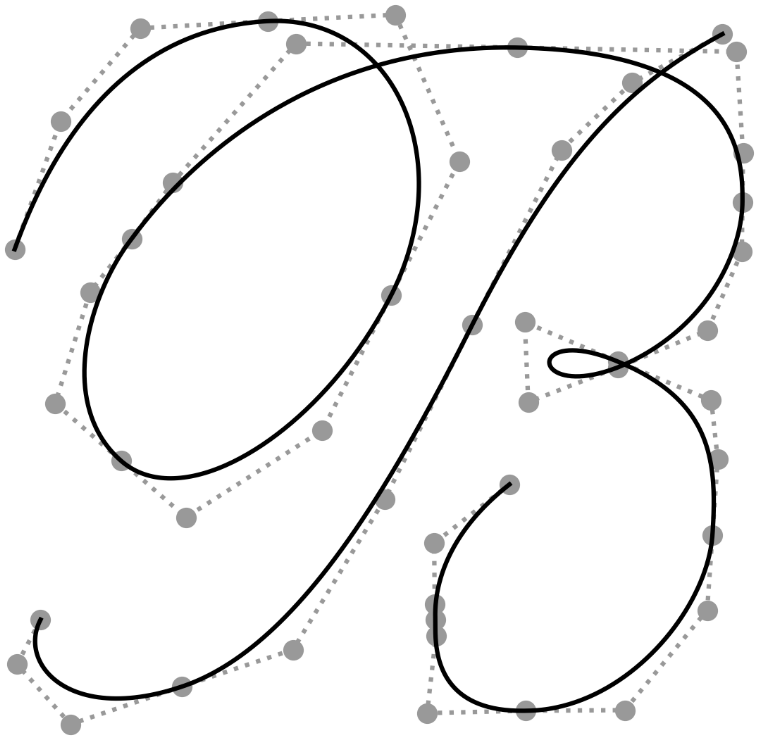
bsplyne

bsplyne



bsplyne is a Python library for working with N-dimensional B-splines. It implements the Cox-de Boor algorithm for basis evaluation, order elevation, knot insertion, and provides a connectivity class for multi-patch structures. Additionally, it includes visualization tools with export capabilities to Paraview.

> **Note:** This library is not yet available on PyPI. To install, please clone the repository and install it manually.

Installation

Since **bsplyne** is not yet on PyPI, you can install it locally as follows:

```
git clone https://github.com/Dorian210/bsplyne
cd bsplyne
pip install -e .
```

Dependencies

Make sure you have the following dependencies installed:

- `numpy`
- `numba`
- `scipy`
- `matplotlib`
- `meshio`
- `tqdm`
- `pathos`

Main Modules

- **BSplineBasis**
  Implements B-spline basis function evaluation using the Cox-de Boor recursion formula.

- **BSpline**
  Provides methods for creating and manipulating N-dimensional B-splines, including order elevation and knot

insertion.

- **MultiPatchBSplineConnectivity**
  Manages the connectivity between multiple N-dimensional B-spline patches.

- **CouplesBSplineBorder** (less documented)
  Handles coupling between B-spline borders.

Examples

Several example scripts demonstrating the usage of **bsplyne** can be found in the `examples/` directory. These scripts cover:

- Basis evaluation on a curved line
- Plotting with Matplotlib
- Order elevation
- Knot insertion
- Surface examples
- Exporting to Paraview

Documentation

The full API documentation is available in the `docs/` directory of the project or via the online documentation portal.

Contributions

At the moment, I am not actively reviewing contributions. However, if you encounter issues or have suggestions, feel free to open an issue.

License

- View Source

bsplyne.b_spline

- View Source

class BSpline:

- View Source

B-Spline from a `NPa` -D parametric space into a NPh-D physical space. NPh is not an attribute of this class.

Attributes

- **NPa** (int): Parametric space dimension.
- **bases** (numpy.array of BSplineBasis): `numpy . array` containing a `BSplineBasis` instance for each of the `NPa` axis of the parametric space.

BSpline(degrees, knots)

- View Source

Parameters

- **degrees** (numpy.array of int): Contains the degrees of the B-spline in each parametric dimension.
- **knots** (list of numpy.array of float): Contains the knot vectors of the B-spline for each parametric dimension.

Returns

- **BSpline** (BSpline instance): Contains the `BSpline` object created.

Examples

Creation of a 2D shape as a `BSpline` instance :

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> BSpline(degrees, knots)
```

NPa

bases

@classmethod
def from_bases(cls, bases):

- View Source

Create a `BSpline` instance from an array of `BSplineBasis` .

Parameters

- **bases** (numpy.ndarray of BSplineBasis): The array of `BSplineBasis` instances.

Returns

- **BSpline**: Contains the `BSpline` object created.

def getDegrees(self):

- View Source

Returns the degree of each basis in the parametric space.

Parameters

- **None**

Returns

- **degrees** (numpy.array of int): Contains the degrees of the B-spline.

Examples

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> spline = BSpline(degrees, knots)
>>> spline.getDegrees()
array([2, 2])
```

def getKnots(self):

- View Source

Returns the knot vector of each basis in the parametric space.

