SoftwareX 9 (2019) 85–94

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

# NURBS-Python: An open-source object-oriented NURBS modeling framework in Python

Onur Rauf Bingol *, Adarsh Krishnamurthy

*Department of Mechanical Engineering, Iowa State University, United States*

ARTICLE INFO

*Article history:*
Received 23 August 2018
Received in revised form 8 November 2018
Accepted 20 December 2018

*Keywords:*
Curve and surface modeling
Non-uniform rational B-splines
Object-oriented programming
Python

ABSTRACT

We introduce *NURBS-Python*, an object-oriented, open-source, Pure Python NURBS evaluation library with no external dependencies. The library is capable of evaluating single or multiple NURBS curves and surfaces, provides a customizable visualization interface, and enables importing and exporting data using popular CAD file formats. The library and the implemented algorithms are designed to be portable and extensible via their abstract base interfaces. The design principles used in NURBS-Python allows users to access, use, and extend the library without any tedious software compilation steps or licensing concerns.



## Code metadata

| | |
|---|---|
| Current code version | v4.3.8 |
| Permanent link to code/repository used for this code version | https://github.com/ElsevierSoftwareX/SOFTX_2018_148 |
| Legal Code License | MIT License |
| Code versioning system used | Git |
| Software code languages, tools, and services used | Python, Matplotlib, Plotly |
| Compilation requirements, operating environments & dependencies | Python v2.7.x, v3.4.x and higher |
| If available Link to developer documentation/manual | https://nurbs-python.readthedocs.io |
| Support email for questions | nurbs-python@googlegroups.com |

## 1. Introduction

Non-Uniform Rational B-Splines (NURBS) are accepted as the industry standard for the representation of geometry in mechanical computer-aided design (CAD) systems. In addition, they are used in many other fields such as robotics and self-driving cars that require dealing with geometric elements for trajectory generation and smoothing. Traditionally, NURBS algorithms are developed using compiled languages, such as C [1] or C++ [2,3] in order to achieve better performance over interpreted languages. However, running simple geometrical queries using compiled libraries require tedious and complicated setup that depends primarily on the operating system and computer architecture. A NURBS library written using an interpreted language can considerably reduce the overhead of compiling the library, and lead to widespread usage of the NURBS algorithms in different applications.

In order to develop a widely accessible NURBS library, we have implemented it using Python. Python [4] is an interpreted high-level programming language that is widely used by non-programmers and scientists. By design, Python code can work on most modern platforms. The reference implementation, *CPython* provides a well-designed C programming interface for interacting with different libraries. As a result, modern libraries, such as Theano, TensorFlow, Cognitive Toolkit (CNTK), provide user interfaces implemented in Python to reduce the programming interface learning effort and development time, while using Python's C programming interface for accessing low level libraries, such as nVidia's CUDA and cuDNN.

This paper describes the NURBS-Python package and its programming interface. NURBS-Python is a computational geometry library specifically designed for evaluating rational and non-rational B-Spline curves and surfaces. NURBS-Python is an open-source library designed to have minimum external dependencies.

* Corresponding author.
*E-mail addresses:* orbingol@iastate.edu (O.R. Bingol), adarsh@iastate.edu (A. Krishnamurthy).

https://doi.org/10.1016/j.softx.2018.12.005

The library provides a fully extensible object-oriented data structure, evaluation, and visualization capabilities implemented using the Python programming language. The library can be utilized using a direct object-oriented application programming interface (API). It does not contain elements that could restrain its flexibility, such as a graphical user interface (GUI) or a domain-specific language implementation. Instead, it allows users implement any graphical user interface using its abstract base classes. The optional visualization component included in the package implements this abstract base and can be used for plotting the curves and surfaces. The abstract base, data API, and evaluation capabilities are self-contained with no external module dependencies. On the other hand, the visualization components implement most commonly used plotting and visualization libraries, such as Matplotlib and Plotly.

This paper presents the different components of the NURBS-Python library. The main contributions of this paper include:

- An object-oriented and self-contained NURBS framework providing easy-to-use data structures and extensible algorithms.
- A pure Python NURBS computational library with no extra dependencies or compilation requirements.
- Utilizing existing plotting libraries to visualize NURBS curves and surfaces.
- A free and open-source extensible framework without any licensing concerns.

This paper is arranged as follows. In Section 2, we highlight our design considerations and compare NURBS-Python with existing packages and In Section 3 we outline the NURBS formulations. In Section 4, we discuss the components of the framework and their the implementation details including the different algorithms used. Finally, in Section 5 we provide some code examples describing the framework with different curve and surface examples.

## 2. Design considerations

NURBS-Python is an object-oriented NURBS evaluation library with data structures suited for geometric operations. NURBS have a compact definition any NURBS shape (curve or surface) can be defined by its degrees, knot vectors, and a set of control points. These are usually input by the user, however, the library is also capable of automatically generating a uniform knot vector, partly simplifying the knot vector input for some use cases. The geometric output variables are computed after evaluating the shape. The curve and surface objects are interactive; it is possible to change any input variable at runtime and the library re-evaluates the shapes automatically.

In order to achieve the best compatibility, NURBS-Python is designed to only use modules that come with the core Python distribution [4], also called as *Python Standard Library* including the mathematical evaluation libraries. Having no external dependencies allows users to have a lightweight and portable package that can be integrated with different architectures with minimal effort. A self-contained pure python library also protects users from binary interface incompatibilities and eliminates extra compilation steps and installation of third-party software for running a simple code segment. Due to its modular and object-oriented nature, the evaluation capability of NURBS-Python is easily extensible to a variety of platforms, such as HPC clusters or GPUs. Moreover, it can be used for various use cases such as integration with CAD, CAM, robotics, and machine learning libraries and for educational purposes for teaching geometric modeling concepts.

