volVIC

volVIC

VOLume-based Virtual Image Correlation (volVIC) library for Python Python library for isogeometric multipatch surface fitting of volumetric images, based on the method described in Bichet et al., 2025.

📌 Overview

`volVIC` provides tools to extract **analysis-ready multipatch B-spline surface models** from volumetric scan data (CT/µCT), handling design deviations and manufacturing defects. Key features:

- Multi-patch B-spline surface representation.
- Deformation to match as-manufactured scans.
- Defect quantification and geometric analysis.

🚀 Installation

```
git clone https://github.com/Dorian210/volVIC.git
cd volVIC
pip install .
```

📜 Citation

```
D. Bichet, J.-C. Passieux, J.-N. Périé, R. Bouclier,
“Isogeometric multipatch surface fitting in tomographic images: application to lattice structures”
,
Computer Methods in Applied Mechanics and Engineering, 436:117729, 2025.
```

- View Source

volVIC.virtual_image

- View Source

```
def g_slide(
    xi: numpy.ndarray[numpy.floating],
    eta: numpy.ndarray[numpy.floating],
    gamma: numpy.ndarray[numpy.floating],
    rho: float,
    bg: float = 0.0,
    fg: float = 1.0
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]:
```

- View Source

Calculate the virtual image function and its derivative with respect to `rho`.

Parameters

- **xi** (np.ndarray[np.floating]): Array representing the xi coordinate.
- **eta** (np.ndarray[np.floating]): Array representing the eta coordinate.
- **gamma** (np.ndarray[np.floating]): Array representing the gamma coordinate.
- **rho** (float): Half width of transition from `bg` to `fg` graylevel.
- **bg** (float, optional): Background graylevel value, default is 0.
- **fg** (float, optional): Foreground graylevel value, default is 1.

Returns

- **g** (np.ndarray[np.floating]): Virtual image function evaluation.
- **g_prime** (np.ndarray[np.floating]): Derivative of the virtual image function with respect to `rho`.

volVIC.membrane_stiffness

- View Source

def make_coordinates_systems(
ctrl_pts: numpy.ndarray[numpy.floating],
dN_dxi: scipy.sparse._matrix.spmatrix,
dN_deta: scipy.sparse._matrix.spmatrix
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], nump

- View Source

Compute the covariant basis vectors and transformation components between covariant and contravariant bases.

Given `ctrl_pts` (control points in physical space) and the derivatives of the basis functions with respect to the isoparametric coordinates (`dN_dxi` , `dN_deta`), this function returns the covariant basis vectors (`A1` , `A2`) and the transformation components (`A11` , `A22` , `A12`) for each evaluation point.

Parameters

- **ctrl_pts** (np.ndarray[np.floating]): Array of control points in physical space, shaped as (`3` , `n_nodes_xi` , `n_nodes_eta`).
- **dN_dxi** (sps.spmatrix): Sparse matrix of derivatives of basis functions with respect to the first isoparametric coordinate (`xi`). Shape: (`n_gauss` , `n_nodes`).
- **dN_deta** (sps.spmatrix): Sparse matrix of derivatives of basis functions with respect to the second isoparametric coordinate (`eta`). Shape: (`n_gauss` , `n_nodes`).

Returns

- **tuple[np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating]]**: Tuple containing:
  - `A1` (`np.ndarray[np.floating]`): Covariant basis vector 1 at each evaluation point.
  - `A2` (`np.ndarray[np.floating]`): Covariant basis vector 2 at each evaluation point.
  - `A11` (`np.ndarray[np.floating]`): First component of the covariant-to-contravariant transformation.
  - `A22` (`np.ndarray[np.floating]`): Second component of the covariant-to-contravariant transformation.
  - `A12` (`np.ndarray[np.floating]`): Third component of the covariant-to-contravariant transformation.

Notes

- The transformation components are computed pointwise for each evaluation point in the isoparametric space.
- The returned tuple has length 5, with each entry corresponding to an array of values at all evaluation points.

def make_H(
A11: numpy.ndarray[numpy.floating],
A22: numpy.ndarray[numpy.floating],
A12: numpy.ndarray[numpy.floating]
) -> scipy.sparse._matrix.spmatrix:

- View Source

Assemble Hooke's tensor in contravariant basis Voigt notation for a membrane material with Poisson's ratio set to 0 and Young's modulus set to 1, using B-spline basis components.

Parameters

- **A11** (np.ndarray[np.floating]): Covariant-to-contravariant basis component array for the (1,1) direction in isoparametric space.
- **A22** (np.ndarray[np.floating]): Covariant-to-contravariant basis component array for the (2,2) direction in isoparametric space.
- **A12** (np.ndarray[np.floating]): Covariant-to-contravariant basis component array for the (1,2) direction in isoparametric space.