Parameters

- **None**

Returns

- **knots** (list of numpy.array of float): Contains the knot vectors of the B-spline.

Examples

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> spline = BSpline(degrees, knots)
>>> spline.getKnots()
[array([0. , 0. , 0. , 0.5, 1. , 1. , 1. ]),
 array([0. , 0. , 0. , 0.5, 1. , 1. , 1. ])]
```

def getNbFunc(self):

- View Source

Compute the number of basis functions of the spline.

Returns

- **int**: Number of basis functions.

def getSpans(self):

- View Source

Return the span of each basis in the parametric space.

Parameters

- **None**

Returns

- **spans** (list of tuple(float, float)): Contains the span of the B-spline.

def linspace(self, n_eval_per_elem=10):

- View Source

Generate `NPa` sets of xi values over the span of each basis.

Parameters

- **n_eval_per_elem** (numpy.array of int or int, optional): Number of values per element over each parametric axis, by default 10

Returns

- **XI** (tuple of numpy.array of float): Set of xi values over each span.

def linspace_for_integration(self, n_eval_per_elem=10, bounding_box=**None**):

- View Source

Generate `NPa` sets of xi values over the span of the basis, centerered on intervals of returned lengths.

Parameters

- **n_eval_per_elem** (numpy.array of int or int, optional): Number of values per element over each parametric axis, by default 10
- **bounding_box** (numpy.array of float , optional): Lower and upper bounds on each axis, by default [[xi0, xin], [eta0, etan], ...]

Returns

- **XI** (tuple of numpy.array of float): Set of xi values over each span.
- **dXI** (tuple of numpy.array of float): Set of integration weight of each point in `XI` .

def gauss_legendre_for_integration(self, n_eval_per_elem=**None**, bounding_box=**None**):

- View Source

Generate a set of xi values with their coresponding weight acording to the Gauss Legendre integration method over a given bounding box.

Parameters

- **n_eval_per_elem** (numpy.array of int, optional): Number of values per element over each parametric axis, by default `self.getDegrees() + 1`
- **bounding_box** (numpy.array of float, optional): Lower and upper bounds, by default `self` . `span`

Returns

- **XI** (tuple of numpy.array of float): Set of xi values over each span.
- **dXI** (tuple of numpy.array of float): Set of integration weight of each point in `XI` .

def normalize_knots(self):

- View Source

Maps the knots vectors to [0, 1].

def DN(self, XI, k=0):

- View Source

Compute the `k` -th derivative of the B-spline basis at the points in the parametric space given as input such that a dot product with the reshaped and transposed control points evaluates the B-spline.

Parameters

- **XI** (numpy.array of float or tuple of numpy.array of float): If `numpy` . `array` of `float` , contains the `NPa` -uplets of parametric coordinates as [[xi_0, ...], [eta_a, ...], ...]. Else, if `tuple` of `numpy` . `array` of `float` , contains the `NPa` parametric coordinates as [[xi_0, ...], [eta_0, ...], ...].
- **k** (list of int or int, optional): If `numpy` . `array` of `int` , or if k is 0, compute the `k` -th derivative of the B-spline basis evaluated on each axis of the parametric space. If `int` , compute the `k` -th derivative along every axis. For example, if `k` is 1, compute the gradient, if `k` is 2, compute the hessian, and so on. , by default 0

Returns

- **DN** (scipy.sparse.csr_matrix of float or numpy.array of scipy.sparse.csr_matrix of float): Contains the basis of the B-spline.

Examples

Evaluation of a 2D BSpline basis on these `XI` values : [[0, 0.5], [1, 0]]

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),
             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> XI = np.array([[0, 0.5], [1, 0]], dtype='float')
>>> spline.DN(XI, [0, 0]).A
array([[0. , 0. , 0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],

       [0. , 0. , 0. , 0. , 0.5, 0. , 0. , 0. , 0.5, 0. , 0. , 0. , 0. , 0. , 0. , 0. ]])
```

