# bsplyne

Exploration and onboarding guide for the library

**Dorian BICHET**

January 29, 2026

**Abstract**

This document serves as a guide for onboarding with the `bsplyne` library. It details the architecture based on the Cox-de Boor algorithm, installation procedures, and provides a pedagogical progression through the tutorial examples.

## 1 Objectives

- ✓ **Navigation:** Master the software architecture and documentation API.
- ✓ **Exploration:** Understand the core objects by modifying the examples.
- ✓ **Efficiency:** Use AI (LLM) as a technical guidance lever.

## 2 Environment and Installation

**1. Environment Setup (Conda)**

Using a virtual environment is recommended to isolate dependencies (`Numba`, `SciPy`, `Matplotlib`):

```
conda create -n bsplyne_env python=3.9
conda activate bsplyne_env
```

**2. Installation in "Editable" Mode**

Installing the library in `-e` mode links the environment directly to the source code in the `bsplyne/` folder:

```
git clone https://github.com/Dorian210/bsplyne
cd bsplyne
pip install -e .       # Minimal installation without visualisation tools
pip install -e .[viz]  # Highly recommended for 3D interactivity
```

**Quick and dirty install**

Installing the library can be as simple as:

```
pip install bsplyne[viz]
```

without regard for creating a virtual environment or managing dependencies manually. This is fine for experimentation or quick testing, but for development or production use, it is strongly recommended to use a dedicated environment (Conda or venv) and install the optional dependencies properly.

## 3 Architecture and Documentation

### 3.1 Software Hierarchy: Layered Structure

The `bsplyne` architecture strictly follows the mathematical construction of splines:

1. `b_spline_basis.py`: Implements the Cox-de Boor recursion for evaluating univariate basis functions $N_{i,p}(u)$. This is the "low-level" layer.
2. `b_spline.py`: Defines the `BSpline` class as a wrapper managing a list of `BSplineBasis`. It extends methods to multivariate bases via tensor product. The mapping $S(\mathbf{u}) = \sum R_i(\mathbf{u})\mathbf{P}_i$ uses control points structured as tensors of order $n+1$ shaped `[dim,` $n_\xi$`,` $n_\eta$`,` $n_\zeta$`, ...]`.

3. `multi_patch_b_spline.py`: Manages complex assemblies of multiple patches and ensures conformal connectivity between domains.

### 3.2 Performance and Visualization

- `plot()` **routing**: The `plot()` method is smart. It automatically calls `plotPV()` (PyVista engine) for smooth 3D interactivity, or switches to `plotMPL()` (Matplotlib engine) for 2D plots or if PyVista is not installed.
- **Performance**: Acceleration via the `@njit` decorator (Numba) is handled under the hood. Users interact with standard class methods that transparently invoke these compiled functions for fast computation.

**AI Lever and Documentation**

The full documentation is available via `docs/index.html` or the file `doc.pdf`.
**Tip:** You can provide `doc.pdf` to an AI (ChatGPT, Claude, Gemini). It can explain complex signatures or generate quick usage examples to test. From experience, the documentation is sufficient for AI models to produce fairly accurate code.

## 4 Working with Examples and Hands-on Practice

Mastery of the library relies on progressively modifying scripts in the `exemples/tutorial/` folder. Each step presents a technical challenge to validate understanding of concepts.

### 4.1 Step 1: Foundations (`01_basis_functions.py`)

"Low-level" approach. We manipulate the `BSplineBasis` object directly to understand the univariate parametric space without control point abstraction.

- **Concept:** Manually build the geometric mapping via the matrix product $y = \mathbf{NP}$ to link basis functions to spatial coordinates.
- **Challenge:** Create a degree-1 basis with a single element, refine it to degree 3 with 4 elements, then manually generate a set of control points to visualize the result. *Optional: Observe that knot insertion and degree elevation operations do not commute.*

### 4.2 Step 2: The `BSpline` Abstraction (`02_curve_construction.py`)

Introduction of the `BSpline` class, a wrapper simplifying joint management of multivariate bases and associated geometry.

- **Concept:** Using control point tensors of shape `[dim,` $n_\xi$`]`. This structure allows intuitive data access without manual flattening.
- **Challenge:** Create a degree-1 `BSpline` object. Refine it by inserting knots **then** elevating the degree. Apply random noise to control points and display the resulting curve. *Optional: Display the basis functions at each step.*

### 4.3 Step 3: Moving to Surfaces (`03_surface_wave.py`)

Extending tensor logic to bivariate bases for surface generation.