Returns

- **sps.spmatrix**: Hooke's tensor as a sparse matrix in contravariant basis Voigt notation, with shape (`3 * n_gauss`, `3 * n_gauss`), where `n_gauss` is the number of isoparametric points.

Notes

- The tensor is block-assembled using `scipy.sparse.bmat` and diagonal matrices from the input arrays.
- The material law is restricted to Poisson's ratio = 0 and Young's modulus = 1.
- The resulting tensor is suitable for use in isogeometric Kirchhoff-Love membrane formulations.

def make_Bm(
dN_dxi: scipy.sparse._matrix.spmatrix,
dN_deta: scipy.sparse._matrix.spmatrix,
A1: numpy.ndarray[numpy.floating],
A2: numpy.ndarray[numpy.floating]
) -> scipy.sparse._matrix.spmatrix:

- View Source

Assemble the Jacobian matrix `Bm` for membrane deformation in Voight notation in isogeometric analysis.

This function constructs the sparse matrix `Bm` by combining the derivatives of the shape functions in the isoparametric (`xi`, `eta`) directions with the corresponding covariant tangent vectors `A1` and `A2`. The resulting matrix maps nodal displacements to membrane strains in Voight notation, and is used in isogeometric membrane finite element formulations.

Parameters

- **dN_dxi** (sps.spmatrix): Sparse matrix of derivatives of the shape functions with respect to the `xi` coordinate in isoparametric space. Shape: (`n_gauss`, `n_nodes`).
- **dN_deta** (sps.spmatrix): Sparse matrix of derivatives of the shape functions with respect to the `eta` coordinate in isoparametric space. Shape: (`n_gauss`, `n_nodes`).
- **A1** (np.ndarray[np.floating]): Covariant tangent vector in the `xi` direction. Shape: (`3`, `n_gauss`).
- **A2** (np.ndarray[np.floating]): Covariant tangent vector in the `eta` direction. Shape: (`3`, `n_gauss`).

Returns

- **sps.spmatrix**: Sparse matrix `Bm` representing the Jacobian of the membrane deformation transformation in Voight notation. Shape: (`3 * n_gauss`, `3 * n_nodes`).

Notes

- The output matrix `Bm` has shape (`3 * n_gauss`, `3 * n_nodes`), where `n_gauss` is the number of quadrature points and `n_nodes` is the number of control points (basis functions).
- The isoparametric space refers to the parametric domain of the B-spline basis.

def make_membrane_stiffness_operator(
spline: bsplyne.b_spline.BSpline,
ctrl_pts: numpy.ndarray[numpy.floating]
) -> scipy.sparse._matrix.spmatrix:

- View Source

Assemble the global membrane stiffness matrix for a B-spline surface patch.

This function computes the global stiffness operator `K` for a B-spline surface patch, considering only in-plane (membrane) strain energy contributions and omitting out-of-plane (bending) effects. The assembly is performed using Gauss-Legendre quadrature over the isoparametric domain.

Parameters

- **spline** (BSpline): B-spline surface patch object defining the geometry and basis functions.
- **ctrl_pts** (np.ndarray[np.floating]): Array of control points defining the physical geometry of the surface. Shape should be (`3`, `n_nodes_xi`, `n_nodes_eta`).

Returns

- **K** (sps.spmatrix): Global stiffness matrix (`sps.spmatrix`) containing only membrane (in-plane) contributions.

Notes

- The integration is performed using quadrature points determined by the spline degrees in each isoparametric direction.
- The returned matrix `K` does not include bending or out-of-plane stiffness terms.

```
def make_membrane_stifness(
 mesh: volVIC.Mesh.Mesh,
 verbose: bool = True,
 disable_parallel: bool = False
) -> scipy.sparse._matrix.spmatrix:
```

- View Source

Assemble the global membrane stiffness matrix for a full multipatch mesh.

This function computes the global membrane stiffness matrix by summing the contributions from all patches of the mesh. Each patch is processed separately using the B-spline geometry and control points of each patch.

Parameters

- **mesh** (Mesh): Mesh object containing B-spline patches.
- **verbose** (bool, optional): If True, displays a progress bar during assembly. Default is True.
- **disable_parallel** (bool, optional): If True, disables parallel computation. Default is False.

Returns

- **sps.spmatrix**: Global membrane stiffness matrix, including all patches, ready for use in isogeometric membrane finite element analysis. The size corresponds to `3 * n_dofs`, where `n_dofs` is the total number of control points in the mesh.