Evaluation of the 2D BSpline basis's derivative along the first axis :

```
>>> XI = (np.array([0, 0.5], dtype='float'),
          np.array([1], dtype='float'))
>>> spline.DN(XI, [1, 0]).A
array([[ 0., -0., -0., -4.,  0.,  0.,  0.,  4.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],

       [ 0.,  0.,  0.,  0., -2.,  0.,  0.,  0.,  2.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

def knotInsertion(
self,
ctrlPts,
knots_to_add: Iterable[Union[numpy.ndarray[Any, numpy.dtype[numpy.float64]], int]]
):

- View Source

Add the knots passed in parameter to the knot vector and modify the attributes so that the evaluation of the spline stays the same.

Parameters

- **ctrlPts** (numpy.array of float): Contains the control points of the B-spline as [X, Y, Z, ...]. Its shape : (NPh, nb elem for dim 1, ..., nb elem for dim `NPa`)
- **knots_to_add** (Iterable[Union[npt.NDArray[np.float64], int]]): Refinement on each axis : If `NDArray` , contains the knots to add on said axis. It must not contain knots outside of the old knot vector's interval. If `int` , correspond to the number of knots to add in each B-spline element.

Returns

- **ctrlPts** (numpy.array of float): Contains the control points of the B-spline as [X, Y, Z, ...]. Its shape : (NPh, nb elem for dim 1, ..., nb elem for dim `NPa`)

Examples

Knot insertion on a 2D BSpline in a 3D space :

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> ctrlPts = np.random.rand(3, 4, 4)
>>> spline = BSpline(degrees, knots)
>>> knots_to_add = [np.array([0.5, 0.75], dtype='float'),
                    np.array([], dtype='float')]
>>> ctrlPts = pline.knotInsertion(ctrlPts, knots_to_add)
```

def orderElevation(self, ctrlPts, t):

- View Source

Performs the order elevation algorithm on every B-spline basis and apply the changes to the control points of the B-spline.

Parameters

- **ctrlPts** (numpy.array of float): Contains the control points of the B-spline as [X, Y, Z, ...]. Its shape : (NPh, nb

elem for dim 1, ..., nb elem for dim `NPa`)
- **t** (numpy.array of int): New degree of each B-spline basis will be its current degree plus the value of `t` corresponding.

Returns

- **ctrlPts** (numpy.array of float): Contains the control points of the B-spline as [X, Y, Z, ...]. Its shape : (NPh, nb elem for dim 1, ..., nb elem for dim `NPa`)

Examples

Order elevation on a 2D BSpline in a 3D space :

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> ctrlPts = np.random.rand(3, 4, 4)
>>> spline = BSpline(degrees, knots)
>>> ctrlPts = spline.knotInsertion(ctrlPts, [1, 0])
```

def greville_abscissa(self, return_weights=**False**): • View Source

Compute the Greville abscissa.

Parameters

- **return_weights** (bool, optional): If `True` , return the weight, the length of the span of the basis function corresponding to each abscissa, by default `False`

Returns

- **greville** (list of np.array of float): Greville abscissa on each parametric axis.
- **weights** (list of np.array of float): Span of each basis function on each parametric axis.

def saveParaview(
self,
ctrlPts,
path,
name,
n_step=1,
n_eval_per_elem=10,
fields=**None**,
groups=**None**,
make_pvd=**True**,
verbose=**True**,
fiels_on_interior_only=**True**
): • View Source

Saves a plot as a set of .vtu files with a .pvd file.

Parameters

- **ctrlPts** (numpy.array of float): Contains the control points of the B-spline as [X, Y, Z, ...]. Its shape : (NPh, nb elem for dim 1, ..., nb elem for dim `NPa`)
- **path** (string): Path of the directory in which all the files to show in Paraview will be dumped.
- **name** (string): Prefix of the files created.
- **n_step** (int): Number of time steps to plot.
- **n_eval_per_elem** (numpy.array of int or int, default 10): Contains the number of evaluation of the B-spline in each direction of the parametric space for each element.
- **fields** (dict of function or of numpy.array of float, default None): Fields to plot at each time step. The name of

the field will be the dict key. If the value given is a `function`, it must take the spline and a `tuple` of parametric points that could be given to `self.DN` for example. It must return its value for each time step and on each combination of parametric points. `function`(`BSpline` spline, `tuple`(`numpy.array` of `float`) XI) -> `numpy.array` of `float` of shape (`n_step`, nb combinations of XI, size for paraview) If the value given is a `numpy.array` of `float`, the shape must be : (`n_step`, size for paraview, *`ctrlPts.shape`[1:])

- **groups** (dict of dict, default None): `dict` (out) of `dict` (in) as :
  - (out) :
    - "interior" : (in) type of `dict`,
    - "elements_borders" : (in) type of `dict`,
    - "control_points" : (in) type of `dict`.
    - other keys from the input that are not checked
  - (in) :
    - "ext" : name of the extention of the group,
    - "npart" : number of parts to plot together,
    - "nstep" : number of time steps.
- **make_pvd** (bool, default True): If True, create a PVD file for all the data in `groups`.
- **verbose** (bool, default True): If True, print the advancement state to the standard output.
- **fiels_on_interior_only** (bool, default True): Whether to save the fields on the control mesh and elements boder too.

Returns

- **groups** (dict of dict): `dict` (out) of `dict` (in) as :
  - (out) :
    - "interior" : (in) type of `dict`,
    - "elements_borders" : (in) type of `dict`,
    - "control_points" : (in) type of `dict`.
    - other keys from the input that are not checked
  - (in) :
    - "ext" : name of the extention of the group,
    - "npart" : number of parts to plot together,
    - "nstep" : number of time steps.

Examples

Save a 2D BSpline in a 3D space in the file file.pvd at the location /path/to/file :