- **Concept:** Manipulating tensors `[dim,` $n_\xi$`,` $n_\eta$`]`. The `plot()` interface automatically routes to 3D interactivity (PyVista) or 2D (Matplotlib).
- **Challenge:** Create a degree-2 planar surface. Elevate degree only along $\xi$ and insert knots only along $\eta$. Then deform control points to obtain a "saddle" surface (e.g., apply $z = x^2 - y^2$ to the control points) and visualize in 3D.

### 4.4 Step 4: Operators and Least Squares (`04_least_squares.py`)

Using the library as a numerical tool for shape fitting.

- **Concept:** Extract matrix operators $\mathbf{N}$ via `spline.DN(XI)` to solve linear systems like $\mathbf{NP} = \mathbf{X}_{target}$.
- **Challenge:** Based on the disk example, project the following "bell" function onto a bivariate B-spline: $f(\xi, \eta) = \frac{8}{\pi} e^{-8\left[(\xi-\frac{1}{2})^2+(\eta-\frac{1}{2})^2\right]}$. Here, the physical dimension is 1 (scalar elevation field). Visually compare the residual when doubling the number of knots and then elevating the basis degree.

### 4.5 Step 5: Multi-Patch Topology (`05_multipatch.py`)

Managing complex assemblies via `MultiPatchBSplineConnectivity`.

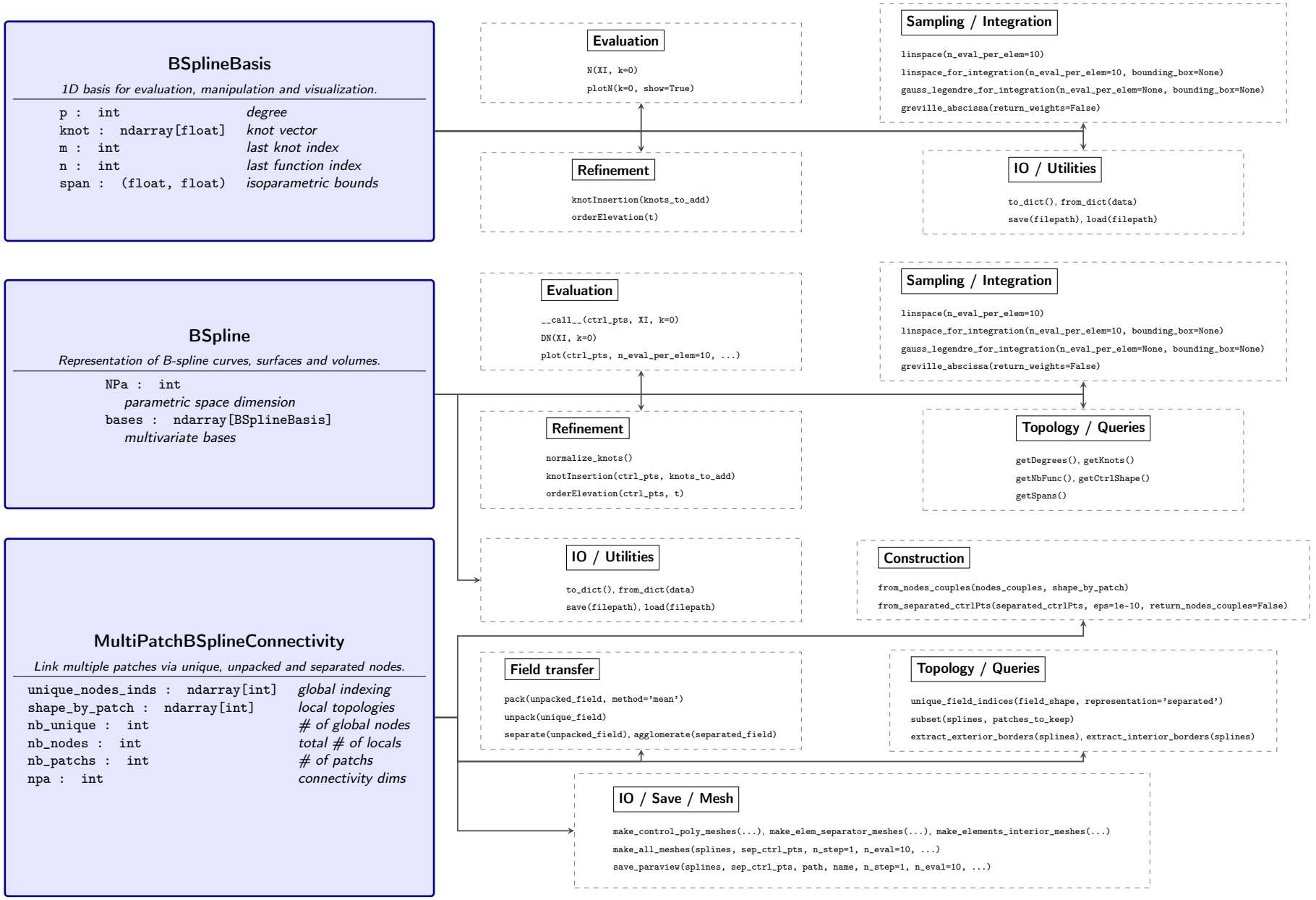
- **Concept:** Distinguishing "unique" control points (global topology) from "separated" points (local data per patch).
- **Challenge:** Generate an assembly of two planar patches sharing an edge. Apply random displacement to control points. Compare ParaView export when manipulating *unique* points (continuity preserved) versus *separated* points (crack opening). *Optional: Adapt this assembly to a complex geometry of your choice.*