Development of NURBS date back to 1950s and therefore, it would be unwise to think that NURBS-Python is the only library of its kind. However, we are not aware of any pure Python standalone NURBS evaluation library for direct comparison. Nevertheless, we would like to compare our library with some of the existing open-source libraries containing NURBS components. We would like to note that the following libraries are not stand-alone NURBS evaluation libraries and mostly designed to support other uses such as isogeometric analysis. One of the most famous and commonly used Python library for isogeometric analysis is `igakit` [5]. Although `igakit` has a complete NURBS evaluation implementation, it is heavily dependent on third-party libraries and require compilation for usage. Additionally, `igakit` does not provide a separate package for its NURBS evaluation module. Moreover, the software design approach between `igakit` and NURBS-Python is different. The NURBS evaluation component of `igakit` uses a mathematical approach, which does not differentiate between the dimension of the NURBS shapes. On the other hand, NURBS-Python's design approach is focused specifically on curves and surfaces, and can be extended to volumes. Although, this is just a matter of design preference, we believe that our approach improves the usability of the library. There are also several NURBS libraries developed using MATLAB/Octave [6–8]. These implementations are also mainly designed to support isogeometric analysis, however, they are developed in programming languages that are considered proprietary or closed-source.

## 3. A brief introduction to NURBS

A NURBS shape is defined as a vector-valued function of one or more parameters which maps a $k$-dimensional space into, at minimum, a $k+1$-dimensional space. This function is simply the vector product (or a tensor product, depending on the $k$ value) of basis or blending functions by a set of $n$-dimensional control points [9,10].

The basis functions in NURBS are evaluated using the *Cox–de Boor* recursion algorithm as described in Eq. (1), where $\xi$ is the parameter value, $N_{i,p}$ is the $i$th order basis function and $p$ is the degree of the shape defined for the parametric dimension in consideration [9].

$$N_{i,0}(\xi) = \begin{cases} 1 & \text{if } \xi_i \leq \xi < \xi_{i+1} \\ 0 & \text{otherwise} \end{cases} \tag{1a}$$

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi) \tag{1b}$$

The derivatives of the basis functions can be evaluated using Eq. (2) [9].

$$\frac{d^k}{d^k\xi} N_{i,p}(\xi) = \frac{p!}{(p-k)!} \sum_{j=0}^{k} \alpha_{k,j} N_{i+j,p-k}(\xi) \tag{2a}$$

$$\alpha_{0,0} = 1 \tag{2b}$$

$$\alpha_{k,0} = \frac{\alpha_{k-1,0}}{\xi_{i+p-k+1} - \xi_i} \tag{2c}$$

$$\alpha_{k,j} = \frac{\alpha_{k-1,0} - \alpha_{k-1,j-1}}{\xi_{i+p+j-k+1} - \xi_{i+j}}, \tag{2d}$$

$$\alpha_{k,k} = \frac{-\alpha_{k-1,k-1}}{\xi_{i+p+1} - \xi_{i+k}} \tag{2e}$$

where $j = 1, \ldots, k-1$. The derivatives are used to compute tangents, normals, and binormals of the NURBS shapes.

After computing the basis functions, a single point on the curve corresponding to the parameter $\xi$ is evaluated by simply multiplying the basis functions with the corresponding control points, $B_{i,j}$ and summing up the multiplication results, as described in Eq. (3) [9].

$$C(\xi) = \sum_{i=1}^{n} N_{i,p}(\xi) P_i \tag{3}$$

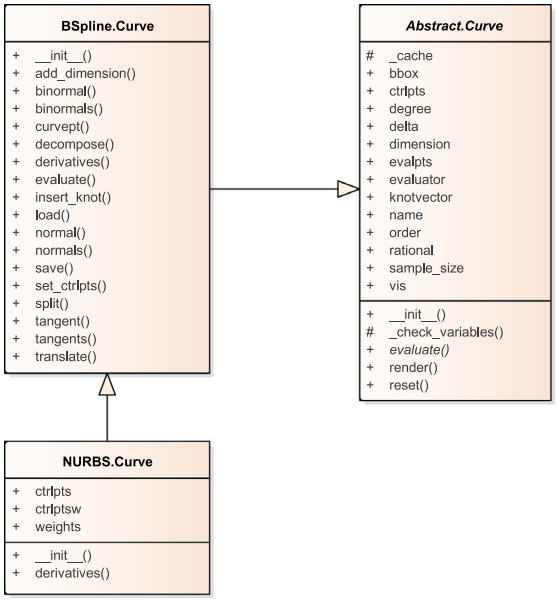


**Fig. 1.** Class diagram showing the inheritance of Curve classes.

The same evaluation method also applies to the surfaces. Surfaces are defined on a 2-dimensional parametric space described using $(\xi, \eta)$, To calculate a single point corresponding to these parameters, the basis functions on each parametric dimension, $N_{i,p}(\xi)$, $M_{i,q}(\eta)$, need to be evaluated and multiplied with the control points, $P_{i,j}$, and summed up as described in Eq. (4) [9].

$$S(\xi, \eta) = \sum_{i=1}^{n} \sum_{j=1}^{m} N_{i,p}(\xi) M_{j,q}(\eta) P_{i,j} \tag{4}$$

These equations are also applicable to Bézier curves and surfaces, since NURBS are a superset of Bézier shapes.

NURBS shapes can be divided into two types with respect to their knot vector structure: clamped and unclamped. A clamped shape can be understood from the repetitions of the knots at the beginning or at the end of its knot vectors. In an unclamped shape, there would be no repeating knots on both ends of the knot vector. Unclamped shapes still follow all NURBS properties and they can be evaluated using the same equations. For instance, the knot vector in Eq. (5) defines a clamped shape at both ends, where $p$ is the degree and $m + 1$ corresponds to the number of knots in the knot vector [9].

$$U = \{\underbrace{a, \ldots, a}_{p+1}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1}\} \tag{5}$$

All knot vectors obey the following equation:

$$m = p + n + 1 \tag{6}$$

where $p$ is the degree, $m + 1$ is the number of knots in the knot vector and $n + 1$ is the number of control points [9]. The values in the knot vectors are non-decreasing. These properties are used to validate the knot vectors.

## 4. Components of the framework

We describe the main components and the features of NURBS-Python on the following sections. More details can be found in Appendix A.

### 4.1. Core components and data structures

The core component involves the data structures for representing the shapes in the form of curves and surfaces and the evaluation functionality as well as the abstraction layer for providing extensibility. It includes input validation methods, which validates all user inputs with respect to the mathematical description of NURBS discussed in Section 3, and a caching system, which is directly integrated to improve interactivity of the library. NURBS-Python provides abstraction via the `Abstract` module. This module provides templates for future extension of the library and tries to maintain the programming interface as standard as possible between the current and the extended modules.