Notes

- The function internally calls `make_membrane_stiffness_operator` for each patch.
- Uses parallel computation if optimal when `disable_parallel` is False.
- The resulting matrix includes only in-plane (membrane) stiffness contributions.

```
def make_membrane_weight(
 mesh: volVIC.Mesh.Mesh,
 image_energies: Iterable[volVIC.virtual_image_correlation_energy.VirtualImageCorrelationEnergyElem],
 membrane_K: scipy.sparse._matrix.spmatrix,
 rho: float = 1.5,
 image_std: float = 5000,
 expected_mean_dist: float = 5,
 n_intg: int = 100
) -> float:
```

- View Source

Compute a membrane regularization weight by matching the expected membrane energy to the expected virtual image correlation (VIC) energy.

The membrane energy is evaluated analytically from a probabilistic model of the B-spline control point displacements, while the VIC energy is estimated from finite differences of the virtual image response.

Parameters

- **mesh** (Mesh): B-spline mesh defining the control points and basis functions.
- **image_energies** (iterable of VirtualImageCorrelationEnergyElem): VIC energy objects associated with each patch.
- **membrane_K** (scipy.sparse.spmatrix): Membrane stiffness matrix of size (3 * n_bf, 3 * n_bf).
- **rho** (float, optional): Image correlation parameter passed to the virtual image operator. By default 1.5.
- **image_std** (float, optional): Standard deviation of the image noise. By default 5_000.

- **expected_mean_dist** (float, optional): Expected mean distance between the converged B-spline geometry and the target image. By default 5.
- **n_intg** (int, optional): Number of integration points used for estimating the VIC energy. By default 100.

Returns

- **membrane_weight** (float): Regularization weight such that the expected membrane energy matches the expected VIC energy.

Notes

Expected membrane energy

Each B-spline control point `a` is assigned a random displacement

```
u_a = d_a * w_a * n_a   ∈ R^3
```

where:

- `d_a = expected_mean_dist * (1 + ε_a)`, with `ε_a ~ N(0, 1)`
- `w_a = B_a(ξ_a)` is the B-spline basis evaluated at the Greville abscissa
- `n_a` is an isotropic random unit vector in R^3

The membrane stiffness matrix `K ∈ R^{3 n_bf × 3 n_bf}` is decomposed into 3×3 blocks `K_{a,b}` such that

```
[K_{a,b}]_{c,d} = K_{c * n_bf + a, d * n_bf + b}
```

The discrete membrane energy reads

```
E_mem(U) = 1/2 * Σ_{a,b} w_a w_b d_a d_b n_aᵀ K_{a,b} n_b
```

Taking the expectation and assuming independence between radial amplitudes and directions yields

```
E[E_mem] = 1/2 * Σ_{a,b} w_a w_b E[d_a d_b] E[n_aᵀ K_{a,b} n_b]
```

For a ≠ b, the isotropy of `n_b` implies

```
E[n_aᵀ K_{a,b} n_b] = 0
```

Hence only diagonal terms remain:

```
E[E_mem] = 1/2 * Σ_a w_a² E[d_a²] E[n_aᵀ K_{a,a} n_a]
```

Using isotropy:

```
E[n_a n_aᵀ] = 1/3 * I_3
⇒ E[n_aᵀ K_{a,a} n_a] = 1/3 * tr(K_{a,a}
)
```

Moreover:

```
E[d_a²] = expected_mean_dist² * E[(1 + ε_a)²] = 2 * expected_mean_dist²
```

Finally:

```
E[E_mem] = expected_mean_dist² / 3 * Σ_a w_a² * tr(K_{a,a})
```

Introducing:

```
W = (w_1, ..., w_{n_bf}, w_1, ..., w_{n_bf}, w_1, ..., w_{n_bf})ᵀ
```

this can be written compactly as:

```
E[E_mem] = expected_mean_dist² / 3 * (W² · diag(K))
```

Expected VIC energy

The observed image near the converged surface is modeled as:

```
f(γ) = g(γ + d) + σ ε
```

where ε ~ N(0,1), σ = image_std, and d is a random normal offset along the surface normal such that E(|d|) = expected_mean_dist.

The VIC energy is defined as

```
E_vic = 1/2 ∫_Ω 1/(2h) ∫_{-h}^{h} (f(γ) - g(γ))² dγ dΩ
```

Substituting the model:

```
f(γ) - g(γ) = g(γ + d) - g(γ) + σ ε
```

and taking the expectation:

```
E[E_vic] = 1/2 ∫_Ω 1/(2h) ∫_{-h}^{h} E[(g(γ + d) - g(γ))²] dγ dΩ
            + 1/2 ∫_Ω 1/(2h) ∫_{-h}^{h} E[σ² ε²] dγ dΩ
```

The cross term vanishes because E[ε] = 0. The noise term evaluates to:

```
1/2 ∫_Ω 1/(2h) ∫_{-h}^{h} σ² dγ dΩ = |Ω| σ² / 2
```

Assuming the virtual image profile g is locally antisymmetric around the surface (antisymmetry of the transition), the difference

```
g(γ + d) - g(γ)
```

is symmetric with respect to -d/2, which implies the squared difference depends only on |d|. As the law of d is otherwise unknown, the expectation is approximated by evaluating the energy for a representative offset

```
d ≈ expected_mean_dist
```

giving the final estimate:

```
E[E_vic] ≈ |Ω|/2 * ( σ² + 1/(2h) ∫_{-h}^{h} (g(γ + expected_mean_dist) - g(γ))² dγ )
```

This is exactly what is implemented in the code: the integral is computed by discrete summation on each patch.

volVIC.Problem

- View Source

class Problem:

- View Source

Virtual Image Correlation (VIC) problem definition and solver.

