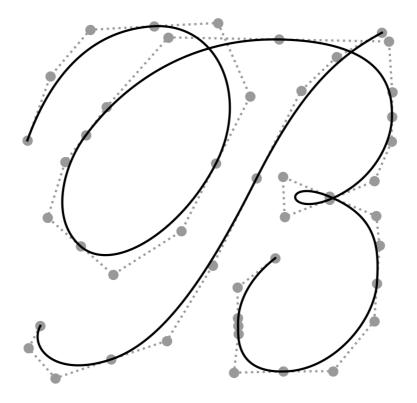# bsplyne

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

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

This project is licensed under the CeCILL License.

- View Source

bsplyne.my_wide_product

-

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

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:
```

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.

```python
@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.

```python
def my_wide_product(
    A: scipy.sparse._matrix.spmatrix,
    B: scipy.sparse._matrix.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]).

bsplyne.geometries_in_3D

def scale_rotate_translate(pts, scale_vector, axis, angle, translation_vector):

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):

base_circle = show

def new_circle(center, normal, radius):

base_disk = show

def new_disk(center, normal, radius):

base_degenerated_disk = show

def new_degenerated_disk(center, normal, radius):

base_quarter_pipe = show

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

base_pipe = show

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

base_quarter_cylinder = show

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

base_cylinder = show

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

base_degenerated_cylinder = [show]

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

- View Source

p_closed = 2

knot_closed = [show]

a = np.float64(0.998323859441081)

b = np.float64(-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.multi_patch_b_spline

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

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

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

class MultiPatchBSplineConnectivity:

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)

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):
```

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):

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):

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 make_control_poly_meshes(
  self,
  splines: Iterable[bsplyne.b_spline.BSpline],
  separated_ctrl_pts: Iterable[numpy.ndarray[numpy.floating]],
  n_eval_per_elem: Union[Iterable[int], int] = 10,
  n_step: int = 1,
  unique_fields: dict = {},
  separated_fields: Optional[dict] = **None**,
  XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = **None**,
  paraview_sizes: dict = {}
) -> list[meshio._mesh.Mesh]:

def make_elem_separator_meshes(
 self,
 splines: Iterable[bsplyne.b_spline.BSpline],
 separated_ctrl_pts: Iterable[numpy.ndarray[numpy.floating]],
 n_eval_per_elem: Union[Iterable[int], int] = 10,
 n_step: int = 1,
 unique_fields: dict = {},
 separated_fields: Optional[dict] = **None**,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = **None**,
 paraview_sizes: dict = {},
 parallel: bool = **True**,
 verbose: bool = **True**
) -> list[meshio._mesh.Mesh]:

- View Source

def make_elements_interior_meshes(
 self,
 splines: Iterable[bsplyne.b_spline.BSpline],
 separated_ctrl_pts: Iterable[numpy.ndarray[numpy.floating]],
 n_eval_per_elem: Union[Iterable[int], int] = 10,
 n_step: int = 1,
 unique_fields: dict = {},
 separated_fields: Optional[dict] = **None**,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = **None**,
 parallel: bool = **True**,
 verbose: bool = **True**
) -> list[meshio._mesh.Mesh]:

- View Source

def make_all_meshes(
 self,
 splines: Iterable[bsplyne.b_spline.BSpline],
 separated_ctrl_pts: Iterable[numpy.ndarray[numpy.floating]],
 n_step: int = 1,
 n_eval_per_elem: Union[int, Iterable[int]] = 10,
 unique_fields: dict = {},
 separated_fields: Optional[list[dict]] = **None**,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = **None**,
 verbose: bool = **True**,
 fields_on_interior_only: Union[bool, Literal['auto'], list[str]] = 'auto'
) -> tuple[list[meshio._mesh.Mesh], list[meshio._mesh.Mesh], list[meshio._mesh.Mesh]]:

- View Source

Generate all mesh representations (interior, element borders, and control points) for a multipatch B-spline geometry.

This method creates three types of meshes for visualization or analysis:

- The interior mesh representing the B-spline surface or volume.
- The element separator mesh showing the borders between elements.
- The control polygon mesh showing the control structure.

Parameters
- **splines** (Iterable[BSpline]): List of B-spline patches to process.
- **separated_ctrl_pts** (Iterable[np.ndarray[np.floating]]): Control points for each patch in separated representation.
- **n_step** (int, optional): Number of time steps to generate. By default, 1.
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each

isoparametric dimension. If an `int` is provided, the same number is used for all dimensions. If an `Iterable` is provided, each value corresponds to a different dimension. By default, 10.

- **unique_fields** (dict, optional): Fields in unique representation to visualize. By default, `{}`. Keys are field names, values are arrays (not callables nor FE fields).
- **separated_fields** (Union[list[dict], None], optional): Fields to visualize at each time step. List of `self.nb_patchs` dictionaries (one per patch) of format: { "field_name": `field_value` } where `field_value` can be either:
  1. A `np.ndarray` with shape (`n_step`, `field_size`, `self.shape_by_patch[patch]`)
  2. A `np.ndarray` with shape (`n_step`, `field_size`, `*grid_shape`)
  3. A function that computes field values (`np.ndarray[np.floating]`) at given points from the `BSpline` instance and `XI`. By default, None.
- **XI_list** (Union[None, Iterable[tuple[np.ndarray[np.floating], ...]]], optional): Parametric coordinates at which to evaluate the B-spline patches and fields. If not `None`, overrides the `n_eval_per_elem` parameter. If `None`, regular grids are generated according to `n_eval_per_elem`. By default, None.
- **verbose** (bool, optional): Whether to print progress information. By default, True.
- **fiels_on_interior_only** (Union[bool, Literal['auto'], list[str]], optionnal): Whether to include fields only on the interior mesh (`True`), on all meshes (`False`), or on specified field names. If set to `'auto'`, fields named `'u'`, `'U'`, `'displacement'` or `'displ'` are included on all meshes while others are only included on the interior mesh. By default, 'auto'.

Returns

- **tuple[list[io.Mesh], list[io.Mesh], list[io.Mesh]]**: Tuple containing three lists of `io.Mesh` objects:
  - Interior meshes for each time step.
  - Element separator meshes for each time step.
  - Control polygon meshes for each time step.

Raises

- **NotImplementedError**: If a callable is passed in `unique_fields`.
- **ValueError**: If a field in `unique_fields` does not have the correct shape.

Notes

- The isoparametric space refers to the parametric space of the B-splines.
- Fields can be visualized as scalars or vectors.
- Supports time-dependent visualization through `n_step`.
- Fields in `unique_fields` must be arrays; to use callables, use `separated_fields`.

Examples

```
>>> interior, borders, control = connectivity.make_all_meshes(splines, separated_ctrl_pts
)
```

```
def save_paraview(
 self,
 splines: Iterable[bsplyne.b_spline.BSpline],
 separated_ctrl_pts: Iterable[numpy.ndarray[numpy.floating]],
 path: str,
 name: str,
 n_step: int = 1,
 n_eval_per_elem: Union[int, Iterable[int]] = 10,
 unique_fields: dict = {},
 separated_fields: Optional[list[dict]] = None,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
 groups: Optional[dict[str, dict[str, Union[str, int]]]] = None,
 make_pvd: bool = True,
 verbose: bool = True,
 fields_on_interior_only: Union[bool, Literal['auto'], list[str]] = 'auto'
):
```

- View Source

Save multipatch B-spline visualization data as Paraview files.

This method generates three types of visualization files for a multipatch B-spline geometry:

- Interior mesh showing the B-spline surface/volume
- Element borders showing the mesh structure
- Control points mesh showing the control structure

All files are saved in VTU format, with an optional PVD file to group them for Paraview.

Parameters

- **splines** (Iterable[BSpline]): List of B-spline patches to save.
- **separated_ctrl_pts** (Iterable[np.ndarray[np.floating]]): Control points for each patch in separated representation.
- **path** (str): Directory path where the files will be saved.
- **name** (str): Base name for the output files.
- **n_step** (int, optional): Number of time steps to save. By default, 1.
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. If an `int` is provided, the same number is used for all dimensions. If an `Iterable` is provided, each value corresponds to a different dimension. By default, 10.
- **unique_fields** (dict, optional): Fields in unique representation to save. By default, `{}`. Keys are field names, values are arrays (not callables nor FE fields).
- **separated_fields** (Union[list[dict], None], optional): Fields to visualize at each time step. List of `self.nb_patchs` dictionaries (one per patch) of format: { "field_name": `field_value` } where `field_value` can be either:

  1. A numpy array with shape (`n_step`, `field_size`, `self.shape_by_patch[patch]`) where:
  - `n_step`: Number of time steps
  - `field_size`: Size of the field at each point (1 for scalar, 3 for vector)
  - `self.shape_by_patch[patch]`: Same shape as the patch's control points grid (excluding `NPh`)
  2. A numpy array with shape (`n_step`, `field_size`, `*grid_shape`) where:
  - `n_step`: Number of time steps
  - `field_size`: Size of the field at each point (1 for scalar, 3 for vector)
  - `*grid_shape`: Shape of the evaluation grid (number of points along each isoparametric axis)
  3. A function that computes field values (`np.ndarray[np.floating]`) at given points from the `BSpline` instance and `XI`, the tuple of arrays containing evaluation points for each dimension (`tuple[np.ndarray[np.floating], ...]`). The result should be an array of shape (`n_step`, `n_points`, `field_size`) where:

- - - `n_step` : Number of time steps
    - `n_points` : Number of evaluation points (n_xi × n_eta × ...)
    - `field_size` : Size of the field at each point (1 for scalar, 3 for vector)

    By default, None.