The data structure and the evaluation operations are available in `NURBS` and `BSpline` modules representing *rational* and *non-rational* versions of the non-uniform basis splines, respectively. The only difference between these modules is the structure of the control points. Evaluation functionality in `NURBS` module requires weighted control points, whereas `BSpline` requires no weights. The logic behind this design is generating an easy-to-understand environment by logically separating rational and non-rational algorithms and eliminating the need for extra information, such as weights, for users who only prefer to work with non-rational surfaces and curves.

The `BSpline` and `NURBS` modules contain two classes for representing the geometrical shapes. `Curve` class represents a single-manifold $n$-dimensional curve shape and `Surface` class represents a two-manifold 3-dimensional surface shape. All NURBS and B-Spline classes implement `evaluate` method for shape evaluation and `evalpts` property to retrieve evaluated points. These classes automatically evaluate the shape when required, such as plotting the shapes or retrieving the evaluated points from the object instance; therefore, it is not needed to explicitly evaluate the shape before any operation that requires the evaluated shape. The class diagrams of `BSpline` and `NURBS` modules showing their inheritance are shown in Figs. 1 and 2. Moreover, the `Multi` module contains two classes, `MultiCurve` and `MultiSurface`, for storing and evaluating multiple curves and surfaces simultaneously.

### 4.2. Curve and surface evaluation

The curve and surface evaluation is handled by the `Evaluator` module and the abstract base class used is `AbstractEvaluator`. The class diagram showing the inheritance of the included `Evaluator` modules are shown in Fig. 3. This module contains knot vector span finding algorithms using linear and binary search techniques, basis function and point evaluation algorithms as suggested by Piegl and Tiller [9]. The `Evaluator` modules are designed to allow users to extend algorithms easily or use them as an evaluation strategy, i.e. change them at runtime without need of re-creating the object instance. Therefore, it makes easier to compare, mix and reuse different evaluation algorithms with the same shape data.

In order to evaluate a shape (a curve or a surface), the user first sets the degrees, the knot vectors, the control points and the sample size or evaluation delta which corresponds to the number of points to be evaluated by NURBS-Python. The included evaluation algorithms can handle clamped and unclamped shapes. The evaluation interval delta, or the `delta` property, should be between 0.0 and 1.0, otherwise the library warns the user to pick a value between the interval. Sample size, represented by the `sample_size` property, can be any number bigger than 1 and it should be an integer value. NURBS-Python is designed to fix any

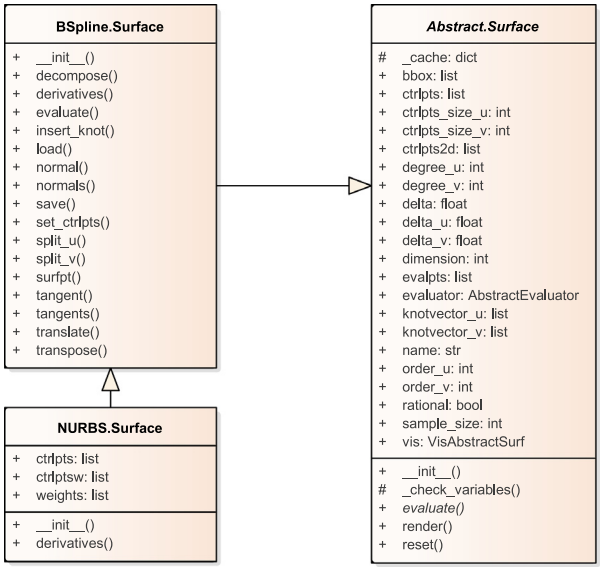


**Fig. 2.** Class diagram showing the inheritance of Surface classes.

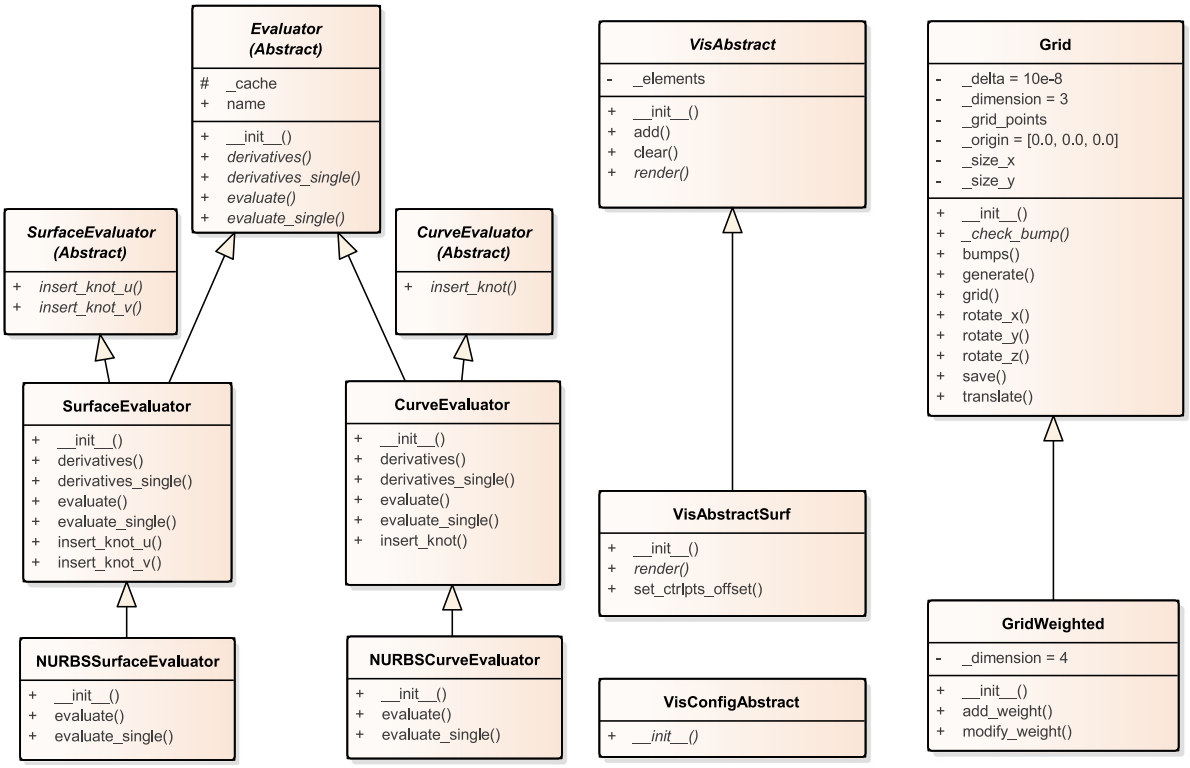


**Fig. 3.** Class diagrams showing the inheritance of abstract Evaluator, surface generator and visualization classes.

discrepancies by applying type casting when a value could properly be converted into the correct type without additional user input.