This class encapsulates the full setup of a surface-based VIC problem: geometry, image model, regularization, constraints, and nonlinear solver. A typical workflow is: 1. Instantiate a Problem from a mesh and an image 2. Call `solve` to estimate the displacement field 3. Export results using `save_paraview` or propagate them to a volume mesh

Attributes

- **mesh** (Mesh): Surface mesh used in the VIC problem.
- **image** (np.ndarray): Image converted to floating-point format for computations.
- **fg, bg** (float): Estimated or user-defined foreground and background gray levels.
- **Rmat** (np.ndarray): Rotation matrix resulting from ICP initialization.
- **tvec** (np.ndarray): Translation vector resulting from ICP initialization.
- **image_energies** (list[VirtualImageCorrelationEnergyElem]): Image energy elements used to assemble the VIC objective.
- **constraints** (DirichletConstraintHandler): Handler storing linear equality constraints (e.g. C^1 continuity).
- **dirichlet** (Dirichlet): Dirichlet constraint object derived from the constraint handler.
- **membrane_K** (scipy.sparse.spmatrix): Membrane stiffness matrix used for regularization.
- **membrane_weight** (float): Weight associated with the membrane regularization term.
- **initial_rho** (float): Initial value of the distance scaling parameter.
- **saved_data_0** (list[dict[str, np.ndarray]]): Cached image energy data from the first iteration, used for post-processing and visualization.

```
Problem(
 mesh: volVIC.Mesh.Mesh,
 image: numpy.ndarray[numpy.uint16],
 ICP_init: bool = True,
 reversed_normals: bool = False,
 fg_bg: Optional[tuple[float, float]] = None,
 fg_bg_method: Literal['otsu', 'interp'] = 'otsu',
 virtual_image: Callable[[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[num
 h: Optional[float] = None,
 width_dx: float = 2.0,
 surf_dx: float = 1.0,
 alpha: Union[float, tuple[tuple[float, float], tuple[float, float]]] = 0.0,
 C1_mode: Union[NoneType, Literal['auto', 'none', 'all'], numpy.ndarray[numpy.integer]] = None,
 membrane_weight: Optional[float] = None,
 initial_rho: float = 5.0,
 expected_mean_dist: float = 5.0,
 n_intg_membrane_weight_comput: int = 100,
 disable_parallel: bool = False,
 verbose: bool = True
)
```

- View Source

Initialize a Virtual Image Correlation (VIC) problem.

This constructor performs the full problem setup, including:

- foreground/background gray-level estimation,
- optional rigid-body ICP alignment between mesh and image,

- construction of virtual image energies,
- assembly of C^1 continuity constraints,
- assembly of membrane regularization,
- estimation of the coresponding regularization weight,
- initialization of all data structures required by the nonlinear solver.

After initialization, the problem is ready to be solved using `solve()`, which performs a Gauss–Newton optimization to estimate the displacement field and the distance scaling parameter.

The mesh coordinates are assumed to be expressed in image voxel units. No unit conversion or rescaling is performed internally.