- **XI_list** (Iterable[tuple[np.ndarray[np.floating], ...]], optional): Parametric coordinates at which to evaluate the B-spline patches and fields. If not `None` , overrides the `n_eval_per_elem` parameter. If `None` , regular grids are generated according to `n_eval_per_elem` .
- **groups** (Union[dict[str, dict[str, Union[str, int]]], None], optional): Nested dictionary specifying file groups for PVD organization. Format: { "group_name": { "ext": str, # File extension (e.g., "vtu") "npart": int, # Number of parts in the group "nstep": int # Number of timesteps } } If provided, existing groups are updated; if `None` , groups are created automatically. By default, `None` .
- **make_pvd** (bool, optional): Whether to create a PVD file grouping all VTU files. By default, `True` .
- **verbose** (bool, optional): Whether to print progress information. By default, `True` .
- **fields_on_interior_only** (Union[bool, Literal['auto'], list[str]], optionnal): Whether to include fields only on the interior mesh (`True`), on all meshes (`False`), or on specified field names. If set to `'auto'` , fields named `'u'` , `'U'` , `'displacement'` or `'displ'` are included on all meshes while others are only included on the interior mesh. By default, 'auto'.

Returns

- **groups** (dict[str, dict[str, Union[str, int]]]): Updated groups dictionary with information about saved files.

Raises

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

Notes

- Creates three types of VTU files for each time step:
    - {name}_interior_{part}_{step}.vtu
    - {name}_elements_borders_{part}_{step}.vtu
    - {name}_control_points_{part}_{step}.vtu
- If `make_pvd=True` , creates a PVD file named {name}.pvd.
- Fields can be visualized as scalars or vectors in Paraview.
- The method supports time-dependent visualization through `n_step` .
- Fields in `unique_fields` must be arrays; to use callables, use `separated_fields` .

Examples

Save a multipatch B-spline visualization:

```
>>> connectivity.save_paraview(splines, separated_ctrl_pts, "./output", "multipatch"
)
```

Save with a custom separated field on a 2 patches multipatch:

```
>>> fields = [{"temperature": np.random.rand(1, 4, 4)}, {"temperature": np.random.rand(1, 7, 3)}]
>>> connectivity.save_paraview(splines, separated_ctrl_pts, "./output", "multipatch", separated_fie
)
```

class CouplesBSplineBorder:

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):

def get_borders_couples(self, separated_field, offset=0):

def get_borders_couples_splines(self, splines):

def compute_border_couple_DN(
 self,
 couple_ind: int,
 splines: list[bsplyne.b_spline.BSpline],
 XI1_border: list[numpy.ndarray],
 k1: list[int]
):

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,
 splines: list[bsplyne.b_spline.BSpline],

[bsplyne](.)**.parallel_utils**

def parallel_blocks_inner(
  funcs: Iterable[Callable],
  all_args: Iterable[tuple],
  num_blocks: int,
  verbose: bool,
  pbar_title: str,
  disable_parallel: bool,
  shared_mem_last_arg: Optional[numpy.ndarray]
) -> list:

Execute a list of functions with their corresponding argument tuples, optionally in parallel blocks.

This function performs the actual execution of tasks either sequentially or in parallel, depending on the `disable_parallel` flag. When running in parallel, the tasks are divided into `num_blocks` groups (blocks), each executed by a separate worker process. Intermediate results are temporarily saved to disk as `.npy` files to limit memory usage and are reloaded sequentially after all processes complete.

Parameters

- **funcs** (Iterable[Callable]): List of functions to execute. Must have the same length as `all_args`. Each function is called as `func(*args)` for its corresponding argument tuple.
- **all_args** (Iterable[tuple]): List of tuples, each containing the arguments for the corresponding function in `funcs`.
- **num_blocks** (int): Number of parallel blocks (i.e., worker processes) to use when `disable_parallel` is False. Determines how many subsets of tasks will be distributed among processes.
- **verbose** (bool): If True, enables progress bars and displays information about block processing and result gathering.
- **pbar_title** (str): Title prefix used for progress bar descriptions.
- **disable_parallel** (bool): If True, all tasks are executed sequentially in the current process. If False, tasks are divided into blocks and processed in parallel using a multiprocessing pool.
- **shared_mem_last_arg** (Union[np.ndarray, None]): Optional NumPy array placed in shared memory and appended automatically as the last argument of each task. This is useful for sharing large read-only data (e.g., images, meshes) without duplicating memory across processes.

Returns

- **list**: List of results obtained from applying each function to its corresponding argument tuple, preserving the original task order.

Notes

- **Sequential mode:** if `disable_parallel` is True, all functions are executed in the current process with an optional progress bar.
- **Parallel mode:**
  - The tasks are split into `num_blocks` subsets.
  - Each subset is processed by a separate worker via `multiprocessing.Pool`.
  - Each worker writes its results as `.npy` files inside a temporary subfolder.
  - After all workers complete, results are reloaded in the original order, and temporary files and folders are deleted.
- **Shared memory:** if `shared_mem_last_arg` is provided, it is stored once in shared memory and accessible by all workers, avoiding redundant copies of large data arrays.
- Compatible with both standard Python terminals and Jupyter notebooks (adaptive progress bars).
- Intended for internal use by higher-level orchestration functions such as `parallel_blocks()`.

```
def parallel_blocks(
 funcs: Union[Callable, Iterable[Callable]],
 all_args: Optional[Iterable[tuple]] = None,
 num_blocks: Optional[int] = None,
 verbose: bool = True,
 pbar_title: str = 'Processing blocks',
 disable_parallel: bool = False,
 est_proc_cost: float = 0.5,
 shared_mem_last_arg: Optional[numpy.ndarray] = None
 ) -> list:
```

Execute a set of independent tasks sequentially or in parallel, depending on their estimated cost.

The function evaluates the runtime of the first task to decide whether parallelization is worth the overhead of process creation. If parallel execution is deemed beneficial, the remaining tasks are distributed across several blocks processed in parallel. Otherwise, all tasks are executed sequentially. This strategy is especially useful when task runtimes are variable or short compared to process spawning costs.

Parameters

- **funcs** (Union[Callable, Iterable[Callable]]): Function or list of functions to execute.
    - If a single function is provided, it will be applied to each argument tuple in `all_args` .
    - If a list of functions is provided, it must have the same length as `all_args` , allowing each task to use a distinct callable.
- **all_args** (Union[Iterable[tuple], None], optional): Iterable of tuples containing the positional arguments for each function call. If the function takes no arguments, set `all_args` to `None` (defaults to empty tuples).
- **num_blocks** (Union[int, None], optional): Number of parallel blocks (i.e., worker processes) to use. Defaults to half the number of CPU cores. A value of 1 forces sequential execution.
- **verbose** (bool, optional): If True, displays timing information and progress bars. Default is True.
- **pbar_title** (str, optional): Title prefix displayed in the progress bar. Default is "Processing blocks".
- **disable_parallel** (bool, optional): If True, forces all computations to run sequentially regardless of estimated profitability. Default is False.
- **est_proc_cost** (float, optional): Estimated process creation cost in seconds. Used to determine whether parallelization will yield a net speedup. Default is 0.5 s.
- **shared_mem_last_arg** (Union[np.ndarray, None], optional): Shared-memory NumPy array to be appended automatically as the last argument in each task. This allows tasks to read from a large, read-only array without duplicating it in memory. Default is None.

Returns

- **list**: List of results, one per task, preserving the input order.

Notes

- The first task is executed sequentially to estimate its runtime.
- Parallelization is enabled only if the estimated time saved exceeds the cost of process creation.
- When parallel mode is used, tasks are executed in blocks, and intermediate results are stored temporarily on disk to limit memory usage, then reloaded and combined sequentially.
- Compatible with Jupyter progress bars (`tqdm.notebook`).

[bsplyne](). b_spline_basis

class BSplineBasis:

BSpline basis in 1D.

A class representing a one-dimensional B-spline basis with functionality for evaluation, manipulation and visualization of basis functions. Provides methods for basis function evaluation, derivatives computation, knot insertion, order elevation, and integration point generation.

Attributes

- **p** (int): Degree of the polynomials composing the basis.
- **knot** (np.ndarray[np.floating]): Knot vector defining the B-spline basis. Contains non-decreasing sequence of isoparametric coordinates.
- **m** (int): Last index of the knot vector (size - 1).
- **n** (int): Last index of the basis functions. When evaluated, returns an array of size `n + 1`.
- **span** (tuple[float, float]): Interval of definition of the basis `(knot[p], knot[m - p])`.