The following equation shows the relation between `delta` ($d$) and the `sample_size` ($S$):

$$S = \frac{1}{d} + 1 \tag{7}$$

The evaluation of $\frac{1}{d}$ mostly results in a floating point value; therefore its result is type-casted to the integer type. The type casting operation rounds the result of the division to the lower integer value. The `delta` property is set to 0.01, by default. The main steps in evaluating curves and surfaces are:

1. Find spans on the given knot interval which are determined via the input sample size.
2. Compute basis functions for the spans using Eq. (1).
3. Evaluate the shape by finding all the control points that belong to the given knot interval and performing the multiplication with the basis functions and summation using Eqs. (3) or (4).

After evaluation, the evaluated points are automatically cached and can then be accessed using the `evalpts` property. The internal

cache is used to speed up the responsiveness of the library, since the evaluated points could only change if any of the evaluation variables (degree, knot vector, control points, or the evaluation delta) changes. If any change occurs in these variables, the cache is reset and the shape is re-evaluated.

NURBS-Python also has capabilities to evaluate derivatives of the curves and surfaces using the algorithms suggested by Piegl and Tiller [9]. The method `derivatives` is designed to evaluate nth order derivatives of the curves and surfaces. The geometric interpretation of the derivatives, such as tangents, normals, and binormals, have their own methods, `tangent`, `normal`, `binormal` respectively. Curve tangents are computed from the 1st derivative at the given parametric value, normals are computed from the 2nd derivative, and binormals are computed by vector cross-product of tangents and normals. These vectors correspond to the Frenet–Serret Frame which is a right-handed system of a pairwise orthonormal vectors that follow the curve. In a Frenet–Serret frame, the tangent ($T$), normal ($N$) and binormal ($B$) vectors are perpendicular to each other and can be computed from 1st derivatives, 2nd derivatives, and via the equality $B = T \times N$, respectively. Surface tangents are computed from the 1st derivative with respect to each parametric direction, and the surface normal is computed by vector cross-product of the tangents in each direction.

### 4.3. Shape splitting and knot insertion

The `Multi` module has capabilities to evaluate multiple curves and surfaces. NURBS-Python uses `Multi` objects to return split or decomposed shapes. Curve classes have `split` and surface classes have `split_u` and `split_v` methods corresponding to each parametric direction. Both curve and surface classes have `decompose` methods for Bézier decomposition. Both shape splitting and decomposition can be achieved by using an evaluator method `insert_knot` which is implemented using the algorithm suggested by Piegl and Tiller [9].

The splitting algorithm takes a parameter value $u$ between 0.0 and 1.0, finds the multiplicity $s$ of the parameter over the knot vector in the chosen parametric direction and inserts $r$ number of knots (calculated using the equation $r = p - s$, where $p$ is the degree of the shape in the chosen parametric direction). Although this operation is simply considered as splitting [9], it is not enough to generate 2 different shapes with separate knot vectors and control points arrays. In order to generate two different shapes, NURBS-Python needs to find the exact span on the knot vector which defines the split point. The split knot span in consideration can be found by adding 1 to the input parameter value, $u + 1$. NURBS-Python then splits the knot vector using the split knot span into 2 separate vectors. Since the generated shapes are always clamped, the $u$ value is added to the end of the first knot vector and to the beginning of the second knot vector to satisfy the rules discussed in Section 3. The control points array is separated into two arrays at the split location by adding the span of the input parameter, $u$ and the number of knots inserted, $r$. Finally, the separated knot vectors and control points arrays are saved into the `MultiCurve` or `MultiSurface` container object.

Bézier decomposition is performed by applying tree expansion on the shapes. The shape is split at the middle knot locations. i.e. the ones that are not 0.0 or 1.0s. The split shapes can be considered as the leaf nodes. If any leaf node is still splittable (i.e. not a Bézier segment or a patch), the algorithm continues splitting at the middle knot locations until no middle knots remain in the knot vectors. For curves, the decomposition algorithm is directly applied as they are described with one parametric direction. The resultant curve segments are stored in a `MultiCurve` container object. For surfaces, the decomposition algorithm is applied on first $v$ parametric direction and then $u$ parametric direction. Similar to the decomposed curves, the resultant Bézier surface patches are stored in a `MultiSurface` container object.

### 4.4. Surface generator

We initially designed the surface generator module to generate sample surfaces for testing our work on incorporation of defects on layered composite structures [11]. The classes `Grid` and `GridWeighted` are used to generate surfaces that are compatible with `BSpline` and `NURBS`, respectively. The class diagrams are displayed on Fig. 3. Both classes are initialized with width, $x$ and height, $y$ with zero thickness. The grid mesh is generated by inputting number of divisions in both width and height directions. These divisions define the control points locations and they are uniformly distributed over the generated surface. These steps are enough for generating a planar surface in desired dimensions with desired number of control points. Additional details on surface generation can be found in Appendix A.1.

### 4.5. Exporting and importing data

Despite being designed as a low-level library, NURBS-Python includes a CAD interoperability and exchange module for extending its usability with the other software. The interoperability module provides control points grid manipulation operations, such as changing the array structures from weighted to unweighted, extracting and replacing weights and changing the row order. The exchange module provides support for exporting control points and evaluated points as CSV and VTK polydata formats, and exporting surfaces using common CAD formats, such as OBJ, STL, and OFF.