```
>>> degrees = np.array([2, 2], dtype='int')
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),

             np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float')]

>>> ctrlPts = np.random.rand(3, 4, 4)
>>> spline = BSpline(degrees, knots)
>>> spline.saveParaview(ctrlPts, "/path/to/file", "file")
```

def getGeomdl(self, ctrl_pts):

- View Source

def plotPV(self, ctrl_pts):

- View Source

```
def plotMPL(
 self,
 ctrl_pts,
 n_eval_per_elem=10,
 ax=None,
 ctrl_color='#1b9e77',
 interior_color='#7570b3',
 elem_color='#666666',
 border_color='#d95f02'
):
```

- View Source

bsplyne.b_spline_basis

- View Source

class BSplineBasis:

- View Source

BSpline basis in 1D.

Attributes

- **p** (int): Degree of the polynomials composing the basis.
- **knot** (numpy.array of float): Knot vector of the BSpline.
- **m** (int): Last index of the knot vector.
- **n** (int): Last index of the basis : when evaluated, returns an array of size `n` + 1.
- **span** (tuple of 2 float): Interval of definition of the basis.

BSplineBasis(p, knot)

- View Source

Create a `BSplineBasis` object that can compute its basis, and the derivatives of these functions.

Parameters

- **p** (int): Degree of the BSpline.
- **knot** (numpy.array of float): Knot vector of the BSpline.

Returns

- **BSplineBasis** (BSplineBasis instance): Contains the `BSplineBasis` object created.

Examples

Creation of a `BSplineBasis` instance of degree 2 and knot vector [0, 0, 0, 1, 1, 1] :

```
>>> BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1], dtype='float'))
```

p

knot

m

n

span

def linspace(self, n_eval_per_elem=10):

- View Source

Generate a set of xi values over the span of the basis.

Parameters

- **n_eval_per_elem** (int, optional): Number of values per element, by default 10

Returns

- **numpy.array of float**: Set of xi values over the span.

def linspace_for_integration(self, n_eval_per_elem=10, bounding_box=None):

- View Source

Generate a set of xi values over the span of the basis, centerered on intervals of returned lengths.

Parameters

- **n_eval_per_elem** (int, optional): Number of values per element, by default 10

- **bounding_box** (numpy.array of float, optional): Lower and upper bounds, by default `self . span`

Returns

- **xi** (numpy.array of float): Set of xi values over the span.
- **dxi** (numpy.array of float): Integration weight of each point.

def gauss_legendre_for_integration(self, n_eval_per_elem=**None**, bounding_box=**None**):

- View Source

Generate a set of xi values with their coresponding weight acording to the Gauss Legendre integration method over a given bounding box.

Parameters

- **n_eval_per_elem** (int, optional): Number of values per element, by default `self.p + 1`
- **bounding_box** (numpy.array of float, optional): Lower and upper bounds, by default `self . span`

Returns

- **xi** (numpy.array of float): Set of xi values over the span.
- **dxi** (numpy.array of float): Integration weight of each point.

def normalize_knots(self):

- View Source

Maps the knots vector to [0, 1].

def N(self, XI, k=0):

- View Source

Compute the `k` -th derivative of the BSpline basis functions for a set of values in the parametric space.

Parameters

- **XI** (numpy.array of float): Values in the parametric space at which the BSpline is evaluated.
- **k** (int, optional): `k` -th derivative of the BSpline evaluated. The default is 0.

Returns

- **DN** (scipy.sparse.coo_matrix of float): Sparse matrix containing the values of the `k` -th derivative of the BSpline basis functions in the rows for each value of `XI` in the columns.

Examples

Evaluation of the BSpline basis on these `XI` values : [0, 0.5, 1]

```
>>> basis = BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1], dtype='float'))

>>> basis.N(np.array([0, 0.5, 1], dtype='float')).A
array([[1.  , 0.  , 0.  ],
       [0.25, 0.5 , 0.25],
       [0.  , 0.  , 1.  ]])
```

Evaluation of the 1st derivative of the BSpline basis on these `XI` values : [0, 0.5, 1]

```
>>> basis = BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1], dtype='float'))

