Splipy: B-Spline and NURBS Modelling in Python

K. A. Johannessen
SINTEF Digital, PO Box 4760, 7465, Trondheim, Norway
Kjetil.Johannessen@sintef.no

E. Fonn
SINTEF Digital, PO Box 4760, 7465, Trondheim, Norway
Eivind.Fonn@sintef.no

Abstract. Splipy is a pure Python library for the creation, evaluation and manipulation of B-spline and NURBS geometries. It supports n-variate splines of any dimension, but emphasis is placed on the use of curves, surfaces and volumes. The library is designed primarily for analysis use, and therefore allows fine-grained control over many aspects which cannot be achieved with conventional CAD tools. It is packaged and distributed through Python Package Index: PyPi which gives easy installation.

1. Motivation and significance
Non-uniform rational B-splines (NURBS) have been a staple technology in computer aided design (CAD) for many decades. Their mathematical precision allows the user to create smooth parametric descriptions of curves and surfaces, which is necessary in many engineering applications. Recent years has seen an increased interest in the use of B-splines and NURBS directly in high-fidelity physics simulations such as the finite element method, thereby uniting geometric modelling and analysis. These methods, termed isogeometric analysis (IGA) were first introduced in [1]. While commercial CAD tools do well to facilitate NURBS-based modelling, IGA requires finer control of discretization details which are conventionally kept hidden from the user. We propose Splipy as a modern easy-to-use Python library that will allow scientists and engineers the detailed control that they need to not only make B-spline meshes that are optimized for analysis, but which also work well for modelling. The primary delivery for this work is to be considered the open source software itself and this manuscript is an accompanying document.

The software allows for rapid generation of high-quality meshes that can be used directly in IGA programs. While Splipy offers everything needed in terms of basis function evaluations, derivatives etc. to build a stand-alone finite element solver, we envision that most users will be content to generate the geometry in Splipy followed by exporting to external solvers.

2. Software description
The core tenet of Splipy is the manipulation of spline objects, by which is meant a mapping from a n-dimensional parameter space,

$$\mathcal{P} = [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_n, r_n]$$

(1)
BSplineBasis ← SplineObject

Curve ← Surface ← Volume

“is a” ← “has a”

Figure 1: Class hierarchy of Splipy.

into physical space $\mathbb{R}^d$. Conventionally, $d \geq n$. In spline terms, each parameter interval $[l_i, r_i]$ is subdivided into a knot vector

$$l_i = k_{0}^{(i)} \leq k_{1}^{(i)} \leq k_{2}^{(i)} \leq \cdots \leq k_{m_i}^{(i)} = r_i,$$

(2)

with nondecreasing knots $k_j^{(i)}$. Given a knot vector and a polynomial degree $p_i$, there is a unique B-spline basis $B_j(x_i)$, $1 \leq j \leq n_i - p_i$, defined on the interval $[l_i, r_i]$. Each basis function $B_j$ is piecewise polynomial with degree $p_i$ in all knot intervals $[k_m, k_{m+1}]$, and is supported on exactly $p_i + 1$ such intervals.

Given a choice of knot vector and degree for each direction $i = 1, \ldots, n$, the full B-spline basis is defined on $\mathcal{P}$ as

$$B_{j_1,\ldots,j_n}(x_1, \ldots, x_n) = B_{j_1}(x_1) \times \cdots \times B_{j_n}(x_n), \quad 1 \leq j_i \leq n_i - p_i$$

(3)

and a spline object is nothing more than a function in the span of this basis, i.e. a function $s: \mathcal{P} \rightarrow \mathbb{R}^d$ such that

$$s(x) = \sum_{j_1,\ldots,j_n} c_{j_1,\ldots,j_n} B_{j_1,\ldots,j_n}(x).$$

(4)

Here, the coefficients $c_{j_1,\ldots,j_n} \in \mathbb{R}^d$ are termed control points.

Splipy is concerned with the creation and manipulation of these mappings and the objects they represent (that is, the image of $\mathcal{P}$ by $s$ in $\mathbb{R}^d$). With the intended emphasis on analysis, fine control over knot vectors and control points are provided. These quantities dictate the properties of the parametric representation and the geometric mapping respectively.