Since the CAD formats defined here require a triangulated surface, NURBS-Python includes a simple triangulation functionality (Fig. 4). This functionality uses the parametric correspondence of the 3-dimensional surface points using the `delta` or `sample_size` property as described using Eq. (7) to generate the triangle mesh. These properties determine the distance between the surface points on the parametric space and this information

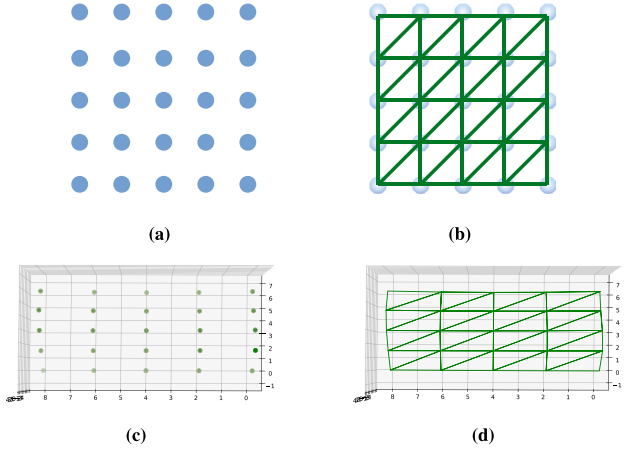


**Fig. 4.** Generating a triangulated surface with NURBS-Python using *delta* = 0.25. (a) Positions of the evaluated points on the parametric space, (b) triangulated surface on the parametric space, (c) A scatter plot on the 3-dimensional space illustrating the positions of the individual evaluated surface points. (d) Triangulated surface.

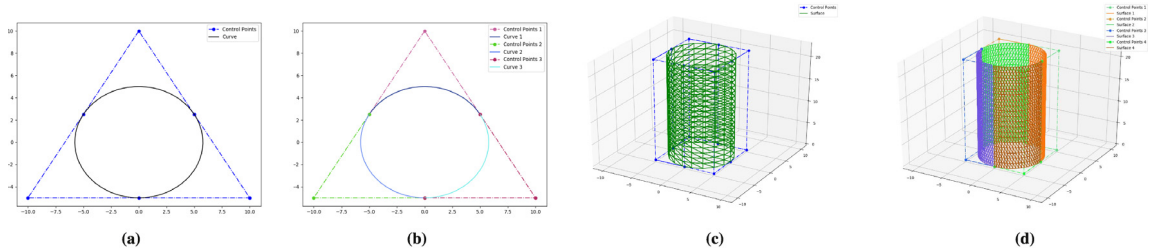


**Fig. 5.** NURBS curves and surfaces generated by the library and plotted using Matplotlib implementation of the visualization component. (a) A circular NURBS shape generated from 7 control points, (b) Bézier decomposition of the circular shape, each curve segment is colored differently, (c) A cylindrical NURBS surface, (d) Bézier decomposition of the cylindrical surface, each surface patch is colored differently. The colors on the decomposed shapes are randomly generated.

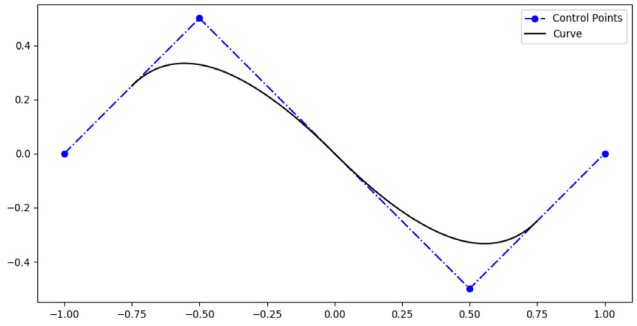


**Fig. 6.** An unclamped curve plotted using Matplotlib implementation of the visualization component. Blue dotted–dashed line shows the control points polygon and the black solid line shows the evaluated curve.

can be used to pick the closest points that could generate non-overlapping triangles. The size of the triangles, and therefore the smoothness of the surface can be fine-tuned by the user input.

NURBS-Python is designed to work with Python's default container classes, such as `list` and `tuple` to import new shapes. We implement some importing functionality from text files and `libconfig`-type files out-of-the-box; however, we prefer not to invent another file format and instead make the package work with Python's default container classes. Any class or package that can output the data as `list` or `tuple` is compatible with NURBS-Python for data import. For libraries that output the control points in different formats, such as $(x, y, z, w)$ in OpenNURBS [3] or separate $(x, y, z)$ and $w$ arrays, where $x, y, z$ are the coordinates of the control points and $w$ is the weight value, NURBS-Python's `compatibility` module can be used to convert the control points into the format that NURBS-Python can read as well as saving them as text files.

Importing from the CAD exchange formats, which directly support NURBS data structures, such as IGES and X3D, is still work in progress and will be released in the next major version of NURBS-Python.

### 4.6. Visualization components

NURBS-Python implements 2 common python visualization libraries, Matplotlib [15] and Plotly [16] using its abstract base classes, `VisAbstract` for general plotting and `VisAbstractSurf` for surface plotting customizations. These provide native ability to visualize the NURBS shapes. Their class diagrams are illustrated on Fig. 3. Moreover, Figs. 5 and 7 illustrate single and multi surface visualization capabilities using different plotting libraries. Fig. 6 illustrates a 2-dimensional curve unclamped on the both sides and plotted using Matplotlib implementation of the visualization component.

The curve visualization classes `VisCurve2D` and `VisCurve3D` directly uses the line plots and surface visualization class `VisSurface` uses the triangulation method discussed in Section 4.5. To plot the control points grid mesh, `VisSurface` uses an algorithm that connects the closest 4 control points to generate quads. The closest points are identified from the structure of the control points array.