Notes

The basis functions are defined over the isoparametric space specified by the knot vector. Basis function evaluation and manipulation methods use efficient algorithms based on Cox-de Boor recursion formulas.

See Also

`numpy.ndarray` : Array type used for knot vector storage

`scipy.sparse` : Sparse matrix formats used for basis function evaluations

BSplineBasis(p: int, knot: Iterable[float])

Initialize a B-spline basis with specified degree and knot vector.

Parameters

- **p** (int): Degree of the B-spline polynomials.
- **knot** (Iterable[float]): Knot vector defining the B-spline basis. Must be a non-decreasing sequence of real numbers.

Returns

- **BSplineBasis**: The initialized `BSplineBasis` instance.

Notes

The knot vector must satisfy these conditions:

- Size must be at least `p + 2`
- Must be non-decreasing
- For non closed B-spline curves, first and last knots must have multiplicity `p + 1`

The basis functions are defined over the isoparametric space specified by the knot vector. The span of the basis is [`knot[p]` , `knot[m - p]`], where `m` is the last index of the knot vector.

Examples

Create a quadratic B-spline basis with uniform knot vector:

```
>>> basis = BSplineBasis(2, [0., 0., 0., 1., 1., 1.])
```

p: int

knot: numpy.ndarray[numpy.floating]

m: int

n: int

span: tuple[float, float]

def linspace(self, n_eval_per_elem: int = 10) -> numpy.ndarray[numpy.floating]:

Generate evenly spaced points over the basis span.

Creates a set of evaluation points by distributing them uniformly within each knot span (element) of the basis. Points are evenly spaced within elements but spacing may vary between different elements.

Parameters

- **n_eval_per_elem** (int, optional): Number of evaluation points per element. By default, 10.

Returns

- **xi** (np.ndarray[np.floating]): Array of evenly spaced points in isoparametric coordinates over the basis span.

Notes

The method:

1. Identifies unique knot spans (elements) in the isoparametric space
2. Distributes points evenly within each element
3. Combines points from all elements into a single array

Examples

```
>>> basis = BSplineBasis(2, [0., 0., 0., 1., 1., 1.])

>>> basis.linspace(5)
array([0. , 0.2, 0.4, 0.6, 0.8, 1. ])
```

def linspace_for_integration(
 self,
 n_eval_per_elem: int = 10,
 bounding_box: Optional[tuple[float, float]] = **None**
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]:

Generate points and weights for numerical integration over knot spans in the isoparametric space. Points are evenly distributed within each element (knot span), though spacing may vary between different elements.

Parameters

- **n_eval_per_elem** (int, optional): Number of evaluation points per element. By default, 10.
- **bounding_box** (Union[tuple[float, float], None], optional): Lower and upper bounds for integration. If `None`, uses the span of the basis. By default, None.

Returns

- **xi** (np.ndarray[np.floating]): Array of integration points in isoparametric coordinates, evenly spaced within each element.
- **dxi** (np.ndarray[np.floating]): Array of corresponding integration weights, which may vary between elements

Notes

The method generates integration points by:

1. Identifying unique knot spans (elements) in the isoparametric space
2. Distributing points evenly within each element

3. Computing appropriate weights for each point based on the element size

When `bounding_box` is provided, integration is restricted to that interval, and elements are adjusted accordingly.

Examples

```
>>> basis = BSplineBasis(2, [0, 0, 0, 1, 1, 1])
>>> xi, dxi = basis.linspace_for_integration(5
)
```

def gauss_legendre_for_integration(
 self,
 n_eval_per_elem: Optional[int] = **None**,
 bounding_box: Optional[tuple[float, float]] = **None**
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]:

- View Source

Generate Gauss-Legendre quadrature points and weights for numerical integration over the B-spline basis.

Parameters

- **n_eval_per_elem** (Union[int, None], optional): Number of evaluation points per element. If `None`, takes the value `self.p + 1`. By default, None.
- **bounding_box** (Union[tuple[float, float], None], optional): Lower and upper bounds for integration. If `None`, uses the span of the basis. By default, None.

Returns

- **xi** (np.ndarray[np.floating]): Array of Gauss-Legendre quadrature points in isoparametric coordinates.
- **dxi** (np.ndarray[np.floating]): Array of corresponding integration weights.

Notes

The method generates integration points and weights by:

1. Identifying unique knot spans (elements) in the isoparametric space
2. Computing Gauss-Legendre points and weights for each element
3. Transforming points and weights to account for element size

When `bounding_box` is provided, integration is restricted to that interval.

Examples

```
>>> basis = BSplineBasis(2, [0, 0, 0, 1, 1, 1])
>>> xi, dxi = basis.gauss_legendre_for_integration(3
)
>>> xi  # Gauss-Legendre points
array([0.11270167, 0.5       , 0.88729833])
>>> dxi  # Integration weights
array([0.27777778, 0.44444444, 0.27777778])
```

def normalize_knots(self**):**

- View Source

Normalize the knot vector to the interval [0, 1].

Maps the knot vector to the unit interval by applying an affine transformation that preserves the relative spacing between knots. Updates both the knot vector and span attributes.

Examples

```
>>> basis = BSplineBasis(2, [0., 0., 0., 2., 2., 2.])

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

def N(
 self,
 XI: numpy.ndarray[numpy.floating],
 k: int = 0
) -> scipy.sparse._coo.coo_matrix: • View Source

Compute the k-th derivative of the B-spline basis functions at specified points.

Parameters

- **XI** (np.ndarray[np.floating]): Points in the isoparametric space at which to evaluate the basis functions.
- **k** (int, optional): Order of the derivative to compute. By default, 0.

Returns

- **DN** (sps.coo_matrix): Sparse matrix containing the k-th derivative values. Each row corresponds to an evaluation point, each column to a basis function. Shape is (`XI.size` , `n + 1`).

Notes

Uses Cox-de Boor recursion formulas to compute basis function derivatives. Returns values in sparse matrix format for efficient storage and computation.

Examples

```
>>> basis = BSplineBasis(2, [0., 0., 0., 1., 1., 1.])
>>> basis.N([0., 0.5, 1.]).A  # Evaluate basis functions
array([[1.  , 0.  , 0.  ],
    [0.25, 0.5 , 0.25],
    [0.  , 0.  , 1.  ]])
>>> basis.N([0., 0.5, 1.], k=1).A  # Evaluate first derivatives

array([[-2.,  2.,  0.],
    [-1.,  0.,  1.],
    [ 0., -2.,  2.]])
```

def plotN(self, k: int = 0, show: bool = **True**):   • View Source

Plot the B-spline basis functions or their derivatives over the span.

Visualizes each basis function N_i(ξ) or its k-th derivative over its support interval using matplotlib. The plot includes proper LaTeX labels and a legend if there are 10 or fewer basis functions.

Parameters

- **k** (int, optional): Order of derivative to plot. By default, 0 (plots the basis functions themselves).
- **show** (bool, optional): Whether to display the plot immediately. Can be useful to add more stuff to the plot. By default, True.

Notes

- Uses adaptive sampling with points only in regions where basis functions are non-zero
- Plots each basis function in a different color with LaTeX-formatted labels
- Legend is automatically hidden if there are more than 10 basis functions
- The x-axis represents the isoparametric coordinate ξ

Examples

```
>>> basis = BSplineBasis(2, [0., 0., 0., 1., 1., 1.])

>>> basis.plotN()  # Plot basis functions
>>> basis.plotN(k=1)  # Plot first derivatives
```

def knotInsertion(
 self,
 knots_to_add: numpy.ndarray[numpy.floating]
) -> scipy.sparse._coo.coo_matrix:

Insert knots into the B-spline basis and return the transformation matrix.

Parameters

- **knots_to_add** (np.ndarray[np.floating]): Array of knots to insert into the knot vector.

Returns

- **D** (sps.coo_matrix): Transformation matrix such that new control points = `D` @ old control points.

Notes

Updates the basis by:

- Inserting new knots into the knot vector
- Incrementing `m` and `n` by the number of inserted knots
- Computing transformation matrix `D` for control points update

Examples

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

>>> basis.knotInsertion(np.array([0.33, 0.67], dtype='float')).A
array([[1.    , 0.    , 0.    ],
       [0.67  , 0.33  , 0.    ],
       [0.2211, 0.5578, 0.2211],
       [0.    , 0.33  , 0.67  ],
       [0.    , 0.    , 1.    ]])
```

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

```
>>> basis.knot
array([0.  , 0.  , 0.  , 0.33, 0.67, 1.  , 1.  , 1.  ])
```

def orderElevation(self, t: int) -> scipy.sparse._coo.coo_matrix:

Elevate the polynomial degree of the B-spline basis and return the transformation matrix.

Parameters

- **t** (int): Amount by which to increase the basis degree. New degree will be current degree plus `t` .