Parameters

- **mesh** (Mesh): Surface mesh defining the geometry to be matched to the image. The mesh coordinates must be expressed in image voxel units. Any conversion from physical units (e.g. mm or m) to voxel coordinates must be performed beforehand by the user. If `ICP_init=True`, a rigid-body alignment is performed, but this does not handle unit conversion or rescaling. ICP only corrects for rotation and translation, assuming consistent units.
- **image** (np.ndarray[np.uint16]): 3D volumetric image (e.g. CT scan). The gray-level distribution is assumed to be bimodal (background / foreground).
- **ICP_init** (bool, optional): If `True` (default), a rigid-body ICP is performed to align the mesh with the image prior to solving the VIC problem. Disable only if the mesh is already well aligned.
- **reversed_normals** (bool, optional): If `True`, swaps the foreground/background convention in the virtual image model. Useful if the mesh normals are oriented inward instead of outward. By default, `False`.
- **fg_bg** (tuple[float, float] or None, optional): Tuple `(fg, bg)` specifying the foreground and background gray levels. If `None` (default), these values are automatically estimated from the image.
- **fg_bg_method** (Literal["otsu", "interp"], optional): Method to compute the foreground and background gray levels. `'otsu'` uses `image_utils.otsu_threshold()` to compute the gray levels based on Otsu's method. `'interp'` uses `image_utils.interp_fg_bg()` to compute the gray levels using interpolation of histogram peaks. Default is `'otsu'`.
- **virtual_image** (Callable, optional): Virtual image model mapping signed distance values to gray levels. The default (`g_slide`) is suitable for sharp material/background transitions. See `virtual_image.g_slide()` for the function signature.
- **h** (float or None, optional): Half-width of the search domain along surface normals. If `None` (default), `h` is automatically estimated from the image during initialization.
- **width_dx** (float, optional): Integration step along the normal direction for image energies.
- **surf_dx** (float, optional): Integration step along the surface for image energies.
- **alpha** (float or tuple, optional): Tangential regularization weight used in the image energies. A value of 0.0 (default) disables tangential smoothing.
- **C1_mode** ({"auto", "none", "all"} or np.ndarray or None, optional): Definition of C^1 continuity constraints between patches: - "auto": automatically selected constraints (recommended) - "none": no C^1 constraints - "all": enforce C^1 everywhere - array: user-defined triplets of constrained control points
- **membrane_weight** (float or None, optional): Weight of the membrane (elastic) regularization term. If None (default), the weight is automatically calibrated.
- **initial_rho** (float, optional): Initial value of the transition distance parameter used in the VIC model.
- **expected_mean_dist** (float, optional): Expected mean distance (in image units) used for automatic membrane weight calibration.
- **n_intg_membrane_weight_comput** (int, optional): Number of integration points used to compute the membrane weight when it is determined automatically.
- **disable_parallel** (bool, optional): If True, disables parallel execution (useful for debugging or reproducibility).
- **verbose** (bool, optional): If True, prints detailed information during initialization and subsequent computations.

mesh: volVIC.Mesh.Mesh

image: numpy.ndarray[numpy.uint16]

fg: float

bg: float

Rmat: numpy.ndarray[numpy.floating]

tvec: numpy.ndarray[numpy.floating]

image_energies: list[volVIC.virtual_image_correlation_energy.VirtualImageCorrelationEnergyElem]

constraints: IGA_for_bsplyne.Dirichlet.DirichletConstraintHandler

dirichlet: IGA_for_bsplyne.Dirichlet.Dirichlet

membrane_K: scipy.sparse._matrix.spmatrix

membrane_weight: float

initial_rho: float

saved_data_0: list[dict[str, numpy.ndarray[numpy.floating]]]

```
def find_fg_bg(
 self,
 method: Literal['otsu', 'interp'] = 'otsu',
 verbose: bool = True
):
```

- View Source

Estimate foreground and background gray levels from the input image.

This method analyzes the gray-level distribution of the volumetric image and returns representative foreground (`fg`) and background (`bg`) values. These values are used by the virtual image model to map signed distances to gray levels in the VIC formulation.

Parameters

- **method** (Literal["otsu", "interp"], optional): Strategy used to estimate foreground and background levels: - "otsu": threshold-based estimation using Otsu's method. - "interp": estimation based on interpolation of histogram peaks. Default is "otsu".
- **verbose** (bool, optional): If True, prints diagnostic information during the estimation process.

Returns

- **fg** (float): Estimated foreground gray level.
- **bg** (float): Estimated background gray level.

```
def initialize_h_mesh_ICP(
 self,
 ICP_init: bool = True,
 h: Optional[float] = None,
 disable_parallel: bool = False,
 verbose: bool = True
):
```

- View Source

Initialize the mesh placement and the normal search half-width `h` .

This method optionally performs a rigid-body Iterative Closest Point (ICP) alignment between the surface mesh

and an isosurface extracted from the image. It also initializes the normal search half-width `h`, which defines the integration domain along surface normals for the image correlation energies.

The isosurface is obtained using a marching-cubes extraction at the mid-gray level between the estimated foreground and background values.

Behavior depends on the input parameters: - If `ICP_init=True`, a rigid-body ICP alignment is performed and the resulting rotation matrix and translation vector are returned. - If `h` is `None`, the value of `h` is automatically estimated from the maximum distance between the mesh and the extracted isosurface. - If `ICP_init=False` and `h` is provided, no alignment is performed and `h` is left unchanged.

Parameters

- **ICP_init** (bool, optional): If `True` (default), performs a rigid-body ICP alignment between the mesh and the image-derived isosurface.
- **h** (float or None, optional): Normal search half-width used in image energy integration. If `None` (default), `h` is automatically estimated from the maximum mesh-to-image distance.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution during ICP and distance-field computations. By default, `False`.
- **verbose** (bool, optional): If `True`, prints diagnostic information during initialization. By default, `True`.