Each visualization class can be configured using a configuration class. The abstract base of the configuration classes is `VisConfigAbstract` as referred in Fig. 3. The configuration class can only make visual changes to the output curve or surface plot, such as changing the figure size and the resolution, hiding legend from the plot as illustrated in Fig. 8. In our design, it is only possible to configure at the visualization instance generation step, otherwise it will use the default configuration implementation with optimal

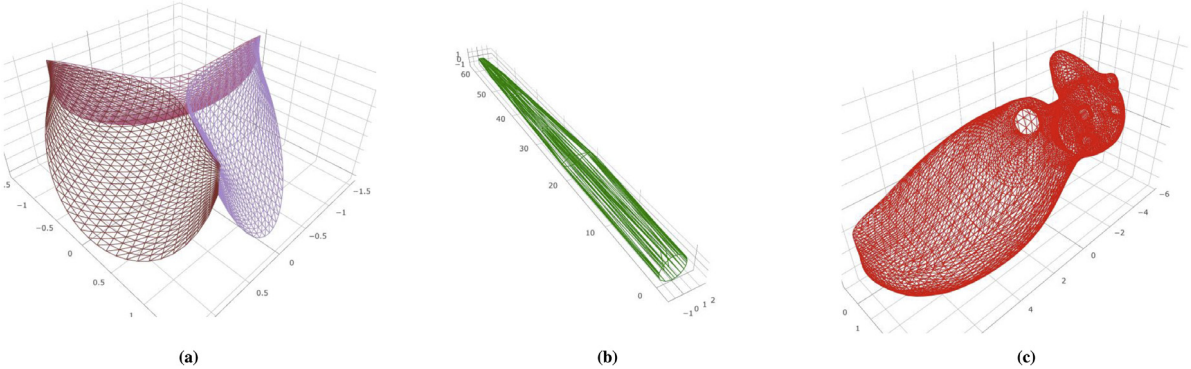


**Fig. 7.** NURBS multi-surfaces (`MultiSurface` class) generated by the library and plotted using Plotly implementation of the visualization component. (a) Heart valve modeled from 3 surface patches [12]. (b) Wind turbine blade modeled from 28 surface patches [13]. (c) Human heart modeled from 297 surface patches [14].

values (i.e. everything is visible on a screen with 1280 × 720 pixels resolution) set inside the visualization modules.

### 4.7. Testing and algorithm validation

To test the library and validate the results of the geometrical algorithms, we have implemented unit and functional tests for NURBS-Python using `pytest` and all tests are connected to a continuous integration (CI) system for automated testing of the library. We have achieved the geometrical evaluation validity by implementing multiple tests with different inputs for checking the complete shape evaluated with a relatively large delta value or only specific regions (e.g. only validating the affected region after the knot insertion operation).

The automated tests included cover all the algorithms, python properties (getters and setters) and most of the helper functionality. We were able to achieve full code coverage via the tests on the algorithms validation excluding the input checking and data validity parts of the methods. At the time of this writing, we were able to achieve around 70% code coverage with 257 automated tests.

## 5. Code examples

The following examples illustrate how to generate a curve and a surface, and then visualize it using NURBS-Python. We start with a 3-dimensional curve example.

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

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

# Set curve degree
crv.degree = 3

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

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

# Evaluate the curve
crv.evaluate()

# Set the visualization component
crv.vis = VisMPL.VisCurve3D()

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

The code listing starts with importing the modules and then we create a curve instance, controlled by the variable `crv`. We set the curve degree and input control points using the property `ctrlpts`. The control points are represented as list of n-dimensional coordinates using Python lists. Using the `utilities` module, we generate a uniform knot vector automatically and set it using the `knotvector` property. Finally, we evaluate the curve, although the library would automatically evaluate the curve or the surface when the evaluated points are requested by an internal component or a user. The evaluation method computes the curve points using the default Evaluator algorithm. For the visualization part, we set the visualization module designed for plotting 3-dimensional curves using the `vis` property and executing `render` method of the Curve class will plot the curve by calling Matplotlib functions.

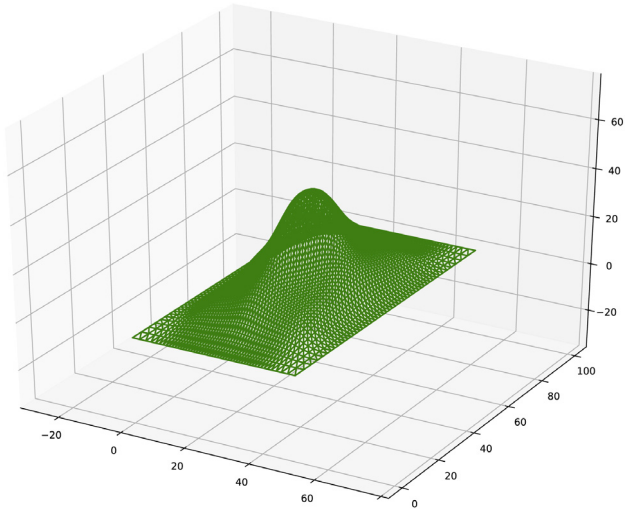


**Fig. 8.** The generated surface is visualized using Matplotlib implementation of the visualization component. The control points polygon and the legend are removed from the figure using the visualization configuration class.

