (Unix, Windows), 141

e
elements (Unix, Windows), 127
evaluators (Unix, Windows), 95
exchange (Unix, Windows), 122

g
geomdl.compatibility, 112
geomdl.convert, 112
geomdl.elements, 127
geomdl.evaluators, 95
geomdl.exchange, 122
geomdl.helpers, 109
geomdl.Multi, 119
geomdl.operations, 100
geomdl.shapes.curve2d, 141
geomdl.shapes.surface, 142
geomdl.tessellate, 126
geomdl.utilities, 102
geomdl.visualization.VisMPL, 133
geomdl.visualization.VisPlotly, 137

h
helpers (Unix, Windows), 109

m
Multi (Unix, Windows), 119

o
operations (Unix, Windows), 100

s
surface (Unix, Windows), 142

t
tessellate (Unix, Windows), 126

U
utilities (Unix, Windows), 102

V
VisMPL (Unix, Windows), 133
VisPlotly (Unix, Windows), 137
color_generator() (in module geomdl.utilities), 103
combine_ctrlpts_weights() (in module geomdl.compatibility), 112
compatibility (module), 112
convert (module), 112
ctrlpts (geomdl.Abstract.Curve attribute), 58
ctrlpts (geomdl.BSpline.Curve attribute), 67
ctrlpts (geomdl.BSpline.Surface attribute), 72
ctrlpts (geomdl.NURBS.Curve attribute), 81
ctrlpts (geomdl.NURBS.Surface attribute), 86
ctrlpts2d (geomdl.Abstract.Surface attribute), 61
ctrlpts2d (geomdl.BSpline.Surface attribute), 72
ctrlpts2d (geomdl.NURBS.Surface attribute), 86
ctrlpts_size_u (geomdl.Abstract.Surface attribute), 61
ctrlpts_size_u (geomdl.BSpline.Surface attribute), 73
ctrlpts_size_u (geomdl.NURBS.Surface attribute), 87
ctrlpts_size_v (geomdl.Abstract.Surface attribute), 61
ctrlpts_size_v (geomdl.BSpline.Surface attribute), 73
ctrlpts_size_v (geomdl.NURBS.Surface attribute), 87
ctrlptsw (geomdl.NURBS.Curve attribute), 81
ctrlptsw (geomdl.NURBS.Surface attribute), 87
Curve (class in geomdl.Abstract), 57
Curve (class in geomdl.BSpline), 66
Curve (class in geomdl.NURBS), 80
curve2d (module), 141
CurveEvaluator (class in geomdl.Abstract), 95
CurveEvaluator (class in geomdl.evaluators), 95
CurveEvaluator2 (class in geomdl.evaluators), 96
curvet() (geomdl.BSpline.Curve method), 67
curvet() (geomdl.NURBS.Curve method), 81
cylinder() (in module geomdl.shapes.surface), 142

data (geomdl.elements.Vertex attribute), 129
decompose_curve() (in module geomdl.operations), 100
decompose_surface() (in module geomdl.operations), 100
degree (geomdl.Abstract.Curve attribute), 58
degree (geomdl.BSpline.Curve attribute), 67
degree (geomdl.NURBS.Curve attribute), 81
degree_u (geomdl.Abstract.Surface attribute), 61
degree_u (geomdl.BSpline.Surface attribute), 73
degree_u (geomdl.NURBS.Surface attribute), 88
degree_v (geomdl.Abstract.Surface attribute), 62
degree_v (geomdl.BSpline.Surface attribute), 73
degree_v (geomdl.NURBS.Surface attribute), 87
delta (geomdl.Abstract.Curve attribute), 58
delta (geomdl.Abstract.Surface attribute), 62
delta (geomdl.BSpline.Curve attribute), 67
delta (geomdl.BSpline.Surface attribute), 73
delta (geomdl.NURBS.Curve attribute), 81
delta (geomdl.NURBS.Surface attribute), 88
delta_u (geomdl.Abstract.Surface attribute), 62
delta_u (geomdl.BSpline.Surface attribute), 74
delta_u (geomdl.NURBS.Surface attribute), 88
delta_v (geomdl.Abstract.Surface attribute), 62
delta_v (geomdl.BSpline.Surface attribute), 74
delta_v (geomdl.NURBS.Surface attribute), 88
derivatives() (geomdl.Abstract.Curve method), 58
derivatives() (geomdl.Abstract.Evaluator method), 95
derivatives() (geomdl.Abstract.Surface method), 62
derivatives() (geomdl.BSpline.Curve method), 67
derivatives() (geomdl.BSpline.Surface method), 74
derivatives() (geomdl.evaluators.CurveEvaluator method), 96
derivatives() (geomdl.evaluators.CurveEvaluator2 method), 96
derivatives() (geomdl.evaluators.NURBS.CurveEvaluator method), 97
derivatives() (geomdl.evaluators.NURBS.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator2 method), 99
derivatives() (geomdl.NURBS.Curve method), 81
derivatives() (geomdl.NURBS.Surface method), 89
derivatives_ctrlpts() (geomdl.evaluators.CurveEvaluator2 static method), 96
derivatives() (geomdl.evaluators.NURBS.CurveEvaluator method), 97
derivatives() (geomdl.evaluators.NURBS.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator2 method), 99
derivatives() (geomdl.evaluators.SurfaceEvaluator2 method), 96
derivatives() (geomdl.evaluators.NURBS.CurveEvaluator method), 97
derivatives() (geomdl.evaluators.NURBS.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator method), 98
derivatives() (geomdl.evaluators.SurfaceEvaluator2 method), 99
dimension (geomdl.Abstract.Curve attribute), 58
dimension (geomdl.Abstract.Surface attribute), 63
dimension (geomdl.BSpline.Curve attribute), 67
dimension (geomdl.BSpline.Surface attribute), 74
dimension (geomdl.NURBS.Curve attribute), 82
dimension (geomdl.NURBS.Surface attribute), 89