>>> basis.N(np.array([0, 0.5, 1], dtype='float'), k=1).A
array([[-2.,  2.,  0.],
       [-1.,  0.,  1.],
       [ 0., -2.,  2.]])
```

def plotN(self, k=0, show=**True**):

- View Source

Plots the basis functions over the span.

Parameters

- **k** (int, optional): `k` -th derivative of the BSpline ploted. The default is 0.
- **show** (bool, optional): Should the plot be displayed ? The default is True.

Returns

- **None.**

def knotInsertion(self, knots_to_add):

- View Source

Performs the knot insersion process on the `BSplineBasis` instance and returns the `D` matrix.

Parameters

- **knots_to_add** (numpy.array of float): Array of knots to append to the knot vector.

Returns

- **D** (scipy.sparse.coo_matrix of float): The matrix `D` such that : newCtrlPtsCoordinate = `D` @ ancientCtrlPtsCoordinate.

Examples

Insert the knots [0.5, 0.5] to the `BSplineBasis` instance and return the operator to apply on the control points.

```
>>> basis = BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1], dtype='float'))

>>> basis.knotInsertion(np.array([0.5, 0.5], dtype='float')).A
array([[1.  , 0.  , 0.  ],
       [0.5 , 0.5 , 0.  ],
       [0.25, 0.5 , 0.25],
       [0.  , 0.5 , 0.5 ],
       [0.  , 0.  , 1.  ]])
```

The knot vector is modified (as well as n and m) :

```
>>> basis.knot
array([0. , 0. , 0. , 0.5, 0.5, 1. , 1. , 1. ])
```

def orderElevation(self, t):

- View Source

Performs the order elevation algorithm on the basis and return a linear transformation to apply on the control points.

Parameters

- **t** (int): New degree of the B-spline basis will be its current degree plus `t` .

Returns

- **STD** (scipy.sparse.coo_matrix of float): The matrix `STD` such that : newCtrlPtsCoordinate = `STD` @ ancientCtrlPtsCoordinate.

Examples

Elevate the orderof the `BSplineBasis` instance by 1 and return the operator to apply on the control points.

```
>>> basis = BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1], dtype='float'))

>>> basis.orderElevation(1).A
array([[1.        , 0.        , 0.        ],
       [0.33333333, 0.66666667, 0.        ],
       [0.        , 0.66666667, 0.33333333],
       [0.        , 0.        , 1.        ]])
```

The knot vector and the degree are modified (as well as n and m) :

```
>>> basis.knot
array([0., 0., 0., 0., 1., 1., 1., 1.])

>>> basis.p
3
```

bsplyne.multi_patch_b_spline

- View Source

@nb.njit(cache=True)
def find(parent, x):

- View Source

@nb.njit(cache=True)
def union(parent, rank, x, y):

- View Source

@nb.njit(cache=True)
def get_unique_nodes_inds(nodes_couples, nb_nodes):

- View Source

class MultiPatchBSplineConnectivity:

- View Source

Contains all the methods to link multiple B-spline patches. It uses 3 representations of the data :

- a unique representation, possibly common with other meshes, containing only unique nodes indices,
- a unpacked representation containing duplicated nodes indices,
- a separated representation containing duplicated nodes indices, separated between patches. It is here for user friendliness.

Attributes

- **unique_nodes_inds** (numpy.ndarray of int): The indices of the unique representation needed to create the unpacked one.
- **shape_by_patch** (numpy.ndarray of int): The shape of the separated nodes by patch.
- **nb_nodes** (int): The total number of unpacked nodes.
- **nb_unique_nodes** (int): The total number of unique nodes.
- **nb_patchs** (int): The number of patches.
- **npa** (int): The dimension of the parametric space of the B-splines.

MultiPatchBSplineConnectivity(unique_nodes_inds, shape_by_patch, nb_unique_nodes)

- View Source

Parameters

- **unique_nodes_inds** (numpy.ndarray of int): The indices of the unique representation needed to create the unpacked one.
- **shape_by_patch** (numpy.ndarray of int): The shape of the separated nodes by patch.
- **nb_unique_nodes** (int): The total number of unique nodes.

unique_nodes_inds: numpy.ndarray

shape_by_patch: numpy.ndarray

nb_nodes: int

nb_unique_nodes: int

nb_patchs: int

npa: int

@classmethod
def from_nodes_couples(cls, nodes_couples, shape_by_patch):

- View Source

Create the connectivity from a list of couples of unpacked nodes.

Parameters

- **nodes_couples** (numpy.ndarray of int): Couples of indices of unpacked nodes that are considered the same. Its shape should be (# of couples, 2)
- **shape_by_patch** (numpy.ndarray of int): The shape of the separated nodes by patch.

Returns

- **MultiPatchBSplineConnectivity**: Instance of `MultiPatchBSplineConnectivity` created.