The following code listing generates a surface using NURBS-Python and plots the surface using the Plotly implementation of the visualization component.

```
from geomdl import BSpline, utilities
from geomdl.visualization import VisPlotly

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

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

# List of control points
control_points = [[0, 0, 0], [0, 4, 0], [0, 8, -3],
                  [2, 0, 6], [2, 4, 0], [2, 8, 0],
                  [4, 0, 0], [4, 4, 0], [4, 8, 3],
                  [6, 0, 0], [6, 4, -3], [6, 8, 0]]

# Set control points
surf.set_ctrlpts(control_points, 4, 3)

# Auto-generate knot vectors
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 = 25

# Evaluate surface
surf.evaluate()

# Set the visualization component
vis_component = VisPlotly.VisSurface()
surf.vis = vis_component

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

The surface generation example is similar to the curve generation example. The main difference is in setting the control points. The control points shown with the variable `control_points` on the above example are stored in a list of 3-dimensional coordinates. However, a surface is defined over a 2-dimensional parametric space and therefore, requires a grid of control points. To allow user input as a single dimensional array of coordinates, we implemented a structure only applicable to the surfaces. On this

structure, 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 is found for the next $u$ value. This variation is controlled by a separate function, `set_ctrlpts` as Python properties cannot be arranged to accept multiple variables as the same time. We also did not want to confuse the users by implementing structures, such as Python dictionaries as the input. The `set_ctrlpts` function takes the control points and the number of control points in $u$ and $v$ directions as the input.

It would not be possible to provide examples for all the features of NURBS-Python in this paper, and hence we have also released a set of example scripts publicly on *Github* with the intention of providing templates to the NURBS-Python users. We constantly add more examples for the new features, integration and usage scenarios that we encounter while using the library. We encourage NURBS-Python users to refer to the examples repository (https://github.com/orbingol/NURBS-Python_Examples) for more possible usage and integration scenarios.

## 6. Conclusions and future work

We have introduced an open-source object-oriented geometric modeling library with visualization options. We have publicly released the library on https://github.com/orbingol/NURBS-Python and in addition, we provide over 40 example scripts that illustrate the features of the library and some sample usage scenarios on a separate *GitHub* repository. The scripts to generate some of the figures illustrated on this paper can also be found in that repository. We also provide a complete class documentation with more examples and figures. The documentation is automatically generated and published on *ReadTheDocs*, a free documentation generation and publishing website. Users can also access to the other reports, such as continuous integration system logs and code coverage graphs via project's *GitHub* page. To increase the accessibility of the library on different platforms and reduce the user effort for installation, we have uploaded NURBS-Python to Python Package Index (pypi.org) and Anaconda Cloud (anaconda.org), allowing users to download the library using the package managers `pip` and `conda`.

NURBS-Python is designed to be an extensible and open-source framework for geometric modeling. Since it is freely available on a public domain, developers can extend the library in their own liking or integrate it in their own works. Nevertheless, we would like to add some comments on our current work and some possible extension paths for the NURBS-Python library. We will be adding additional spline algorithms, such as knot removal, degree elevation and reduction, as well as fitting, trimming, offsetting and volume parameterizations. We are currently developing a module called `shapes` for allowing users to generate commonly used NURBS shapes, such as circles, cylinders, torus, etc. Finally, extending the framework to support truncated hierarchical B-splines (THB-splines), T-splines, and polynomial/rational splines over hierarchical T-meshes (PHT/RHT-splines) [17] for adaptive geometric design would be a nice path for further extension of the library to support engineering applications, such as isogeometric analysis for structural mechanics [18].

## Acknowledgments

We would like to express our deepest gratitudes to all NURBS-Python users and contributors around the world for their time and efforts in testing NURBS-Python, reporting the bugs and commenting on the features. These contributions helped us to develop a solid NURBS evaluation framework for Python. As a courtesy, we have included their names in `CONTRIBUTING.rst` file on our *GitHub* repository. Adarsh Krishnamurthy is partly funded by the National Science Foundation under the grant numbers 1644441 and 1750865.

## Conflict of interest

The authors declare that they have no conflict of interest.

## Appendix A. Additional components of the framework

### A.1. Surface generator customization

Although generating a planar surface grid in desired size and exporting it as a text file for further customizations could be enough for most users, NURBS-Python also provides facilities to manipulate the shape of the generated surface. The `bumps` method includes an algorithm that allows users to generate hills (or bumps) on the surface. This algorithm generates 2 random numbers corresponding to width and height on the interval of the generated surface. These numbers correspond to the location (coordinates) of the peak of the hill to be generated. Then, the algorithm checks for surrounding locations for existing hills (i.e. non-zero $z$ value). If there are no hills generated previously, then the method applies the $z$ value, which is a user input argument named as *bump height*, to the peak location and the surroundings are generated by gradually dividing $z$ value to value computed by another input argument `base_extent` which simply generates a gradient from the peak of the hill to the base. In addition, the users can input a padding value using `base_adjust` argument which confines (i.e. a negative `base_adjust` value) or extends (i.e. a positive `base_adjust` value) the area on the $x$-$y$ plane of the grid where the hills are generated. The algorithm can pick either $+z$ or $-z$ direction to generate the hill. Since the algorithm depends on random value generation, it could get stuck on an infinite loop. Therefore, the algorithm stops after 25 hill generation trials by default, and number of trials can be changed using the `max_trials` input argument.

In addition to the hill generation algorithm, the surface generator also provides geometric operators for rotating the surface on x, y, and z axes about the input angle, and translation of the surface center to the input 3-dimensional position using the `translate` method.

Users can query the bounding box of the shape using `bbox` property. This property, when called by the user, automatically computes the bounding box of the evaluated shape and caches the values to eliminate excess bounding box computations. After the first computation, the values are always returned from the internal cache.

### A.2. Visualization customization

The visualization component can be set or changed at runtime using the `vis` property of the `Curve` and `Surface` classes. The plotting of the shape takes place when the user calls `render` method of the these classes. The plotting behavior can be controlled with additional input keyword arguments of the `render`

method. For instance; the user can save the plot with or without opening the plotting window or change the color of the control points and shape plots.

The library allows re-using all possible visualization options on the designated shape element. This means that a single `VisSurface` instance can be used to plot different surfaces contained in different `Surface` instances in `BSpline` or `NURBS` modules. The same applies to the `Curve` classes. However, it is not possible to use a surface visualization object with a curve class instance, or vice versa, due to inherent differences in the data structures.

## Appendix B. Additional code examples

The following code listing demonstrates the surface generator module, `CPGen` and its interoperation with the `BSpline` module.

```
from geomdl import BSpline, CPGen, utilities
from geomdl.visualization import VisMPL as vis
from geomdl import exchange

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

# Generate a 10x10 grid
surfgrid.generate(10, 10)

# Generate 1 bump at the center of the grid
surfgrid.bumps(num_bumps=1, all_positive=True, bump_height=45,
    base_extent=4, base_adjust=-1)

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

# Set order of the surface
surf.order_u = 4
surf.order_v = 4

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

# Set knot vectors
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 of the surface
surf.sample_size = 30

# Visualization component and its configuration
conf = vis.VisConfig(ctrlpts=False,legend=False)
surf.vis  = vis.VisSurface(conf)