Returns

- **STD** (sps.coo_matrix): Transformation matrix for control points such that: new_control_points = `STD` @ old_control_points

Notes

The method:

1. Separates B-spline into Bézier segments via knot insertion
2. Elevates degree of each Bézier segment
3. Recombines segments into elevated B-spline via knot removal
4. Updates basis degree, knot vector and other attributes

Examples

Elevate quadratic basis to cubic:

```
>>> 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](#).b_spline

• View Source

class BSpline:

• View Source

BSpline class for representing and manipulating B-spline curves, surfaces and volumes.

A class providing functionality for evaluating, manipulating and visualizing B-splines of arbitrary dimension. Supports knot insertion, order elevation, and visualization through Paraview and Matplotlib.

Attributes

- **NPa** (int): Dimension of the isoparametric space.
- **bases** (np.ndarray[BSplineBasis]): Array containing `BSplineBasis` instances for each isoparametric dimension.

Notes

- Supports B-splines of arbitrary dimension (curves, surfaces, volumes, etc.)
- Provides methods for evaluation, derivatives, refinement and visualization
- Uses Cox-de Boor recursion formulas for efficient basis function evaluation
- Visualization available through Paraview (VTK) and Matplotlib

See Also

`BSplineBasis` : Class representing one-dimensional B-spline basis functions

`numpy.ndarray` : Array type used for control points and evaluations

`scipy.sparse` : Sparse matrix formats used for basis function evaluations

BSpline(
degrees: Iterable[int],
knots: Iterable[numpy.ndarray[numpy.floating]]
)

• View Source

Initialize a `BSpline` instance with specified degrees and knot vectors.

Creates a `BSpline` object by generating basis functions for each isoparametric dimension using the provided polynomial degrees and knot vectors.

Parameters

- **degrees** (Iterable[int]): Collection of polynomial degrees for each isoparametric dimension. The length determines the dimensionality of the parametric space (`NPa`). For example:
  - [p] for a curve
  - [p, q] for a surface
  - [p, q, r] for a volume
  - ...
- **knots** (Iterable[np.ndarray[np.floating]]): Collection of knot vectors for each isoparametric dimension. Each knot vector must be a numpy array of `floats` . The number of knot vectors must match the number of degrees. For a degree `p` , the knot vector must have size `m + 1` where `m>=p` .

Notes

- The number of control points in each dimension will be `m - p` where `m` is the size of the knot vector minus 1 and `p` is the degree
- Each knot vector must be non-decreasing
- The multiplicity of each knot must not exceed `p + 1`

Examples

Create a 2D B-spline surface:

```
>>> degrees = [2, 2]
>>> 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)
```

Create a 1D B-spline curve:

```
>>> degree = [3]
>>> knot = [np.array([0, 0, 0, 0, 1, 1, 1, 1], dtype='float')]

>>> curve = BSpline(degree, knot)
```

NPa: int

bases: numpy.ndarray[bsplyne.b_spline_basis.BSplineBasis]

@classmethod
def from_bases(
 cls,
 bases: Iterable[bsplyne.b_spline_basis.BSplineBasis]
) -> BSpline: • View Source

Create a BSpline instance from an array of `BSplineBasis` objects. This is an alternative constructor that allows direct initialization from existing basis functions rather than creating new ones from degrees and knot vectors.

Parameters
  • **bases** (Iterable[BSplineBasis]): An iterable (e.g. list, tuple, array) containing `BSplineBasis` instances. Each basis represents one parametric dimension of the resulting B-spline. The number of bases determines the dimensionality of the parametric space.

Returns
  • **BSpline**: A new `BSpline` instance with the provided basis functions.

Notes
  • The method initializes a new `BSpline` instance with empty degrees and knots
  • The bases array is populated with the provided `BSplineBasis` objects
  • The dimensionality (`NPa`) is determined by the number of basis functions

Examples

```
>>> basis1 = BSplineBasis(2, np.array([0, 0, 0, 1, 1, 1]))
>>> basis2 = BSplineBasis(2, np.array([0, 0, 0, 0.5, 1, 1, 1]))

>>> spline = BSpline.from_bases([basis1, basis2])
```

def getDegrees(self**) -> numpy.ndarray[numpy.integer]:**   • View Source

Returns the polynomial degree of each basis function in the isoparametric space.

Returns
  • **degrees** (np.ndarray[np.integer]): Array containing the polynomial degrees of the B-spline basis functions. The array has length `NPa` (dimension of isoparametric space), where each element represents the degree of the corresponding isoparametric dimension.

Notes
  • For a curve (1D), returns [degree_xi]
  • For a surface (2D), returns [degree_xi, degree_eta]
  • For a volume (3D), returns [degree_xi, degree_eta, degree_zeta]

- ...

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) -> list[numpy.ndarray[numpy.floating]]:

- View Source

Returns the knot vector of each basis function in the isoparametric space.

This method collects all knot vectors from each `BSplineBasis` instance stored in the `bases` array. The knot vectors define the isoparametric space partitioning and the regularity properties of the B-spline.

Returns

- **knots** (list[np.ndarray[np.floating]]): List containing the knot vectors of the B-spline basis functions. The list has length `NPa` (dimension of isoparametric space), where each element is a `numpy.ndarray` containing the knots for the corresponding isoparametric dimension.

Notes

- For a curve (1D), returns [`knots_xi`]
- For a surface (2D), returns [`knots_xi` , `knots_eta`]
- For a volume (3D), returns [`knots_xi` , `knots_eta` , `knots_zeta`]
- Each knot vector must be non-decreasing
- The multiplicity of interior knots determines the continuity at that point

Examples

```
>>> degrees = [2, 2]
>>> 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) -> int:

- View Source

Compute the total number of basis functions in the B-spline.

This method calculates the total number of basis functions by multiplying the number of basis functions in each isoparametric dimension ($n + 1$ for each dimension).

Returns

- **int**: Total number of basis functions in the B-spline. This is equal to the product of ($n + 1$) for each basis, where n is the last index of each basis function.

Notes

- For a curve (1D), returns ($n + 1$)
- For a surface (2D), returns ($n1 + 1$) × ($n2 + 1$)
- For a volume (3D), returns ($n1 + 1$) × ($n2 + 1$) × ($n3 + 1$)
- The number of basis functions equals the number of control points needed

Examples

```
>>> degrees = [2, 2]
>>> 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.getNbFunc()
16
```

def getSpans(self) -> list[tuple[float, float]]:   • View Source

Returns the span of each basis function in the isoparametric space.

This method collects the spans (intervals of definition) from each `BSplineBasis` instance stored in the `bases` array.

Returns

- **spans** (list[tuple[float, float]]): List containing the spans of the B-spline basis functions. The list has length `NPa` (dimension of isoparametric space), where each element is a tuple (`a`, `b`) containing the lower and upper bounds of the span for the corresponding isoparametric dimension.

Notes

- For a curve (1D), returns [(`xi_min`, `xi_max`)]
- For a surface (2D), returns [(`xi_min`, `xi_max`), (`eta_min`, `eta_max`)]
- For a volume (3D), returns [(`xi_min`, `xi_max`), (`eta_min`, `eta_max`), (`zeta_min`, `zeta_max`)]
- The span represents the interval where the B-spline is defined
- Each span is determined by the `p`-th and `(m - p)`-th knots, where `p` is the degree and `m` is the last index of the knot vector

Examples

```
>>> degrees = [2, 2]
>>> 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.getSpans()
[(0.0, 1.0), (0.0, 1.0)]
```

def linspace(
  self,
  n_eval_per_elem: Union[int, Iterable[int]] = 10
) -> tuple[numpy.ndarray[numpy.floating], ...]:   • View Source

Generate sets of evaluation points over the span of each basis in the isoparametric space.

This method creates evenly spaced points for each isoparametric dimension by calling `linspace` on each `BSplineBasis` instance stored in the `bases` array.

Parameters

- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. If an `int` is provided, the same number is used for all dimensions. If an `Iterable` is provided, each value corresponds to a different dimension. By default, 10.

Returns

- **XI** (tuple[np.ndarray[np.floating], ...]): Tuple containing arrays of evaluation points for each isoparametric dimension. The tuple has length `NPa` (dimension of isoparametric space).