dimensional (geomdl.evaluators.CurveEvaluator method), 96
dimensional() (geomdl.evaluators.NURBS.CurveEvaluator method), 97
dimensional() (geomdl.evaluators.NURBS.SurfaceEvaluator method), 98
dimensional() (geomdl.evaluators.SurfaceEvaluator method), 98
dimensional() (geomdl.evaluators.SurfaceEvaluator2 method), 99
edges (geomdl.elements.Triangle attribute), 128
elements (module), 127
evalpts (geomdl.Abstract.Curve attribute), 58
evalpts (geomdl.Abstract.Surface attribute), 63
evalpts (geomdl.BSpline.Curve attribute), 68
evalpts (geomdl.BSpline.Surface attribute), 74
evalpts (geomdl.NURBS.Curve attribute), 82
evalpts (geomdl.NURBS.Surface attribute), 89
evaluate() (geomdl.Abstract.Curve method), 58
evaluate() (geomdl.Abstract.Evaluator method), 95
evaluate() (geomdl.Abstract.Surface method), 63
evaluate() (geomdl.BSpline.Curve method), 68
evaluate() (geomdl.BSpline.Surface method), 75
evaluate() (geomdl.evaluators.CurveEvaluator method), 96
evaluate() (geomdl.evaluators.CurveEvaluator2 method), 97
evaluate() (geomdl.evaluators.NURBSCurveEvaluator method), 97
evaluate() (geomdl.evaluators.NURBSSurfaceEvaluator method), 98
evaluate() (geomdl.evaluators.SurfaceEvaluator method), 98
evaluate() (geomdl.evaluators.SurfaceEvaluator2 method), 99
evaluate() (geomdl.NURBS.Curve method), 82
evaluate() (geomdl.NURBS.Surface method), 89
evaluate_bounding_box() (in module geomdl.utilities), 103
evaluate_list() (geomdl.Abstract.Curve method), 58
evaluate_list() (geomdl.Abstract.Surface method), 63
evaluate_list() (geomdl.BSpline.Curve method), 68
evaluate_list() (geomdl.BSpline.Surface method), 75
evaluate_list() (geomdl.NURBS.Curve method), 82
evaluate_list() (geomdl.NURBS.Surface method), 90
evaluate_single() (geomdl.Abstract.Curve method), 59
evaluate_single() (geomdl.Abstract.Evaluator method), 95
evaluate_single() (geomdl.Abstract.Surface method), 63
evaluate_single() (geomdl.BSpline.Curve method), 68
evaluate_single() (geomdl.BSpline.Surface method), 75
evaluate_single() (geomdl.evaluators.CurveEvaluator method), 96
evaluate_single() (geomdl.evaluators.CurveEvaluator2 method), 97
evaluate_single() (geomdl.evaluators.NURBSCurveEvaluator method), 97
evaluate_single() (geomdl.evaluators.NURBSSurfaceEvaluator method), 98
evaluate_single() (geomdl.evaluators.SurfaceEvaluator method), 98
evaluate_single() (geomdl.evaluators.SurfaceEvaluator2 method), 99
evaluate_single() (geomdl.NURBS.Curve method), 82
evaluate_single() (geomdl.NURBS.Surface method), 90
Evaluator (class in geomdl.Abstract), 95
evaluator (geomdl.Abstract.Curve attribute), 59
evaluator (geomdl.Abstract.Surface attribute), 63
evaluator (geomdl.BSpline.Curve attribute), 68
evaluator (geomdl.BSpline.Surface attribute), 75
evaluator (geomdl.NURBS.Curve attribute), 82
evaluator (geomdl.NURBS.Surface attribute), 90
evaluators (module), 95
export_cfg() (in module geomdl.exchange), 122
export_csv() (in module geomdl.exchange), 122
export_obj() (in module geomdl.exchange), 122
export_off() (in module geomdl.exchange), 122
export_smesh() (in module geomdl.exchange), 122
export_stl() (in module geomdl.exchange), 123
export_txt() (in module geomdl.exchange), 123
export_vtk() (in module geomdl.exchange), 123