2.1. Software Architecture

There are three main classes which are exposed to the user: Curve, Surface and Volume. All inherit from the SplineObject superclass, which contains common methods such as affine transformations, evaluation and get methods. See Figure 1. It is possible to create such spline objects manually, by specifying knot vectors and control points, however most users will be using a wide range of “factory” functions. These can be found in the submodules curve_factory, surface_factory and volume_factory. More advanced geometries can then be created by using the rich interface for manipulating spline objects. Generally Splipy scripts follow a bottom-up construction scheme where curves are created from points, surfaces are created from curves and finally volumes are created from the surfaces.

2.2. Software Functionalities

The library allows for rapid creation of elementary mathematical constructs such as line, circle, disc, sphere, torus, cylinder and teapots. Some of these have multiple parameterization options, such as disc or sphere, see Section 2.2.3.
The factory classes also contain spline-related constructors, such as

**Curve Factories**
- Bezier curves
- Hermite Interpolation
- Cubic Curve Interpolation
- B-spline Interpolation
- Least Square Fit
- Adaptive Curve Fit

**Surface Factories**
- Sweep
- Revolve
- Loft
- Extrude
- Interpolation
- Least Square Fit
- Edge Curves (interior from four edges)

**Volume factories**
- Revolve
- Extrude
- Loft
- Interpolation
- Least Square Fit

We now elaborate on a few selected functions.
2.2.1. **Adaptive Curve fitting** For parametric representation of some target curve \( x(t) \) we provide a fit function in curve_factory which employs a posteriori error estimation to provide a best fit \( x_h(t) \) to the curve. This algorithm constructs a knot vector iteratively, in four steps as follows. The initial knot vector is arbitrarily chosen to be uniform, cubic and supports 10 basis functions.

**Step 1:** Compute the Greville abscissae \( t^*_i = \sum_{j=0}^{i+p} k_j \) and interpolate on these points \( x(t^*_i) = x_h(t^*_i) \).

**Step 2:** Estimate the error \( \| x(t) - x_h(t) \|_{L^2(k_i,k_{i+1})} \) for all knot spans \( (k_i, k_{i+1}) \)

**Step 3:** For all knot spans \( (k_i, k_{i+1}) \), compute the expected error \( \varepsilon_i \) and interpolation error \( \varepsilon_{h,i} \) defined as

\[
\varepsilon_i = \frac{k_{i+1} - k_i}{k_{i+1} - k_0} \varepsilon
\]

\[
\varepsilon_{h,i} = \int_{k_i}^{k_{i+1}} |x(t) - x_h(t)| \, dt
\]

where \( \varepsilon \) is some user-defined global tolerance

**Step 4:** Our approximation should converge at a rate of \( \| x(t) - x_h(t) \|_{L^2} = \mathcal{O}(h^{p+1}) \) with respect to the knot span size \( h = k_{i+1} - k_i \) (see [2]). Therefore we can expect that inserting

\[
n = 2^{-p} \frac{\varepsilon_{h,i}}{\varepsilon_i}
\]

new knots will give us the target error. These knots are then inserted into their relevant knot span and a new interpolation is computed.

These iterations are carried out until the global tolerance is achieved.

2.2.2. **Interior from edges** The default method to produce the interior mesh from 4 edge curves is an algorithm known as Coons patch [3]. The algorithm works as follows. Assume that parameterized left and right curves are given as \( c_1(v) \) and \( c_2(v) \), as well as a bottom and top curves \( c_3(u) \) and \( c_4(u) \), and without loss of generality assume a consistent parameter space \( u, v \in [0, 1] \). We create three surfaces \( S_1, S_2, S_3 \) which interpolate the left and right, top and bottom, and the four corners, respectively.

\[
S_1(u, v) = (1 - u)c_1(v) + uc_2(v)
\]

\[
S_2(u, v) = (1 - v)c_3(u) + vc_4(u)
\]

\[
S_3(u, v) = (1 - u)(1 - v)c_1(0) + u(1 - v)c_2(0) + (u - 1)vc_4(0) + uvc_4(1)
\]

The following surface will interpolate all edges simultaneously.

\[
S(u, v) = S_1(u, v) + S_2(u, v) - S_3(u, v)
\]