Notes

- For a curve (1D), returns (`xi` points,)
- For a surface (2D), returns (`xi` points, `eta` points)
- For a volume (3D), returns (`xi` points, `eta` points, `zeta` points)
- The number of points returned for each dimension depends on the number of elements in that dimension times the value of `n_eval_per_elem`

Examples

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

...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> xi, eta = spline.linspace(n_eval_per_elem=2)
>>> xi
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>>> eta
array([0. , 0.5, 1. ])
```

def linspace_for_integration(
  self,
  n_eval_per_elem: Union[int, Iterable[int]] = 10,
  bounding_box: Optional[Iterable] = **None**
) -> tuple[tuple[numpy.ndarray[numpy.floating], ...], tuple[numpy.ndarray[numpy.floating], ...]]:   • View Source

Generate sets of evaluation points and their integration weights over each basis span.

This method creates evenly spaced points and their corresponding integration weights for each isoparametric dimension by calling `linspace_for_integration` on each `BSplineBasis` instance stored in the `bases` array.

Parameters

- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. If an `int` is provided, the same number is used for all dimensions. If an `Iterable` is provided, each value corresponds to a different dimension. By default, 10.
- **bounding_box** (Union[Iterable[tuple[float, float]], None], optional): Lower and upper bounds for each isoparametric dimension. If `None`, uses the span of each basis. Format: [(`xi_min`, `xi_max`), (`eta_min`, `eta_max`), ...]. By default, None.

Returns

- **XI** (tuple[np.ndarray[np.floating], ...]): Tuple containing arrays of evaluation points for each isoparametric dimension. The tuple has length `NPa` (dimension of isoparametric space).
- **dXI** (tuple[np.ndarray[np.floating], ...]): Tuple containing arrays of integration weights for each isoparametric dimension. The tuple has length `NPa` (dimension of isoparametric space).

Notes

- For a curve (1D), returns ((`xi` points), (`xi` weights))
- For a surface (2D), returns ((`xi` points, `eta` points), (`xi` weights, `eta` weights))
- For a volume (3D), returns ((`xi` points, `eta` points, `zeta` points), (`xi` weights, `eta` weights, `zeta` weights))
- The points are centered in their integration intervals
- The weights represent the size of the integration intervals

Examples

```
>>> degrees = [2, 2]
>>> 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, eta), (dxi, deta) = spline.linspace_for_integration(n_eval_per_elem=2
)
>>> xi   # xi points
array([0.125, 0.375, 0.625, 0.875])
>>> dxi  # xi weights
array([0.25, 0.25, 0.25, 0.25])
```

def gauss_legendre_for_integration(
  self,
  n_eval_per_elem: Union[int, Iterable[int], NoneType] = **None**,
  bounding_box: Optional[Iterable] = **None**
) -> tuple[tuple[numpy.ndarray[numpy.floating], ...], tuple[numpy.ndarray[numpy.floating], ...]]:   ● View Source

Generate sets of evaluation points and their Gauss-Legendre integration weights over each basis span.

This method creates Gauss-Legendre quadrature points and their corresponding integration weights for each isoparametric dimension by calling `gauss_legendre_for_integration` on each `BSplineBasis` instance stored in the `bases` array.

Parameters

- **n_eval_per_elem** (Union[int, Iterable[int], None], optional): Number of evaluation points per element for each isoparametric dimension. If an `int` is provided, the same number is used for all dimensions. If an `Iterable` is provided, each value corresponds to a different dimension. If `None`, uses `(p + 2)//2` points per element where `p` is the degree of each basis. This number of points ensures an exact integration of a `p`-th degree polynomial. By default, None.
- **bounding_box** (Union[Iterable[tuple[float, float]], None], optional): Lower and upper bounds for each isoparametric dimension. If `None`, uses the span of each basis. Format: [(`xi_min`, `xi_max`), (`eta_min`, `eta_max`), ...]. By default, None.

Returns

- **XI** (tuple[np.ndarray[np.floating], ...]): Tuple containing arrays of Gauss-Legendre points for each isoparametric dimension. The tuple has length `NPa` (dimension of isoparametric space).
- **dXI** (tuple[np.ndarray[np.floating], ...]): Tuple containing arrays of Gauss-Legendre weights for each isoparametric dimension. The tuple has length `NPa` (dimension of isoparametric space).

Notes

- For a curve (1D), returns ((`xi` points), (`xi` weights))
- For a surface (2D), returns ((`xi` points, `eta` points), (`xi` weights, `eta` weights))
- For a volume (3D), returns ((`xi` points, `eta` points, `zeta` points), (`xi` weights, `eta` weights, `zeta` weights))
- The points and weights follow the Gauss-Legendre quadrature rule
- When `n_eval_per_elem` is `None`, uses `(p + 2)//2` points per element for exact integration of polynomials up to degree `p`

Examples

```
>>> degrees = [2, 2]
>>> 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, eta), (dxi, deta) = spline.gauss_legendre_for_integration()

>>> xi   # xi points
array([0.10566243, 0.39433757, 0.60566243, 0.89433757])
>>> dxi  # xi weights
array([0.25, 0.25, 0.25, 0.25])
```

def normalize_knots(self):

Maps all knot vectors to the interval [0, 1] in each isoparametric dimension.

This method normalizes the knot vectors of each `BSplineBasis` instance stored in the `bases` array by applying an affine transformation that maps the span interval to [0, 1].

Notes

- The transformation preserves the relative spacing between knots
- The transformation preserves the multiplicity of knots
- The transformation is applied independently to each isoparametric dimension
- This operation modifies the knot vectors in place

Examples

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

...           np.array([0, 0, 0, 2, 4, 4, 4], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> spline.getKnots()
[array([-1., -1., -1.,  0.,  1.,  1.,  1.]),
 array([0., 0., 0., 2., 4., 4., 4.])]
>>> spline.normalize_knots()
>>> spline.getKnots()
[array([0., 0., 0., 0.5, 1., 1., 1.]),
 array([0., 0., 0., 0.5, 1., 1., 1.])]
```

def DN(
  self,
  XI: Union[numpy.ndarray[numpy.floating], tuple[numpy.ndarray[numpy.floating], ...]],
  k: Union[int, Iterable[int]] = 0
) -> Union[scipy.sparse._matrix.spmatrix, numpy.ndarray[scipy.sparse._matrix.spmatrix]]:

Compute the `k` -th derivative of the B-spline basis at given points in the isoparametric space.

This method evaluates the basis functions or their derivatives at specified points, returning a matrix that can be used to evaluate the B-spline through a dot product with the control points.

Parameters

- **XI** (Union[np.ndarray[np.floating], tuple[np.ndarray[np.floating], ...]]): Points in the isoparametric space where to evaluate the basis functions. Two input formats are accepted:
  1. `numpy.ndarray` : Array of coordinates with shape (`NPa` , n_points). Each column represents one evaluation point [`xi` , `eta` , ...]. The resulting matrices will have shape (n_points, number of functions).
  2. `tuple` : Contains `NPa` arrays of coordinates (`xi` , `eta` , ...). The resulting matrices will have (n_xi × n_eta × ...) rows.
- **k** (Union[int, Iterable[int]], optional): Derivative orders to compute. Two formats are accepted:
  1. `int` : Same derivative order along all axes. Common values:
  - `k=0` : Evaluate basis functions (default)
  - `k=1` : Compute first derivatives (gradient)
  - `k=2` : Compute second derivatives (hessian)
  2. `list[int]` : Different derivative orders for each axis. Example: `[1, 0]` computes first derivative w.r.t `xi` , no derivative w.r.t `eta` . By default, 0.

Returns

- **DN** (Union[sps.spmatrix, np.ndarray[sps.spmatrix]]): Sparse matrix or array of sparse matrices containing the basis evaluations:
  - If `k` is a `list` or is 0: Returns a single sparse matrix containing the mixed derivative specified by the list.

- If `k` is an `int` > 0: Returns an array of sparse matrices with shape [`NPa`]* `k` . For example, if `k=1` , returns `NPa` matrices containing derivatives along each axis.

Notes

- For evaluating the B-spline with control points in `NPh` -D space: `values = DN @ ctrl_pts.reshape((NPh, -1)).T`
- When using tuple input format for `XI` , points are evaluated at all combinations of coordinates
- When using array input format for `XI` , each column represents one evaluation point
- The gradient (`k=1`) returns `NPa` matrices for derivatives along each axis
- Mixed derivatives can be computed using a list of derivative orders

Examples

Create a 2D quadratic B-spline:

```
>>> degrees = [2, 2]
>>> 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)
```

Evaluate basis functions at specific points using array input:

```
>>> XI = np.array([[0, 0.5, 1],      # xi coordinates
...                [0, 0.5, 1]])     # eta coordinates
>>> N = spline.DN(XI, k=0)
>>> N.A  # Convert sparse matrix to dense for display
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.25, 0.25, 0., 0., 0.25, 0.25, 0., 0., 0., 0., 0.],

       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

Compute first derivatives using tuple input:

```
>>> xi = np.array([0, 0.5])
>>> eta = np.array([0, 1])
>>> dN = spline.DN((xi, eta), k=1)  # Returns [NPa] matrices
>>> len(dN)  # Number of derivative matrices
2
>>> dN[0].A  # Derivative w.r.t xi
array([[-4., 0., 0., 0., 4., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

       [ 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.],

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

>>> dN[1].A  # Derivative w.r.t eta
array([[-4., 4., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

       [ 0., 0.,-4., 4., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],

       [ 0., 0., 0., 0.,-2., 2., 0., 0.,-2., 2., 0., 0., 0., 0., 0., 0.],

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

Compute mixed derivatives:

```
>>> d2N = spline.DN((xi, eta), k=[1, 1])  # Second derivative: d²/dxi·deta
>>> d2N.A
array([[16.,-16.,  0.,  0.,-16.,16.,  0., 0., 0., 0., 0., 0., 0., 0., 0.],

       [ 0.,  0., 16.,-16.,  0., 0.,-16.,16., 0., 0., 0., 0., 0., 0., 0.],

       [ 0.,  0.,  0.,  0.,  8.,-8.,  0., 0.,-8., 8., 0., 0., 0., 0., 0.],

       [ 0.,  0.,  0.,  0.,  0., 0.,  8.,-8., 0.,-0.,-8., 8., 0., 0., 0.]])
```

def knotInsertion(
 self,
 ctrl_pts: numpy.ndarray[numpy.floating],
 knots_to_add: Iterable[Union[numpy.ndarray[numpy.float64], int]]
) -> numpy.ndarray[numpy.floating]:

- View Source

Add knots to the B-spline while preserving its geometry.