F
Face (class in geomdl.elements), 127
faces (geomdl.elements.Body attribute), 127
find_ctrlpts() (in module geomdl.operations), 100
find_multiplicity() (in module geomdl.helpers), 111
find_span_binsearch() (in module geomdl.helpers), 111
find_span_linear() (in module geomdl.helpers), 111
find_spans() (in module geomdl.helpers), 111
flip_ctrlpts() (in module geomdl.compatibility), 112
flip_ctrlpts2d() (in module geomdl.compatibility), 113
flip_ctrlpts2d_file() (in module geomdl.compatibility), 113
flip_ctrlpts_u() (in module geomdl.compatibility), 113
frange() (in module geomdl.utilities), 103
full_circle() (in module geomdl.shapes.curve2d), 141
full_circle2() (in module geomdl.shapes.curve2d), 141

G
generate() (geomdl.CPGen.Grid method), 116
generate() (geomdl.CPGen.GridWeighted method), 118
generate_ctrlpts2d_weights() (in module geomdl.compatibility), 113
generate_ctrlpts2d_weights_file() (in module geomdl.compatibility), 114
generate_ctrlpts_weights() (in module geomdl.compatibility), 114
generate_ctrlptsw() (in module geomdl.compatibility), 114
generate_ctrlptsw2d() (in module geomdl.compatibility), 114
generate_ctrlptsw2d_file() (in module geomdl.compatibility), 115
generate_knot_vector() (in module geomdl.utilities), 103
geomdl.compatibility (module), 112
geomdl.convert (module), 112
geomdl.elements (module), 127
sample_size_v (geomdl.NURBS.Surface attribute), 92
save() (geomdl.BSpline.Curve method), 70
save() (geomdl.BSpline.Surface method), 78
save() (geomdl.CPGen.Grid method), 117
save() (geomdl.CPGen.GridWeighted method), 118
save() (geomdl.NURBS.Curve method), 84
save() (geomdl.NURBS.Surface method), 93
save_figure_as() (geomdl.visualization.VisMPL.VisConfig
static method), 133
separate_ctrlpts_weights() (in module geomdl.compatibility), 115
set_axes_equal() (geomdl.visualization.VisMPL.VisConfig
static method), 133
set_ctrlpts() (geomdl.Abstract.Curve method), 60
set_ctrlpts() (geomdl.Abstract.Surface method), 65
set_ctrlpts() (geomdl.BSpline.Curve method), 70
set_ctrlpts() (geomdl.BSpline.Surface method), 78
set_ctrlpts() (geomdl.NURBS.Curve method), 84
set_ctrlpts() (geomdl.NURBS.Surface method), 93
set_ctrlpts_offset() (geomdl.Abstract.VisAbstractSurf
method), 132
set_ctrlpts_offset() (geomdl.visualization.VisMPL.VisSurface
method), 137
set_ctrlpts_offset() (geomdl.visualization.VisMPL.VisSurfScatter
method), 135
set_ctrlpts_offset() (geomdl.visualization.VisMPL.VisSurfTriangle
method), 136
set_ctrlpts_offset() (geomdl.visualization.VisMPL.VisSurfWireframe
method), 136
set_ctrlpts_offset() (geomdl.visualization.VisPlotly.VisSurface
method), 139
split_curve() (in module geomdl.operations), 101
split_surface_u() (in module geomdl.operations), 101
split_surface_v() (in module geomdl.operations), 101
Surface (class in geomdl.Abstract), 61
Surface (class in geomdl.BSpline), 71
Surface (class in geomdl.NURBS), 86
surface (module), 142
SurfaceEvaluator (class in geomdl.Abstract), 95
SurfaceEvaluator (class in geomdl.evaluators), 98
SurfaceEvaluator2 (class in geomdl.evaluators), 99
surfpt() (geomdl.BSpline.Surface method), 79
surfpt() (geomdl.NURBS.Surface method), 93
tangent() (geomdl.BSpline.Curve method), 71
tangent() (geomdl.BSpline.Surface method), 79
tangent() (geomdl.NURBS.Curve method), 85
tangent() (geomdl.NURBS.Surface method), 93
tangent() (in module geomdl.operations), 101
Tessellate (class in geomdl.Abstract), 125
tessellate (module), 126
tessellate() (geomdl.Abstract.Surface method), 65
tessellate() (geomdl.Abstract.Tessellate method), 125