```
@classmethod
def from_separated_ctrlPts(
    cls,
    separated_ctrlPts,
    eps=1e-10,
    return_nodes_couples: bool = False
):
```

- View Source

Create the connectivity from a list of control points given as a separated field by comparing every couple of points.

Parameters

- **separated_ctrlPts** (list of numpy.ndarray of float): Control points of every patch to be compared in the separated representation. Every array is of shape : (`NPh` , nb elem for dim 1, ..., nb elem for dim `npa`)
- **eps** (float, optional): Maximum distance between two points to be considered the same, by default 1e-10
- **return_nodes_couples** (bool, optional): If `True` , returns the `nodes_couples` created, by default False

Returns

- **MultiPatchBSplineConnectivity**: Instance of `MultiPatchBSplineConnectivity` created.

```
def unpack(self, unique_field):
```

- View Source

Extract the unpacked representation from a unique representation.

Parameters

- **unique_field** (numpy.ndarray): The unique representation. Its shape should be : (field, shape, ..., `self . nb_unique_nodes`)

Returns

- **unpacked_field** (numpy.ndarray): The unpacked representation. Its shape is : (field, shape, ..., `self . nb_nodes`)

```
def pack(self, unpacked_field, method='mean'):
```

- View Source

Extract the unique representation from an unpacked representation.

Parameters

- **unpacked_field** (numpy.ndarray): The unpacked representation. Its shape should be : (field, shape, ..., `self . nb_nodes`)
- **method** (str): The method used to group values that could be different

Returns

- **unique_nodes** (numpy.ndarray): The unique representation. Its shape is : (field, shape, ..., `self . nb_unique_nodes`)

```
def separate(self, unpacked_field):
```

- View Source

Extract the separated representation from an unpacked representation.

Parameters

- **unpacked_field** (numpy.ndarray): The unpacked representation. Its shape is : (field, shape, ..., `self . nb_nodes`)

Returns

- **separated_field** (list of numpy.ndarray): The separated representation. Every array is of shape : (field, shape, ..., nb elem for dim 1, ..., nb elem for dim `npa`)

def agglomerate(self, separated_field):

- View Source

Extract the unpacked representation from a separated representation.

Parameters

- **separated_field** (list of numpy.ndarray): The separated representation. Every array is of shape : (field, shape, ..., nb elem for dim 1, ..., nb elem for dim `npa`)

Returns

- **unpacked_field** (numpy.ndarray): The unpacked representation. Its shape is : (field, shape, ..., `self . nb_nodes`)

def unique_field_indices(self, field_shape, representation='separated'):

- View Source

Get the unique, unpacked or separated representation of a field's unique indices.

Parameters

- **field_shape** (tuple of int): The shape of the field. For example, if it is a vector field, `field_shape` should be (3,). If it is a second order tensor field, it should be (3, 3).
- **representation** (str, optional): The user must choose between `"unique"` , `"unpacked"` , and `"separated"` . It corresponds to the type of representation to get, by default "separated"

Returns

- **unique_field_indices** (numpy.ndarray of int or list of numpy.ndarray of int): The unique, unpacked or separated representation of a field's unique indices. If unique, its shape is (*`field_shape`* , *`self . nb_unique_nodes`* *). If unpacked, its shape is : (* `field_shape` , `self . nb_nodes`). If separated, every array is of shape : (* `field_shape` , nb elem for dim 1, ..., nb elem for dim `npa`).

def get_duplicate_unpacked_nodes_mask(self):

- View Source

Returns a boolean mask indicating which nodes in the unpacked representation are duplicates.

Returns

- **duplicate_nodes_mask** (numpy.ndarray): Boolean mask of shape (nb_nodes,) where True indicates a node is duplicated across multiple patches and False indicates it appears only once.

def extract_exterior_borders(self, splines):

- View Source

Extract exterior borders from B-spline patches.

Parameters

- **splines** (list[BSpline]): Array of B-spline patches to extract borders from.

Returns

- **border_connectivity** (MultiPatchBSplineConnectivity): Connectivity information for the border patches.
- **border_splines** (list[BSpline]): Array of B-spline patches representing the borders.
- **border_unique_to_self_unique_connectivity** (numpy.ndarray of int): Array mapping border unique nodes to original unique nodes.

Raises

- **AssertionError**: If isoparametric space dimension is less than 2.

def extract_interior_borders(self, splines): • View Source

Extract interior borders from B-spline patches where nodes are shared between patches.

Parameters

- **splines** (list[BSpline]): Array of B-spline patches to extract borders from.

Returns

- **border_connectivity** (MultiPatchBSplineConnectivity): Connectivity information for the border patches.
- **border_splines** (list[BSpline]): Array of B-spline patches representing the borders.
- **border_unique_to_self_unique_connectivity** (numpy.ndarray of int): Array mapping border unique nodes to original unique nodes.

Raises

- **AssertionError**: If parametric space dimension is less than 2.

def subset(self, splines, patches_to_keep): • View Source

Create a subset of the multi-patch B-spline connectivity by keeping only selected patches.

Parameters

- **splines** (list[BSpline]): Array of B-spline patches to subset.
- **patches_to_keep** (numpy.array of int): Indices of patches to keep in the subset.

Returns

- **new_connectivity** (MultiPatchBSplineConnectivity): New connectivity object containing only the selected patches.
- **new_splines** (list[BSpline]): Array of B-spline patches for the selected patches.
- **new_unique_to_self_unique_connectivity** (numpy.ndarray of int): Array mapping new unique nodes to original unique nodes.