This method performs knot insertion by adding new knots to each isoparametric dimension and computing the new control points to maintain the exact same geometry. The method modifies the `BSpline` object by updating its basis functions with the new knots.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Control points defining the B-spline geometry. Shape: (`NPh` , n1, n2, ...) where:
    - `NPh` is the dimension of the physical space
    - ni is the number of control points in the i-th isoparametric dimension
- **knots_to_add** (Iterable[Union[np.ndarray[np.floating], int]]): Refinement specification for each isoparametric dimension. For each dimension, two formats are accepted:
    1. `numpy.ndarray` : Array of knots to insert. These knots must lie within the span of the existing knot vector.
    2. `int` : Number of equally spaced knots to insert in each element.

Returns

- **new_ctrl_pts** (np.ndarray[np.floating]): New control points after knot insertion. Shape: (`NPh` , m1, m2, ...) where mi ≥ ni is the new number of control points in the i-th isoparametric dimension.

Notes

- Knot insertion preserves the geometry and parameterization of the B-spline
- The number of new control points depends on the number and multiplicity of inserted knots
- When using integer input, knots are inserted with uniform spacing in each element
- The method modifies the basis functions but maintains `C^{p-m}` continuity, where `p` is the degree and `m` is the multiplicity of the inserted knot

Examples

Create a 2D quadratic B-spline and insert knots:

```
>>> degrees = [2, 2]
>>> 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)
>>> ctrl_pts = np.random.rand(3, 4, 4)  # 3D control points
```

Insert specific knots in first dimension only:

```
>>> knots_to_add = [np.array([0.25, 0.75], dtype='float'),
...                   np.array([], dtype='float')]
>>> new_ctrl_pts = spline.knotInsertion(ctrl_pts, knots_to_add
)
>>> new_ctrl_pts.shape
(3, 6, 4)  # Two new control points added in first dimension
>>> spline.getKnots()[0]  # The knot vector is modified
array([0.  , 0.  , 0.  , 0.25, 0.5 , 0.75, 1.  , 1.  , 1.  ])
```

Insert two knots per element in both dimensions:

```
>>> new_ctrl_pts = spline.knotInsertion(new_ctrl_pts, [1, 1])
>>> new_ctrl_pts.shape
(3, 10, 6)  # Uniform refinement in both dimensions
>>> spline.getKnots()[0]  # The knot vectors are further modified
array([0.   , 0.   , 0.   , 0.125, 0.25 , 0.375, 0.5  , 0.625, 0.75 ,

       0.875, 1.   , 1.   , 1.   ])
```

def orderElevation(
 self,
 ctrl_pts: numpy.ndarray[numpy.floating],
 t: Iterable[int]
) -> numpy.ndarray[numpy.floating]:   • View Source

Elevate the polynomial degree of the B-spline while preserving its geometry.

This method performs order elevation by increasing the polynomial degree of each isoparametric dimension and computing the new control points to maintain the exact same geometry. The method modifies the `BSpline` object by updating its basis functions with the new degrees.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Control points defining the B-spline geometry. Shape: (`NPh` , n1, n2, ...) where:
  - `NPh` is the dimension of the physical space
  - ni is the number of control points in the i-th isoparametric dimension
- **t** (Iterable[int]): Degree elevation for each isoparametric dimension. For each dimension i, the new degree will be `p_i + t_i` where `p_i` is the current degree.

Returns

- **new_ctrl_pts** (np.ndarray[np.floating]): New control points after order elevation. Shape: (`NPh` , m1, m2, ...) where mi ≥ ni is the new number of control points in the i-th isoparametric dimension.

Notes

- Order elevation preserves the geometry and parameterization of the B-spline
- The number of new control points depends on the current degree and number of elements
- The method modifies the `BSpline` object by updating its basis functions
- This operation is more computationally expensive than knot insertion

Examples

Create a 2D quadratic B-spline and elevate its order:

```
>>> degrees = [2, 2]
>>> 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)
>>> ctrl_pts = np.random.rand(3, 4, 4)  # 3D control points
```

Elevate order by 1 in first dimension only:

```
>>> t = [1, 0]  # Increase degree by 1 in first dimension
>>> new_ctrl_pts = spline.orderElevation(ctrl_pts, t)
>>> new_ctrl_pts.shape
(3, 6, 4)  # Two new control points added in first dimension (one per element)

>>> spline.getDegrees()  # The degrees are modified
array([3, 2])
```

def greville_abscissa(
  self,
  return_weights: bool = **False**
) -> Union[list[numpy.ndarray[numpy.floating]], tuple[list[numpy.ndarray[numpy.floating]], list[numpy.ndarray[

- View Source

Compute the Greville abscissa and optionally their weights for each isoparametric dimension.

The Greville abscissa can be interpreted as the "position" of the control points in the isoparametric space. They are often used as interpolation points for B-splines.

Parameters

- **return_weights** (bool, optional): If `True`, also returns the weights (span lengths) of each basis function. By default, False.

Returns

- **greville** (list[np.ndarray[np.floating]]): List containing the Greville abscissa for each isoparametric dimension. The list has length `NPa`, where each element is an array of size `n + 1`, `n` being the last index of the basis functions in that dimension.
- **weights** (list[np.ndarray[np.floating]], optional): Only returned if `return_weights` is `True`. List containing the weights for each isoparametric dimension. The list has length `NPa`, where each element is an array containing the span length of each basis function.

Notes

- For a curve (1D), returns [`xi` abscissa]
- For a surface (2D), returns [`xi` abscissa, `eta` abscissa]
- For a volume (3D), returns [`xi` abscissa, `eta` abscissa, `zeta` abscissa]
- The Greville abscissa are computed as averages of `p` consecutive knots
- The weights represent the size of the support of each basis function
- The number of abscissa in each dimension equals the number of control points

Examples

Compute Greville abscissa for a 2D B-spline:

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

...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> greville = spline.greville_abscissa()
>>> greville[0]  # xi coordinates
array([0.  , 0.25, 0.75, 1.  ])
>>> greville[1]  # eta coordinates
array([0. , 0.5, 1. ])
```

Compute both abscissa and weights:

```
>>> greville, weights = spline.greville_abscissa(return_weights=True
)
>>> weights[0]  # weights for xi direction
array([0.5, 1. , 1. , 0.5])
```

def make_control_poly_meshes(
  self,
  ctrl_pts: numpy.ndarray[numpy.floating],
  n_eval_per_elem: Union[int, Iterable[int]] = 10,
  n_step: int = 1,
  fields: dict = {},
  XI: Optional[tuple[numpy.ndarray[numpy.floating], ...]] = **None**,
  paraview_sizes: dict = {}
) -> list[meshio._mesh.Mesh]:
- View Source

Create meshes containing all the data needed to plot the control polygon of the B-spline.

This method generates a list of `io.Mesh` objects representing the control mesh (polygonal connectivity) of the B-spline, suitable for visualization (e.g. in Paraview). It supports time-dependent fields and arbitrary dimension.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Array of control points of the B-spline, with shape (`NPh` , number of elements for dim 1, ..., number of elements for dim `NPa`), where `NPh` is the physical space dimension and `NPa` is the dimension of the isoparametric space.
- **n_step** (int, optional): Number of time steps to plot. By default, 1.
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. By default, 10.
  - If an `int` is provided, the same number is used for all dimensions.
  - If an `Iterable` is provided, each value corresponds to a different dimension.
- **n_step** (int, optional): Number of time steps to plot. By default, 1.
- **fields** (dict, optional): Dictionary of fields to plot at each time step. Keys are field names. Values can be:
  - a `function` taking (`BSpline` spline, `tuple` of `np.ndarray[np.floating]` XI) and returning a `np.ndarray[np.floating]` of shape (`n_step` , number of combinations of XI, field size),
  - a `np.ndarray[np.floating]` defined **on the control points**, of shape (`n_step` , field size, * `ctrl_pts.shape[1:]`), which is then interpolated using the B-spline basis functions,
  - a `np.ndarray[np.floating]` defined **on the evaluation grid**, of shape (`n_step` , field size, *grid shape), where `grid shape` matches the discretization provided by XI or `n_eval_per_elem` . In this case, the field is interpolated in physical space using `scipy.interpolate.griddata` .
- **XI** (tuple[np.ndarray[np.floating], ...], optional): Parametric coordinates at which to evaluate the B-spline and fields. If not `None` , overrides the `n_eval_per_elem` parameter. If `None` , a regular grid is generated according to `n_eval_per_elem` .
- **paraview_sizes** (dict, optionnal): The fields present in this `dict` are overridden by `np.NaN` s. The keys must be the fields names and the values must be the fields sizes for paraview. By default, {}.

Returns

- **list[io.Mesh]**: List of `io.Mesh` objects, one for each time step, containing the control mesh geometry and associated fields.

Notes

- The control mesh is constructed by connecting control points along each isoparametric direction.
- Fields can be provided either as functions evaluated at the Greville abscissae, or as arrays defined on the control points or on a regular parametric grid (in which case they are interpolated at the Greville abscissae).
- The first axis of the field array or function output corresponds to the time step, even if there is only one.
- The method is compatible with B-splines of arbitrary dimension.

Examples