tessellate() (geomdl.BSpline.Surface method), 79
tessellate() (geomdl.NURBS.Surface method), 94
tessellate() (geomdl.tessellate.TriangularTessellate
method), 126
tessellator (geomdl.Abstract.Surface attribute), 65
tessellator (geomdl.BSpline.Surface attribute), 79
tessellator (geomdl.NURBS.Surface attribute), 94
translate() (geomdl.CPGen.Grid method), 117
translate() (geomdl.CPGen.GridWeighted method), 118
translate() (in module geomdl.operations), 102
transpose() (geomdl.BSpline.Surface method), 79
transpose() (geomdl.NURBS.Surface method), 94
Triangle (class in geomdl.elements), 128
triangle_center() (in module geomdl.utilities), 107
triangle_normal() (in module geomdl.utilities), 107
triangles (geomdl.Abstract.Tessellate attribute), 126
triangles (geomdl.elements.Face attribute), 128
triangles (geomdl.tessellate.TriangularTessellate attribute), 126
TriangularTessellate (class in geomdl.tessellate), 126
trims (geomdl.Abstract.Surface attribute), 66
trims (geomdl.BSpline.Surface attribute), 79
trims (geomdl.NURBS.Surface attribute), 94
U
u (geomdl.elements.Vertex attribute), 129
utilities (module), 102
uv (geomdl.elements.Vertex attribute), 130
V
v (geomdl.elements.Vertex attribute), 130
vector_angle_between() (in module geomdl.utilities), 107
vector_cross() (in module geomdl.utilities), 107
vector_dot() (in module geomdl.utilities), 107
vector_generate() (in module geomdl.utilities), 108
vector_is_zero() (in module geomdl.utilities), 108
vector_magnitude() (in module geomdl.utilities), 108
vector_mean() (in module geomdl.utilities), 108
vector_multiply() (in module geomdl.utilities), 109
vector_normalize() (in module geomdl.utilities), 109
Vertex (class in geomdl.elements), 129
vertex_ids (geomdl.elements.Triangle attribute), 128
vertex_ids_zero (geomdl.elements.Triangle attribute), 128
vertices (geomdl.Abstract.Tessellate attribute), 126
vertices (geomdl.elements.Triangle attribute), 129
vertices (geomdl.tessellate.TriangularTessellate attribute), 126
vertices_raw (geomdl.elements.Triangle attribute), 129
vertices_uv (geomdl.elements.Triangle attribute), 129
vis (geomdl.Abstract.Curve attribute), 60
vis (geomdl.Abstract.Multi attribute), 119
vis (geomdl.Abstract.Surface attribute), 66
vis (geomdl.BSpline.Curve attribute), 71
vis (geomdl.BSpline.Surface attribute), 79
vis (geomdl.Multi.MultiCurve attribute), 120
vis (geomdl.Multi.MultiSurface attribute), 121
vis (geomdl.NURBS.Curve attribute), 85
vis (geomdl.NURBS.Surface attribute), 94
VisAbstract (class in geomdl.Abstract), 131
VisAbstractSurf (class in geomdl.Abstract), 132
VisConfig (class in geomdl.visualization.VisMPL), 133
VisConfig (class in geomdl.visualization.VisPlotly), 137
VisConfigAbstract (class in geomdl.Abstract), 132
VisCurve2D (class in geomdl.visualization.VisMPL), 134
VisCurve2D (class in geomdl.visualization.VisPlotly),
VisCurve3D (class in geomdl.visualization.VisMPL), 134
VisCurve3D (class in geomdl.visualization.VisPlotly), 138
VisMPL (module), 133
VisPlotly (module), 137
VisSurface (class in geomdl.visualization.VisMPL), 136
VisSurface (class in geomdl.visualization.VisPlotly), 139
VisSurfScatter (class in geomdl.visualization.VisMPL),
VisSurfTriangle (class in geomdl.visualization.VisMPL), 135
VisSurfWireframe (class in geomdl.visualization.VisMPL), 136

W
weight (geomdl.CPGen.GridWeighted attribute), 118
weights (geomdl.NURBS.Curve attribute), 85
weights (geomdl.NURBS.Surface attribute), 94

X
x (geomdl.elements.Vertex attribute), 130

Y
y (geomdl.elements.Vertex attribute), 130

Z
z (geomdl.elements.Vertex attribute), 130