```
def save_block(
    self,
    splines,
    block,
    separated_ctrl_pts,
    path,
    name,
    n_step,
    n_eval_per_elem,
    separated_fields,
    fiels_on_interior_only
):
```
• View Source

Process a block of patches, saving the meshes in their corresponding file. Each block has its own progress bar.

```
def save_paraview(
    self,
    splines,
    separated_ctrl_pts,
    path,
    name,
    n_step=1,
    n_eval_per_elem=10,
    unique_fields={},
    separated_fields=None,
    verbose=True,
    fields_on_interior_only=True
):
```

- View Source

Save the multipatch B-spline data to Paraview format using parallel processing.

Parameters

- **splines** (list[BSpline]): Array of B-spline patches to save
- **separated_ctrl_pts** (list[numpy.ndarray]): Control points for each patch in separated representation
- **path** (str): Directory path where files will be saved
- **name** (str): Base name for the saved files
- **n_step** (int, optional): Number of time steps, by default 1
- **n_eval_per_elem** (int or list[int], optional): Number of evaluation points per element, by default 10
- **unique_fields** (dict, optional): Fields in unique representation to save, by default {}
- **separated_fields** (list[dict], optional): Fields in separated representation to save, by default None
- **verbose** (bool, optional): Whether to show progress bars, by default True
- **fields_on_interior_only** (bool, optional): Whether to save fields on interior only, by default True

Raises

- **NotImplementedError**: If a callable is passed in unique_fields
- **ValueError**: If pool is not running and cannot be restarted

class CouplesBSplineBorder:

- View Source

```
CouplesBSplineBorder(
    spline1_inds,
    spline2_inds,
    axes1,
    axes2,
    front_sides1,
    front_sides2,
    transpose_2_to_1,
    flip_2_to_1,
    NPa
)
```

- View Source

spline1_inds

spline2_inds

axes1

axes2

front_sides1

front_sides2

transpose_2_to_1

flip_2_to_1

NPa

nb_couples

```
@classmethod
def extract_border_pts(cls, field, axis, front_side, field_dim=1, offset=0):
```
- View Source

```
@classmethod
def extract_border_spline(cls, spline, axis, front_side):
```
- View Source

```
@classmethod
def transpose_and_flip(cls, field, transpose, flip, field_dim=1):
```
- View Source

```
@classmethod
def transpose_and_flip_knots(cls, knots, spans, transpose, flip):
```
- View Source

```
@classmethod
def transpose_and_flip_back_knots(cls, knots, spans, transpose, flip):
```
- View Source

```
@classmethod
def transpose_and_flip_spline(cls, spline, transpose, flip):
```
- View Source

```
@classmethod
def from_splines(cls, separated_ctrl_pts, splines):
```
- View Source

```
def append(self, other):
```
- View Source

```
def get_operator_allxi1_to_allxi2(self, spans1, spans2, couple_ind):
```
- View Source