```
>>> degrees = [2, 2]
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),
...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> ctrl_pts = np.random.rand(3, 4, 3)  # 3D control points for a 2D surface

>>> meshes = spline.make_control_poly_meshes(ctrl_pts)
>>> mesh = meshes[0]
```

def make_elem_separator_meshes(
 self,
 ctrl_pts: numpy.ndarray[numpy.floating],
 n_eval_per_elem: Union[int, Iterable[int]] = 10,
 n_step: int = 1,
 fields: dict = {},
 XI: Optional[tuple[numpy.ndarray[numpy.floating], ...]] = **None**,
 paraview_sizes: dict = {}
) -> list[meshio._mesh.Mesh]:   • View Source

Create meshes representing the boundaries of every element in the B-spline for visualization.

This method generates a list of `io.Mesh` objects containing the geometry and optional fields needed to plot the limits (borders) of all elements from the isoparametric space of the B-spline. Supports time-dependent fields and arbitrary dimension.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Array of control points of the B-spline, with shape (`NPh`, number of elements for dim 1, ..., number of elements for dim `NPa`), where `NPh` is the physical space dimension and `NPa` is the dimension of the isoparametric space.
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. By default, 10.
  - If an `int` is provided, the same number is used for all dimensions.
  - If an `Iterable` is provided, each value corresponds to a different dimension.
- **n_step** (int, optional): Number of time steps to plot. By default, 1.
- **fields** (dict, optional): Dictionary of fields to plot at each time step. Keys are field names. Values can be:
  - a `function` taking (`BSpline` spline, `tuple` of `np.ndarray[np.floating]` XI) and returning a `np.ndarray[np.floating]` of shape (`n_step`, number of combinations of XI, field size),
  - a `np.ndarray[np.floating]` defined **on the control points**, of shape (`n_step`, field size, * `ctrl_pts.shape[1:]`), which is then interpolated using the B-spline basis functions,
  - a `np.ndarray[np.floating]` defined **on the evaluation grid**, of shape (`n_step`, field size, *grid shape), where `grid shape` matches the discretization provided by XI or `n_eval_per_elem`. In this case, the field is interpolated in physical space using `scipy.interpolate.griddata`.
- **XI** (tuple[np.ndarray[np.floating], ...], optional): Parametric coordinates at which to evaluate the B-spline and fields. If not `None`, overrides the `n_eval_per_elem` parameter. If `None`, a regular grid is generated according to `n_eval_per_elem`.
- **paraview_sizes** (dict, optionnal): The fields present in this `dict` are overridden by `np.NaN`s. The keys must be the fields names and the values must be the fields sizes for paraview. By default, {}.

Returns

- **list[io.Mesh]**: List of `io.Mesh` objects, one for each time step, containing the element boundary geometry and associated fields.

Notes

- The element boundary mesh is constructed by connecting points along the unique knot values in each isoparametric direction, outlining the limits of each element.
- Fields can be provided either as callable functions, as arrays defined on the control points, or as arrays

already defined on a regular evaluation grid.

- When fields are defined on a grid, they are interpolated in the physical space using `scipy.interpolate.griddata` with linear interpolation.
- The first axis of the field array or function output corresponds to the time step, even if there is only one.
- The method supports B-splines of arbitrary dimension.

Examples

```
>>> degrees = [2, 2]
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),
...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> ctrl_pts = np.random.rand(3, 4, 3)  # 3D control points for a 2D surface

>>> meshes = spline.make_elem_separator_meshes(ctrl_pts)
>>> mesh = meshes[0]
```

def make_elements_interior_meshes(
  self,
  ctrl_pts: numpy.ndarray[numpy.floating],
  n_eval_per_elem: Union[int, Iterable[int]] = 10,
  n_step: int = 1,
  fields: dict = {},
  XI: Optional[tuple[numpy.ndarray[numpy.floating], ...]] = **None**
) -> list[meshio._mesh.Mesh]:

- View Source

Create meshes representing the interior of each element in the B-spline.

This method generates a list of `io.Mesh` objects containing the geometry and optional fields for the interior of all elements, suitable for visualization (e.g., in ParaView). Supports time-dependent fields and arbitrary dimension.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Array of control points of the B-spline, with shape (`NPh`, number of points for dim 1, ..., number of points for dim `NPa`), where `NPh` is the physical space dimension and `NPa` is the dimension of the isoparametric space.
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. By default, 10.
  - If an `int` is provided, the same number is used for all dimensions.
  - If an `Iterable` is provided, each value corresponds to a different dimension.
- **n_step** (int, optional): Number of time steps to plot. By default, 1.
- **fields** (dict, optional): Dictionary of fields to plot at each time step. Keys are field names. Values can be:
  - a `function` taking (`BSpline` spline, `tuple` of `np.ndarray[np.floating]` XI) and returning a `np.ndarray[np.floating]` of shape (`n_step`, number of combinations of XI, field size),
  - a `np.ndarray[np.floating]` defined **on the control points**, of shape (`n_step`, field size, * `ctrl_pts.shape[1:]`), in which case it is interpolated using the B-spline basis functions,
  - a `np.ndarray[np.floating]` defined **directly on the evaluation grid**, of shape (`n_step`, field size, *grid shape), where `grid shape` is the shape of the discretization XI (i.e., number of points along each parametric axis). By default, `{}` (no fields).
- **XI** (tuple[np.ndarray[np.floating], ...], optional): Parametric coordinates at which to evaluate the B-spline and fields. If not `None`, overrides the `n_eval_per_elem` parameter. If `None`, a regular grid is generated according to `n_eval_per_elem`.

Returns

- **list[io.Mesh]**: List of `io.Mesh` objects, one for each time step, containing the element interior geometry and associated fields.

Notes

- The interior mesh is constructed by evaluating the B-spline at a regular grid of points in the isoparametric space, with connectivity corresponding to lines (1D), quads (2D), or hexahedra (3D).
- Fields can be provided either as arrays (on control points or on the discretization grid) or as functions.
- Arrays given on control points are automatically interpolated using the B-spline basis functions.
- Arrays already given on the evaluation grid are used directly without interpolation.
- The first axis of the field array or function output must correspond to the time step, even if there is only one.
- The method is compatible with B-splines of arbitrary dimension.

Examples

```
>>> degrees = [2, 2]
>>> knots = [np.array([0, 0, 0, 0.5, 1, 1, 1], dtype='float'),
...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> ctrl_pts = np.random.rand(3, 4, 3)  # 3D control points for a 2D surface