Returns

- **Rmat** (np.ndarray): 3x3 rotation matrix resulting from the ICP alignment. Identity if no ICP is performed.
- **tvec** (np.ndarray): Translation vector resulting from the ICP alignment. Zero vector if no ICP is performed.
- **h** (float): Initialized normal search half-width.

```
def make_membrane_weight(
 self,
 rho: Optional[float] = None,
 expected_mean_dist: float = 5.0,
 n_intg: int = 100
) -> float:
```

View Source

Compute and set the membrane regularization weight.

This method computes the weight associated with the membrane (elastic) regularization term based on the current problem configuration and image statistics. The weight is calibrated such that the regularization term is consistent with the expected image-to-surface distance scale.

The actual computation is delegated to `volVIC.membrane_stiffness.make_membrane_weight()`.

Parameters

- **rho** (float or None, optional): Value of the transition distance parameter used in the VIC model. If `None` (default), `self.initial_rho` is used.
- **expected_mean_dist** (float, optional): Expected mean signed distance (in image units) between the surface and the image features. This value is used to calibrate the membrane weight. Default is `5.0`.
- **n_intg** (int, optional): Number of integration points used to estimate the membrane weight. Higher values improve accuracy at the cost of additional computation. Default is `100`.

Returns

- **float**: Computed membrane regularization weight.

```
def make_dirichlet(self):
```

View Source

Build the reduced-DOF Dirichlet representation from linear constraints.

The object `self.constraints` stores linear equality constraints of the form `D @ u = c`. This method converts

this representation into a reduced-DOF Dirichlet mapping of the form

```
u = C @ dof + k ,
```

where `C` is a basis of the null space of `D` (`D @ C = 0`) and `k` is a particular solution satisfying `D @ k = c`.

The resulting mapping is stored in `self.dirichlet` and is used to eliminate constrained degrees of freedom in the solver.

```
def one_gauss_newton_iter(
 self,
 u_field: numpy.ndarray[numpy.floating],
 rho: float,
 verbose: bool = True,
 disable_parallel: bool = False
) -> tuple[float, numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix, float, float]:
```

- View Source

Perform a single Gauss-Newton iteration for the VIC problem.

This method updates the displacement field and the transition distance parameter `rho` by computing the incremental corrections using the Gauss-Newton scheme. The actual computation is performed by `volVIC.solve.iteration()` .

Parameters

- **u_field** (np.ndarray): Current displacement field of the mesh nodes, shape (3, N).
- **rho** (float): Current value of the transition distance parameter used in the VIC model.
- **verbose** (bool, optional): If `True` , prints diagnostic information during the iteration. By default, `True` .
- **disable_parallel** (bool, optional): If `True` , disables parallel execution during the iteration. By default, `False` .

Returns

- **du_field** (np.ndarray): Incremental displacement field computed during this iteration.
- **drho** (float): Incremental update of the transition distance parameter `rho` .

```
def solve(
 self,
 u_field: numpy.ndarray[numpy.floating] = None,
 rho: float = None,
 eps: float = 0.05,
 max_iter: int = 20,
 verbose: bool = True,
 disable_parallel: bool = False
) -> tuple[numpy.ndarray[numpy.floating], float]:
```

- View Source

Solve the VIC problem using a Gauss-Newton optimization.

This method iteratively updates the displacement field and the transition distance parameter `rho` to minimize the VIC energy. Each iteration is performed using `one_gauss_newton_iter()` .

The iteration stops when either the maximum number of iterations is reached or the relative update of the displacement field is below `eps` .

The displacement field is updated in-place and stored in `u_field` . The first iteration's data from all image energies are cached in `self.saved_data_0` for post-processing and visualization.

Parameters

- **u_field** (np.ndarray, optional): Initial displacement field of shape (3, N). If `None` (default), initializes to zero.

- **rho** (float, optional): Initial value of the distance scaling parameter. If `None` (default), `self.initial_rho` is used.
- **eps** (float, optional): Convergence tolerance. The relative norm of the displacement update is compared to `eps` to decide convergence. Default is `5e-2`.
- **max_iter** (int, optional): Maximum number of Gauss-Newton iterations. Default is `20`.
- **verbose** (bool, optional): If `True`, prints iteration diagnostics including `rho` and relative displacement updates. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution during the iterations. Default is `False`.

Returns

- **u_field** (np.ndarray): Final displacement field after convergence or reaching `max_iter`.
- **rho** (float): Final value of the transition distance parameter after convergence.

Notes

This method relies on `one_gauss_newton_iter()` to compute the incremental updates for each iteration.

```
def save_paraview(
    self,
    u_field: numpy.ndarray[numpy.floating],
    folder: str,
    name: str,
    disable_parallel: bool = False,
    verbose: bool = True
):
```

- View Source

Export the current VIC results to Paraview-compatible VTK files.