## 5 Conclusion

The `bsplyne` library was designed to balance the mathematical rigor of B-splines with the numerical performance required for computations on complex geometries. Whether for mesh generation, surface approximation, or structural analysis, the layered structure (`BSplineBasis` → `BSpline` → `MultiPatchBSplineConnectivity`) allows for progressive scaling.

Beyond simple geometric construction, the strength of the tool lies in its ability to manipulate fields analytically. The `.DN()` method allows instant access to partial derivatives of any order with respect to the parametric space (e.g., `k=[0, 0, 1]` for a directional derivative along $\zeta$ in a trivariate `BSpline`).

By mastering the manipulation of control point tensors and these derivative operators, you gain a powerful lever to tackle complex physical problems efficiently, with near-C performance thanks to JIT compilation.


BSplineBasis
1D basis for evaluation, manipulation and visualization.
p : int degree
knot : ndarray[float] knot vector
m : int last knot index
n : int last function index
span : (float, float) isoparametric bounds
Evaluation
N(XI, k=0)
plotN(k=0, show=True)
Refinement
knotInsertion(knots_to_add)
orderElevation(t)
Sampling / Integration
linspace(n_eval_per_elem=10)
linspace_for_integration(n_eval_per_elem=10, bounding_box=None)
gauss_legendre_for_integration(n_eval_per_elem=None, bounding_box=None)
greville_abscissa(return_weights=False)
IO / Utilities
to_dict(), from_dict(data)
save(filepath), load(filepath)
BSpline
Representation of B-spline curves, surfaces and volumes.
NPa : int
parametric space dimension
bases : ndarray[BSplineBasis]
multivariate bases
Evaluation
__call__(ctrl_pts, XI, k=0)
DN(XI, k=0)
plot(ctrl_pts, n_eval_per_elem=10, ...)
Refinement
normalize_knots()
knotInsertion(ctrl_pts, knots_to_add)
orderElevation(ctrl_pts, t)
Sampling / Integration
linspace(n_eval_per_elem=10)
linspace_for_integration(n_eval_per_elem=10, bounding_box=None)
gauss_legendre_for_integration(n_eval_per_elem=None, bounding_box=None)
greville_abscissa(return_weights=False)
Topology / Queries
getDegrees(), getKnots()
getNbFunc(), getCtrlShape()
getSpans()
IO / Utilities
to_dict(), from_dict(data)
save(filepath), load(filepath)
Construction
from_nodes_couples(nodes_couples, shape_by_patch)
from_separated_ctrlPts(separated_ctrlPts, eps=1e-10, return_nodes_couples=False)
MultiPatchBSplineConnectivity
Link multiple patches via unique, unpacked and separated nodes.
unique_nodes_inds : ndarray[int] global indexing
shape_by_patch : ndarray[int] local topologies
nb_unique : int # of global nodes
nb_nodes : int total # of locals
nb_patchs : int # of patchs
npa : int connectivity dims
Field transfer
pack(unpacked_field, method='mean')
unpack(unique_field)
separate(unpacked_field), agglomerate(separated_field)
Topology / Queries
unique_field_indices(field_shape, representation='separated')
subset(splines, patches_to_keep)
extract_exterior_borders(splines), extract_interior_borders(splines)
IO / Save / Mesh
make_control_poly_meshes(...), make_elem_separator_meshes(...), make_elements_interior_meshes(...)
make_all_meshes(splines, sep_ctrl_pts, n_step=1, n_eval=10, ...)
save_paraview(splines, sep_ctrl_pts, path, name, n_step=1, n_eval=10, ...)