# Plot the surface
surf.render()

# Export the surface as a .stl file
exchange.export_stl(surf, "surface.stl")
```

In this example, we have generated the control points grid using the surface generator module, represented by `CPGen`. Then, we generate a bi-cubic surface and automatically generate uniform knot vectors on each parametric direction. The generated surface is plotted using the Matplotlib component of the visualization module and finally, saved as a *.stl* file.

The following example illustrates the control points import facility of NURBS-Python along with Bézier decomposition and translation functionalities. The control points file *ex_surface03.cptw* is an ASCII text file and it can be found on the examples repository.

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

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

# Set degrees
surf.degree_u = 1
surf.degree_v = 2

# Set control points
surf.set_ctrlpts(*exchange.import_txt("ex_surface03.cptw",
    two_dimensional=True))

# Set knot vector
surf.knotvector_u = [0, 0, 1, 1]
surf.knotvector_v = [0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75,
     1, 1, 1]

# Decompose the surface
surfaces = operations.decompose_surface(surf)

# Translate one of the surface patch
operations.translate(surfaces[1], (-0.25, 0.25, 0), inplace=
    True)

# Set number of samples for all split surfaces
surfaces.sample_size = 50

# Plot decomposed surfaces
vis_comp = VisMPL.VisSurfWireframe()
surfaces.vis = vis_comp
surfaces.render()
```

As described in the previous examples, we generate a NURBS surface instance using a control points file. The initial surface is decomposed into Bézier patches and right after the decomposition, one of the Bézier patches is translated by the vector $[-0.25, -0.25, 0]$. Finally, decomposed surfaces are plotted via Matplotlib implementation of the visualization module.

## Appendix C. Performance metrics

It would not be possible to reach any conclusions from the running time of the interpreted code. However we have used a performance improvement method using an external module called *Cython* [19]. Cython corresponds to a compiler specifically designed for wrapping external code into a compiled Python module.

To assess the performance difference between the interpreted and the compiled versions, we compiled NURBS-Python with the *Cython* compiler and tested using a sample curve and a surface. We used a sample size (i.e. number of evaluated points) $S = 16384$ for the curves and $S = 1024$ for both parametric directions of the surface, resulting in a total of 1048576 evaluated surface points for each surface. Table C.1 shows our evaluation results in the format of mean $\pm$ standard deviation obtained from a computer with Intel Core i7-7700HQ CPU and 16 GB of RAM. The results are measured by applying IPython's `%timeit` magic on the `evaluate` method with 7 runs. The software versions used for the analysis are Python v3.6.6 and IPython v6.5.0.

As expected, we were able to get faster evaluation speeds using the compiled version. The speed increase we obtained by direct

**Table C.1**
Comparison of evaluation time between interpreted and Cython-compiled versions of NURBS-Python. Sample sizes: $S_{Curve} = 16384$ and $S_{Surface} = 1048576$.

| Library type | Curve | Surface |
|---|---|---|
| Interpreted | 167 ms $\pm$ 6.97 ms | 18 s $\pm$ 10.8 s |
| Compiled | 89 ms $\pm$ 2.55 ms | 6.41 s $\pm$ 263 ms |

Cython compilation was around 2 and 3 times on curves and surfaces, respectively. The most important thing to consider while performing the Cython compilation is that due to NURBS-Python being a pure Python library with no external dependencies, the compilation and linking requires no additional libraries other than the Python standard library.

## References


[1] SINTEF. The SINTEF Spline Library; 2018. URL https://github.com/SINTEF-Geometry/SISL. [Accessed June 2018].

[2] Eason Kang. libnurbs; 2018. URL https://github.com/yiskang/libnurbs. [Accessed June 2018].

[3] Robert McNeel & Associates. OpenNURBS; 2018. URL https://www.rhino3d.com/opennurbs.

[4] Rossum G. Python reference manual. Tech. rep., Amsterdam, The Netherlands, The Netherlands: CWI (Centre for Mathematics and Computer Science); 1995.

[5] Dalcin L, Collier N. igakit; 2018. URL https://bitbucket.org/dalcinl/igakit. [Accessed October 2018].

[6] de Falco C, Reali A, Vázquez R. GeoPDEs: a research tool for isogeometric analysis of PDEs. Adv Eng Softw 2011;42(12):1020–34.

[7] Vázquez R. A new design for the implementation of isogeometric analysis in Octave and Matlab: GeoPDEs 3.0. Comput Math Appl 2016;72(3):523–54.

[8] Nguyen VP, Anitescu C, Bordas SP, Rabczuk T. Isogeometric analysis: an overview and computer implementation aspects. Math Comput Simulation 2015;117:89–116.

[9] Piegl L, Tiller W. The NURBS book. Springer Science & Business Media; 2012.

[10] Cottrell JA, Hughes TJ, Bazilevs Y. Isogeometric analysis: toward integration of CAD and FEA. John Wiley & Sons; 2009.

[11] Bingol OR, Schiefelbein B, Grandin RJ, Holland SD, Krishnamurthy A. An integrated framework for solid modeling and structural analysis of layered composites with defects. Comput Aided Des 2019;106:1–12.

[12] Xu F, Morganti S, Zakerzadeh R, Kamensky D, Auricchio F, Reali A, Hughes TJ, Sacks MS, Hsu M-C. A framework for designing patient-specific bioprosthetic heart valves using immersogeometric fluid–structure interaction analysis. Int J. Numer Methods Biomed Eng. 2018;34(4). e2938.

[13] Herrema AJ, Wiese NM, Darling CN, Ganapathysubramanian B, Krishnamurthy A, Hsu M-C. A framework for parametric design optimization using isogeometric analysis. Comput Methods Appl Mech Engrg 2017;316:944–65.

[14] Krishnamurthy A, Gonzales MJ, Sturgeon G, Segars WP, McCulloch AD. Biomechanics simulations using cubic hermite meshes with extraordinary nodes for isogeometric cardiac modeling. Comput Aided Geom Design 2016;43:27–38.

[15] Hunter JD. Matplotlib: A 2D graphics environment. Comput Sci Eng 2007;9(3):90–5.

[16] Plotly Technologies Inc.. Plotly visualization library; 2015. URL https://plot.ly.

[17] Nguyen-Thanh N, Zhou K, Zhuang X, Areias P, Nguyen-Xuan H, Bazilevs Y, Rabczuk T. Isogeometric analysis of large-deformation thin shells using rht-splines for multiple-patch coupling. Comput Methods Appl Mech Engrg 2017;316:1157–78.

[18] Hughes TJ, Cottrell JA, Bazilevs Y. Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement. Comput Methods Appl Mech Engrg 2005;194(39–41):4135–95.

[19] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The best of both worlds. Comput Sci Eng 2011;13(2):31–9.