This method computes the signed distance fields before and after the displacement update, associates them with the mesh nodes, and calls the mesh's `Mesh.save_paraview()` method to save all fields for visualization in Paraview.

The exported fields include: - `u`: displacement field provided in `u_field`. - `d0`: signed distance fields at the first iteration (cached in `self.saved_data_0`). - `d`: signed distance fields at the current iteration (not computed from `u_field`).

Parameters

- **u_field** (np.ndarray): Displacement field of shape (3, N) to be saved.
- **folder** (str): Destination folder where the VTK files will be written.
- **name** (str): Base name for the exported VTK files.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution for distance field computation and mesh export. Default is `False`.
- **verbose** (bool, optional): If `True` (default), prints diagnostic messages during computation and export.

Notes

The method automatically constructs the `XI_list` from the parametric coordinates of each image energy element and sets `fields_on_interior_only="auto"` for the mesh export.

```
def plot_results(
    self,
    u_field: Optional[numpy.ndarray[numpy.floating]] = None,
    disable_parallel: bool = False,
    verbose: bool = True,
    n_colors: int = 15,
    interior_only: bool = True,
    plt_ctrl_mesh: bool = False,
    pv_plotter: Optional[pyvista.plotting.plotter.Plotter] = None,
    show: bool = True,
    elem_sep_color: str = 'black',
    ctrl_poly_color: str = 'green',
    **pv_add_mesh_kwargs
):
```

- View Source

Visualize VIC results with PyVista, optionally applying a displacement field.

This method plots the signed distance fields of the mesh with respect to the image features. If a displacement field `u_field` is provided, it is applied to a copy of the mesh for visualization; otherwise, the distances from the first iteration are displayed. It is recommended to pass the displacement field obtained from `solve` to make sure to match the distance map to the correct displacement field.

The visualization can be customized, including the number of colors, control mesh display, scalar bar, and interior-only rendering. Any additional keyword arguments are passed to `Mesh.plot()`, which ultimately forwards them to `pv.Plotter.add_mesh()`.

Parameters

- **u_field** (np.ndarray or None, optional): Displacement field to apply to the mesh for visualization. If `None` (default), the distances from the first iteration (`self.saved_data_0`) are displayed. If provided, the mesh is warped according to this displacement. To correctly display the most recent VIC results, pass the `u_field` output from the `solve` method, as the distance map displayed corresponds to the last iteration computed by the image energy terms.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation of distance fields. Default is `False`.
- **verbose** (bool, optional): If `True`, prints diagnostic messages. Default is `True`.
- **n_colors** (int, optional): Number of discrete colors in the colormap. Default is `15`.
- **interior_only** (bool, optional): If `True`, only interior elements of the mesh are plotted. Default is `True`.
- **plt_ctrl_mesh** (bool, optional): If `True`, overlays the control mesh on the plot. Default is `False`.
- **pv_plotter** (pv.Plotter or None, optional): PyVista plotter object to use. If `None`, a new plotter is created.
- **show** (bool, optional): If `True`, displays the plot immediately. Default is `True`.
- **elem_sep_color** (str, optional): Color used for separating elements. Only applied if `interior_only` is `False`. Default is `'black'`.
- **ctrl_poly_color** (str, optional): Color of the control mesh. Only applied if `plt_ctrl_mesh` is `True` and `interior_only` is `False`. Default is `'green'`.
- ****pv_add_mesh_kwargs** (dict, optional): Additional keyword arguments passed to `Mesh.plot()`, such as `cmap`, `clim`, `scalar_bar_args`, etc.

Returns

- **pv.Plotter or None**: The PyVista plotter object used for the visualization.

Notes

- The signed distance field `d` is computed from the image energies via `volVIC.VirtualImageCorrelationEnergyElem.compute_distance_field()`.
- The default colormap is a resampled `"RdBu"` with `n_colors`.
- Scalar bar properties are automatically set but can be overridden via

`pv_add_mesh_kwargs["scalar_bar_args"]` .