>>> # Field given on control points (needs interpolation)
>>> field_on_ctrl_pts = np.random.rand(1, 1, 4, 3)
>>> # Field given directly on the evaluation grid (no interpolation)
>>> field_on_grid = np.random.rand(1, 1, 10, 10)
>>> meshes = spline.make_elements_interior_meshes(
...     ctrl_pts,
...     fields={'temperature': field_on_ctrl_pts, 'pressure': field_on_grid
}
...     XI= # TODO
... )
>>> mesh = meshes[0]
```

def saveParaview(
 self,
 ctrl_pts: numpy.ndarray[numpy.floating],
 path: str,
 name: str,
 n_step: int = 1,
 n_eval_per_elem: Union[int, Iterable[int]] = 10,
 fields: Optional[dict] = **None**,
 XI: Optional[tuple[numpy.ndarray[numpy.floating], ...]] = **None**,
 groups: Optional[dict[str, dict[str, Union[str, int]]]] = **None**,
 make_pvd: bool = **True**,
 verbose: bool = **True**,
 fields_on_interior_only: Union[bool, Literal['auto'], list[str]] = 'auto'
) -> dict[str, dict[str, typing.Union[str, int]]]:

- View Source

Save B-spline visualization data as Paraview files.

This method creates three types of visualization files:

- Interior mesh showing the B-spline surface/volume
- Element borders showing the mesh structure
- Control points mesh showing the control structure

All files are saved in VTU format with an optional PVD file to group them.

Parameters
- **ctrl_pts** (np.ndarray[np.floating]): Control points defining the B-spline geometry. Shape: (NPh , n1, n2, ...) where:
  - NPh is the dimension of the physical space
  - ni is the number of control points in the i-th isoparametric dimension
- **path** (str): Directory path where the PV files will be saved
- **name** (str): Base name for the output files
- **n_step** (int, optional): Number of time steps to save. By default, 1.

- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for each isoparametric dimension. By default, 10.
  - If an `int` is provided, the same number is used for all dimensions.
  - If an `Iterable` is provided, each value corresponds to a different dimension.
- **fields** (Union[dict, None], optional): Fields to visualize at each time step. Dictionary format: { "field_name": `field_value` } where `field_value` can be either:

  1. A numpy array with shape (`n_step`, `field_size`, `*ctrl_pts.shape[1:]`) where:
  - `n_step` : Number of time steps
  - `field_size` : Size of the field at each point (1 for scalar, 3 for vector)
  - `*ctrl_pts.shape[1:]` : Same shape as control points (excluding `NPh`)
  2. A numpy array with shape (`n_step`, `field_size`, `*grid_shape`) where:
  - `n_step` : Number of time steps
  - `field_size` : Size of the field at each point (1 for scalar, 3 for vector)
  - `*grid_shape` : Shape of the evaluation grid (number of points along each isoparametric axis)
  3. A function that computes field values (`np.ndarray[np.floating]`) at given points from the `BSpline` instance and `XI`, the tuple of arrays containing evaluation points for each dimension (`tuple[np.ndarray[np.floating], ...]`). The result should be an array of shape (`n_step`, `n_points`, `field_size`) where:
  - `n_step` : Number of time steps
  - `n_points` : Number of evaluation points (n_xi × n_eta × ...)
  - `field_size` : Size of the field at each point (1 for scalar, 3 for vector)

  By default, None.

- **XI** (tuple[np.ndarray[np.floating], ...], optional): Parametric coordinates at which to evaluate the B-spline and fields. If not `None`, overrides the `n_eval_per_elem` parameter. If `None`, a regular grid is generated according to `n_eval_per_elem`.
- **groups** (Union[dict[str, dict[str, Union[str, int]]], None], optional): Nested dictionary specifying file groups for PVD organization. Format: { "group_name": { "ext": str, # File extension (e.g., "vtu") "npart": int, # Number of parts in the group "nstep": int # Number of timesteps } } The method automatically creates/updates three groups:

  - "interior": For the B-spline surface/volume mesh
  - "elements_borders": For the element boundary mesh
  - "control_points": For the control point mesh

  If provided, existing groups are updated; if None, these groups are created. By default, None.

- **make_pvd** (bool, optional): Whether to create a PVD file grouping all VTU files. By default, True.
- **verbose** (bool, optional): Whether to print progress information. By default, True.
- **fields_on_interior_only** (Union[bool, Literal['auto'], list[str]], optionnal): Whether to include fields only on the interior mesh (`True`), on all meshes (`False`), or on specified field names. If set to `'auto'` , fields named `'u'` , `'U'` , `'displacement'` or `'displ'` are included on all meshes while others are only included on the interior mesh. By default, 'auto'.

Returns

- **groups** (dict[str, dict[str, Union[str, int]]]): Updated groups dictionary with information about saved files.

Notes

- Creates three types of VTU files for each time step:
  - {name}_interior_{part}_{step}.vtu
  - {name}_elements_borders_{part}_{step}.vtu
  - {name}_control_points_{part}_{step}.vtu
- If `make_pvd=True` , creates a PVD file named {name}.pvd
- Fields can be visualized as scalars or vectors in Paraview
- The method supports time-dependent visualization through `n_step`

Examples

Save a 2D B-spline visualization:

```
>>> degrees = [2, 2]
>>> 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)
>>> ctrl_pts = np.random.rand(3, 4, 4)  # 3D control points
>>> spline.saveParaview(ctrl_pts, "./output", "bspline")
```

Save with a custom field:

```
>>> def displacement(spline, XI):
...     # Compute displacement field
...     return np.random.rand(1, np.prod([x.size for x in XI]), 3)
>>> fields = {"displacement": displacement}
>>> spline.saveParaview(ctrl_pts, "./output", "bspline", fields=fields
)
```

def getGeomdl(self, ctrl_pts):

- View Source

def plotMPL(
 self,
 ctrl_pts: numpy.ndarray[numpy.floating],
 n_eval_per_elem: Union[int, Iterable[int]] = 10,
 ax: Optional[matplotlib.axes._axes.Axes] = **None**,
 ctrl_color: str = '#1b9e77',
 interior_color: str = '#7570b3',
 elem_color: str = '#666666',
 border_color: str = '#d95f02',
 language: Union[Literal['english'], Literal['français']] = 'english'
 ):

- View Source

Plot the B-spline using Matplotlib.

Creates a visualization of the B-spline geometry showing the control mesh, B-spline surface/curve, element borders, and patch borders. Supports plotting 1D curves and 2D surfaces in 2D space, and 2D surfaces and 3D volumes in 3D space.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Control points defining the B-spline geometry. Shape: (NPh, n1, n2, ...) where:
  - NPh is the dimension of the physical space (2 or 3)
  - ni is the number of control points in the i-th isoparametric dimension
- **n_eval_per_elem** (Union[int, Iterable[int]], optional): Number of evaluation points per element for visualizing the B-spline. Can be specified as:
  - Single integer: Same number for all dimensions
  - Iterable of integers: Different numbers for each dimension By default, 10.
- **ax** (Union[mpl.axes.Axes, None], optional): Matplotlib axes for plotting. If None, creates a new figure and axes. For 3D visualizations, must be a 3D axes if provided (created with `projection='3d'`). Default is None (creates new axes).
- **ctrl_color** (str, optional): Color for the control mesh visualization:
  - Applied to control points (markers)
  - Applied to control mesh lines Default is '#1b9e77' (green).
- **interior_color** (str, optional): Color for the B-spline geometry:
  - For curves: Line color
  - For surfaces: Face color (with transparency)

- For volumes: Face color of boundary surfaces (with transparency) Default is '#7570b3' (purple).
- **elem_color** (str, optional): Color for element boundary visualization:
  - Shows internal mesh structure
  - Helps visualize knot locations Default is '#666666' (gray).
- **border_color** (str, optional): Color for patch boundary visualization:
  - Outlines the entire B-spline patch
  - Helps distinguish patch edges Default is '#d95f02' (orange).
- **language** (str, optional): Language for the plot labels. Can be 'english' or 'français'. Default is 'english'.

Notes

Visualization components:

- Control mesh: Shows control points and their connections
- B-spline: Shows the actual curve/surface/volume
- Element borders: Shows the boundaries between elements
- Patch borders: Shows the outer boundaries of the B-spline

Supported configurations:

- 1D B-spline in 2D space (curve)
- 2D B-spline in 2D space (surface)
- 2D B-spline in 3D space (surface)
- 3D B-spline in 3D space (volume)

For 3D visualization:

- Surfaces are shown with transparency
- Volume visualization shows the faces with transparency
- View angle is automatically set for surfaces based on surface normal

Examples

Plot a 2D curve in 2D space:

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

>>> spline = BSpline(degrees, knots)
>>> ctrl_pts = np.random.rand(2, 3)  # 2D control points
>>> spline.plotMPL(ctrl_pts)
```

Plot a 2D surface in 3D space:

```
>>> degrees = [2, 2]
>>> knots = [np.array([0, 0, 0, 1, 1, 1], dtype='float'),
...          np.array([0, 0, 0, 1, 1, 1], dtype='float')]
>>> spline = BSpline(degrees, knots)
>>> ctrl_pts = np.random.rand(3, 3, 3)  # 3D control points

>>> spline.plotMPL(ctrl_pts)
```

Plot on existing axes with custom colors:

```
>>> fig = plt.figure()
>>> ax = fig.add_subplot(projection='3d')
>>> spline.plotMPL(ctrl_pts, ax=ax, ctrl_color='red', interior_color='blue'
)
```

bsplyne.save_utils

def writePVD(fileName: str, groups: dict[str, dict]):

Writes a Paraview Data (PVD) file that references multiple VTK files.

Creates an XML-based PVD file that collects and organizes multiple VTK files into groups, allowing visualization of multi-part, time-series data in Paraview.

Parameters

- **fileName** (str): Base name for the mesh files (without numbers and extension)
- **groups** (dict[str, dict]): Nested dictionary specifying file groups with format: {"group_name": {"ext": "file_extension", "npart": num_parts, "nstep": num_timesteps}}

Notes

VTK files must follow naming pattern: {fileName}_{group}_{part}_{timestep}.{ext} Example: for fileName="mesh", group="fluid", part=1, timestep=5, ext="vtu": mesh_fluid_1_5.vtu

Returns

- **None**

def merge_meshes(meshes: Iterable[meshio._mesh.Mesh]) -> meshio._mesh.Mesh:

Merges multiple meshio.Mesh objects into a single mesh.

Parameters

- **meshes** (Iterable[io.Mesh]): An iterable of meshio.Mesh objects to merge.

Returns

- **io.Mesh**: A single meshio.Mesh object containing the merged meshes with combined vertices, cells and point data.

def merge_saves(
 path: str,
 name: str,
 nb_patchs: int,
 nb_steps: int,
 group_names: list[str]
) -> **None**:

Merge multiple mesh files and save the merged results.

This function reads multiple mesh files for each group and time step, merges them into a single mesh, and writes the merged mesh to a new file. It also generates a PVD file to describe the collection of merged meshes.

Parameters

- **path** (str): The directory path where the mesh files are located.
- **name** (str): The base name of the mesh files.
- **nb_patchs** (int): The number of patches to merge for each group and time step.
- **nb_steps** (int): The number of time steps for which meshes are available.
- **group_names** (list[str]): A list of group names to process.

Returns

- **None**