```
def get_connectivity(self, shape_by_patch):
```
- View Source

```
def get_borders_couples(self, separated_field, offset=0):
```
- View Source

```
def get_borders_couples_splines(self, splines):
```
- View Source

```
def compute_border_couple_DN(self, couple_ind: int, splines: list, XI1_border: list, k1: list):
```
- View Source

bsplyne.geometries_in_3D

- View Source

p = 2

knot = array([0. , 0. , 0. , 0.25, 0.5 , 0.75, 1. , 1. , 1.])

C = array([1.00002282, 1.00002282, 0.84721156, 0.56754107, 0.19634954, 0.])

base_quarter_circle = show

def new_quarter_circle(center, normal, radius):

- View Source

base_circle = show

def new_circle(center, normal, radius):

- View Source

base_disk = show

def new_disk(center, normal, radius):

- View Source

base_degenerated_disk = show

def new_degenerated_disk(center, normal, radius):

- View Source

base_quarter_pipe = show

def new_quarter_pipe(center_front, orientation, radius, length):

- View Source

base_pipe = show

def new_pipe(center_front, orientation, radius, length):

- View Source

base_quarter_cylinder = show

def new_quarter_cylinder(center_front, orientation, radius, length):

- View Source

base_cylinder = show

def new_cylinder(center_front, orientation, radius, length):

- View Source

base_degenerated_cylinder = show

def new_degenerated_cylinder(center_front, orientation, radius, length): • View Source

p_closed = 2

knot_closed = show

a = 0.998323859441081

b = -0.4135192822211451

C_closed = show

base_closed_circle = show

def new_closed_circle(center, normal, radius): • View Source

base_closed_disk = show

def new_closed_disk(center, normal, radius): • View Source

base_closed_pipe = show

def new_closed_pipe(center_front, orientation, radius, length): • View Source

base_closed_cylinder = show

def new_closed_cylinder(center_front, orientation, radius, length): • View Source

def new_quarter_strut(center_front, orientation, radius, length): • View Source

def new_cube(center, orientation, side_length): • View Source

bsplyne.my_wide_product

- View Source

```
@nb.njit(cache=True)
def wide_product_max_nnz(a_indptr: numpy.ndarray, b_indptr: numpy.ndarray, height: int) -> int:
```

- View Source

Compute the maximum number of nonzeros in the result.

Parameters

- **a_indptr** (np.ndarray): CSR pointer array for matrix A.
- **b_indptr** (np.ndarray): CSR pointer array for matrix B.
- **height** (int): Number of rows (must be the same for both A and B).

Returns

- **max_nnz** (int): Total number of nonzero elements in the resulting matrix.

```
@nb.njit(cache=True)
def wide_product_row(
    a_data: numpy.ndarray,
    a_indices: numpy.ndarray,
    b_data: numpy.ndarray,
    b_indices: numpy.ndarray,
    b_width: int,
    out_data: numpy.ndarray,
    out_indices: numpy.ndarray
) -> int:
```

- View Source

Compute the wide product for one row.

For each nonzero in the row of A and each nonzero in the row of B, it computes: out_index = a_indices[i] * b_width + b_indices[j] out_value = a_data[i] * b_data[j]

Parameters

- **a_data** (np.ndarray): Nonzero values for the row in A.
- **a_indices** (np.ndarray): Column indices for the row in A.
- **b_data** (np.ndarray): Nonzero values for the row in B.
- **b_indices** (np.ndarray): Column indices for the row in B.
- **b_width** (int): Number of columns in B.
- **out_data** (np.ndarray): Preallocated output array for the row's data.
- **out_indices** (np.ndarray): Preallocated output array for the row's indices.

Returns

- **off** (int): Number of nonzero entries computed for this row.

```
@nb.njit(cache=True)
def wide_product_numba(
 height: int,
 a_data: numpy.ndarray,
 a_indices: numpy.ndarray,
 a_indptr: numpy.ndarray,
 a_width: int,
 b_data: numpy.ndarray,
 b_indices: numpy.ndarray,
 b_indptr: numpy.ndarray,
 b_width: int
):
```

- View Source

Compute the row-wise wide (Khatri-Rao) product for two CSR matrices.

For each row i, the result[i, :] = kron(A[i, :], B[i, :]), i.e. the Kronecker product of the i-th rows of A and B.

Parameters

- **height** (int): Number of rows in A and B.
- **a_data** (np.ndarray): Data array for matrix A (CSR format).
- **a_indices** (np.ndarray): Indices array for matrix A.
- **a_indptr** (np.ndarray): Index pointer array for matrix A.
- **a_width** (int): Number of columns in A.
- **b_data** (np.ndarray): Data array for matrix B (CSR format).
- **b_indices** (np.ndarray): Indices array for matrix B.
- **b_indptr** (np.ndarray): Index pointer array for matrix B.
- **b_width** (int): Number of columns in B.

Returns

- **out_data** (np.ndarray): Data array for the resulting CSR matrix.
- **out_indices** (np.ndarray): Indices array for the resulting CSR matrix.
- **out_indptr** (np.ndarray): Index pointer array for the resulting CSR matrix.
- **total_nnz** (int): Total number of nonzero entries computed.

```
def my_wide_product(
 A: scipy.sparse._base.spmatrix,
 B: scipy.sparse._base.spmatrix
) -> scipy.sparse._csr.csr_matrix:
```

- View Source

Compute a "1D" Kronecker product row by row.

For each row i, the result C[i, :] = kron(A[i, :], B[i, :]). Matrices A and B must have the same number of rows.

Parameters

- **A** (scipy.sparse.spmatrix): Input sparse matrix A in CSR format.
- **B** (scipy.sparse.spmatrix): Input sparse matrix B in CSR format.

Returns

- **C** (scipy.sparse.csr_matrix): Resulting sparse matrix in CSR format with shape (A.shape[0], A.shape[1]*B.shape[1]).