While Coons patches are incredibly versatile, they are not general enough that it will work on all cases. For some inputs the resulting surface \( S(u, v) \) might be self-intersecting and contain folds. For a more robust method it is possible to formulate the parameterization problem as Poisson problem or elasticity problem (linear or finite strain). For the Poisson case this is done by setting up the following partial differential equation:

\[
\frac{\partial^2 x}{\partial u^2} + \frac{\partial^2 x}{\partial v^2} = 0 , \quad u, v \in [0, 1]
\]

\[
x(0, v) = c_1(u) , \quad v = 0
\]

\[
x(1, v) = c_2(u) , \quad v = 1
\]

\[
x(u, 0) = c_3(v) , \quad u = 0
\]

\[
x(u, 1) = c_4(v) , \quad u = 1
\]

(9)
5

Radial: or polar parametrization

Square: parametrization with four corners

Hybrid: five different patches

(a) Radial: or polar (b) Square: parametrization (c) Hybrid: five different patches

Figure 3: Disc: multiple discretization options for the same geometry is provided. Here we show three different versions of the disc $\|x\|^2 < 1$, each with different benefits. In particular, the vanishing Jacobian can be located at the domain center, boundary or interior respectively, depending on the particular needs of the application.

and then solving for $x$, typically through some numerical scheme such as the finite element or finite difference method. In Splipy this is done by the use of the Nutils library [4] and is provided as an optional dependency. To complete the parametrization one would have to solve for the parametrization of $y$ in addition to the one for $x$.

For details on both the Poisson and the elasticity implementation see the work by Hansvold [5].

2.2.3. Sphere parametrization It is well known that perfect circles cannot be represented as B-splines, but can be represented as non-uniform rational B-splines (NURBS). However, there are multiple ways of parametrizing the circle and its surface and volume equivalent. In particular, we could create a polar parametrization which would include a geometric singularity point at the origin, or one might create what is called a “square” parametrization which computes the interior of the four edge curves given by four circle segments. These can be seen in Figure 3 along with a hybrid parametrization which uses five different patches. Note that the square parametrization has a singular Jacobian at its four corners, where the isocurves in both parametric directions are parallel. Either choice might be suitable depending on circumstances.

The volumetric parametrization of the sphere also has a “square” version which avoids the north- and south-pole singularity given by a spherical coordinate parametrization. For details on the derivation of this particular parametrization, see Cobb [6]. These are shown in Figure 4.

3. Meshing an NREL 5MW reference wind turbine blade

The following describes in brief detail the meshing of a NREL 5MW reference wind turbine blade, as described in [7]. This meshing procedure is described in more detail in [8] and was performed entirely with Splipy. This blade is defined as a surface interpolating a number of different airfoils placed at increasing distance from the origin, each of them with a specific shape (e.g. NACA or DU standard airfoil designs), size and angle of attack.

The construction of the blade itself is relatively straightforward. Splipy makes possible the description of each defining airfoil as a periodic spline curve with consistent discretization (that is, equal knot vectors). From this, the entire blade can be lofted in a single operation, see Figure 5 (left) and [8]. In order to prevent the resulting surface from becoming self-intersecting, an intermediate lower-order interpolation step is inserted. This strategy is also described in [8].
Figure 4: **Sphere:** multiple discretization options for the same geometry is provided. Here we show two different versions of the unit ball $\|x\|^2 < 1$. In particular, the vanishing Jacobian can be located at the domain center or boundary. NURBS representations allow for perfect geometry representation in either case.

The objective of the mesh is to perform flow simulations *around* the blade, so it is necessary also to create a volumetric mesh of a suitably large surrounding region. At each designated airfoil, a circular curve is generated with a certain radius, again with a consistent discretization in order to facilitate interpolation. The surface between the airfoil and the circle is then mapped using a technique known as *orthogonal transfinite interpolation* (OTI), see [8] for details. This technique is intended to ensure that the mesh lines near the airfoil lie orthogonal to the boundary, while the mesh itself remains suitably regular and not self-intersecting. To ensure this, OTI uses an iterative process, generating the enveloping curve one mesh line at a time from the interior toward the exterior. At each step, the enveloping curve is defined as a weighted mean between two curves: a standard linear interpolation between the previous enveloping curve and the exterior circle, and the orthogonal outward projection of the previous enveloping curve by a suitable distance. The weight of this mean is chosen so that the orthogonal projection dominates near the airfoil, while the linear interpolation dominates further away. In addition to this, each enveloping curve is then smoothed by a Laplacian operator, further reducing the risk of self intersections. Each enveloping curve thus defined, the region between the airfoil and the circle can be meshed by lofting, and the volumetric mesh is likewise created by a further lofting step, see Figure 5 (right).

This method illustrates the standard strategy of building geometry “by dimension”, from curves via surfaces to volumes. Splipy’s curve manipulation functions make it easy to implement this algorithm, which is somewhat involved, in a robust and clear manner.

### 4. Impact

Isogeometric analysis was introduced in 2005 [1] for applying native B-spline basis functions for both the geometry and finite element solution. One of the limiting factors of this endeavor has been sparse support for high quality B-spline mesh generation in conventional CAD software. Traditional meshing tools used for finite element technologies typically produce Lagrangian piecewise linear simplices such as tetrahedrons. Commercial software typically allow the user to create low-quality geometries containing gaps, overlaps and non-matching interfaces. These
Figure 5: Airfoil cross-sections of an NREL 5MW reference wind turbine blade.

Figure 6: The computational volume domain surrounding the turbine blade.

are often referred to as “dirty” geometries by the industry, and while it is possible to create analysis-suitable meshes, the process is currently tedious and time-consuming.

We hope that by this package allows researchers and engineers, especially in the isogeometric community, to consider more complex geometries.

While the package is a library and does not contain a dedicated graphical user interface (GUI), one should be able to create a CAD interface on top of it. We note that of particular interest is FreeCAD, which is an open source CAD software that supports Python plugins. It would not be unreasonable to see a FreeCAD plugin of Splipy that would greatly increase its utility and user-friendliness.

5. Conclusions
Splipy is a Python programming library which empowers users to work with B-spline and NURBS representations of geometries. It lowers the barrier of entry to create complex geometric shapes and allows for fine-grained control over parametrizations to ensure high-quality meshes which are tailored specifically for analysis.

Acknowledgments
The authors gratefully acknowledge the financial support from the Norwegian Research Council and the industrial partners of the FSI-WT project (RCN grant no: 216465/E20): Meteorologisk
Institutt, FFI, Statoil, Trønder Energi, Kjeller Vindteknikk and WindSim AS. Additionally we acknowledge the support from the Norwegian Research Council and the industrial partners of OPWIND: Operational Control for Wind Power Plants (Grant No.: 268044/E20).

6. References

[1] Hughes T J R, Cottrell J A and Bazilevs Y 2005 Computer Methods in Applied Mechanics and Engineering 194 4135–4195

[2] De Boor C, De Boor C, Mathématicien E U, De Boor C and De Boor C 1978 A practical guide to splines vol 27 (springer-verlag New York)

[3] Coons S A 1967 Surfaces for computer-aided design of space forms Tech. rep. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC

[4] van Zwieten G, van Zwieten J, Verhoosel C, Fonn E, van Opstal T and Hoitinga W 2018 Nutils

[5] Mothes Hansvold K A 2015 Mesh generation techniques for isogeometric analysis

[6] Cobb J E 1988 Tiling the sphere with rational bezier patches Tech. rep.

[7] Jonkman J, Butterfield S, Musial W and Scott G 2009 Definition of a 5-MW Reference Wind Turbine for Offshore System Development Tech. Rep. NREL/TP-500-36060 National Renewable Energy Laboratory Golden, Colorado, USA

[8] Fonn E, Rasheed A, Kvarving A M and Kvamsdal T 2015 Energy Procedia 80 294–301

Current code version

Table 1: Code metadata

| Current code version | 1.4 |
|----------------------|-----|
| Repository           | https://github.com/sintefmath/splipy |
| License              | GNU General Public License v3.0 |
| Code versioning system | Git |
| Programming language | Python |
| Dependencies         | Numpy ≥ 1.15, Scipy ≥ 1.2 |
| Documentation         | https://sintefmath.github.io/splipy/ |
| Support email for questions | Kjetil.Johannessen@sintef.no |