- To make sure the deformation matches the distance field, always pass the displacement field returned by `solve()` .

```
def propagate_displacement_to_volume_mesh(
    self,
    u_field: numpy.ndarray[numpy.floating],
    volume_mesh: volVIC.Mesh.Mesh,
    disable_parallel: bool = False
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Propagate the surface displacement field to a volumetric mesh.

This method maps the displacement field computed on the surface mesh (`self.mesh`) to a target volume mesh (`volume_mesh`). The mapping uses the surface-to-volume interpolation implemented in `Mesh.propagate_field_from_submesh()` . The resulting volumetric displacement field is returned in the same coordinate system as the original surface mesh (before ICP).

Parameters

- **u_field** (np.ndarray[np.floating]): Surface displacement field to propagate. Typically, this is the `u_field` obtained from the `solve()` method.
- **volume_mesh** (Mesh): Target volumetric mesh on which to propagate the displacement.
- **disable_parallel** (bool, optional): If `True` , disables parallel computation. Default is `False` .

Returns

- **np.ndarray[np.floating]**: Displacement field defined on the volume mesh, in the same coordinate system as the input surface displacement.

Notes

- The propagation is performed via `Mesh.propagate_field_from_submesh()` , which interpolates the surface displacement onto the volume nodes.
- The method internally applies and then removes the rigid-body rotation `self.Rmat` used during VIC initialization to ensure consistency between surface and volume coordinate frames.

volVIC.virtual_image_correlation_energy

- View Source

class VirtualImageCorrelationEnergyElem:

- View Source

Class representing the elementary VIC energy computation over a B-spline patch in isoparametric space.

This class encapsulates the setup and evaluation of the Virtual Image Correlation (VIC) energy, including the construction of integration points and operators for a normal neighborhood of a B-spline surface. It provides methods to compute the energy, its gradient, and Hessian with respect to both the B-spline control point displacements and the virtual image parameter, as well as utilities for interpolation and residual computation.

Attributes

- **spline** (BSpline): The `BSpline` surface object defining the isoparametric space and mapping.
- **ctrl_pts** (np.ndarray[np.floating]): The control points of the `BSpline` surface, as a `numpy` array of shape (3, N_xi, N_eta).
- **virtual_image** (Callable[[np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating], float], tuple[np.ndarray[np.floating], np.ndarray[np.floating]]]): Function to compute the virtual image and its derivative with respect to `rho`.
- **h** (float): Half width of the normal neighborhood.
- **xi** (np.ndarray[np.floating]): Integration points in the `xi` isoparametric direction.
- **dxi** (np.ndarray[np.floating]): Weights of the integration points in the `xi` isoparametric direction.
- **eta** (np.ndarray[np.floating]): Integration points in the `eta` isoparametric direction.
- **deta** (np.ndarray[np.floating]): Weights of the integration points in the `eta` isoparametric direction.
- **gamma** (np.ndarray[np.floating]): Integration points along the normal direction of the B-spline.
- **dgamma** (np.ndarray[np.floating]): Weights of the integration points along the normal direction.
- **Xv0** (np.ndarray[np.floating]): Flattened coordinates of the reference points in the normal neighborhood.
- **Uv_p** (sps.spmatrix): Sparse linear operator mapping control point displacements to displacements of points in the normal neighborhood.
- **wdetJs** (np.ndarray[np.floating]): xi and eta quadrature weights scaled by the surface Jacobian determinant.
- **wdetJ** (np.ndarray[np.floating]): xi, eta and gamma quadrature weights scaled by the surface Jacobian determinant.

Notes

- Integration points and weights in the isoparametric space and along the normal direction are computed automatically.
- The `virtual_image` function must return a tuple (`g` , `g_prime`), where `g` is the virtual image and `g_prime` its derivative with respect to `rho` .
- Methods are provided for generating integration grids, constructing displacement operators, interpolating image values, evaluating the virtual image, computing residuals, and assembling the VIC energy and its derivatives.

VirtualImageCorrelationEnergyElem(
spline: bsplyne.b_spline.BSpline,
ctrl_pts: numpy.ndarray[numpy.floating],
surf_dx: float,
alpha: Union[float, tuple[tuple[float, float], tuple[float, float]]],
width_dx: float,
h: float,
virtual_image: Callable[[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[num
)

- View Source

Initialize a `VirtualImageCorrelationEnergyElem` object for VIC energy computation over a B-spline surface.

This constructor sets up the isoparametric space, integration points, and operators required for evaluating the VIC energy in a normal neighborhood of the B-spline surface. It allows customization of the integration grid and the virtual image model.

Parameters

- **spline** (BSpline): The `BSpline` surface object defining the isoparametric space and mapping.
- **ctrl_pts** (np.ndarray[np.floating]): The control points of the `BSpline` surface, as a `numpy` array of shape (3, N_xi, N_eta).
- **surf_dx** (float): Target mapped distance between integration points in the isoparametric directions.
- **alpha** (Union[float, tuple[tuple[float, float], tuple[float, float]]]): Distance to ignore on the border of the patch in each isoparametric direction. If a `float`, the same value is used for all boundaries. If a tuple of tuples, ((`dist_xi_0`, `dist_xi_1`), (`dist_eta_0`, `dist_eta_1`)), each value specifies the ignored distance at the corresponding boundary.
- **width_dx** (float): Target mapped distance between integration points along the normal direction.
- **h** (float): Half-width of the neighborhood along the normal direction.
- **virtual_image** (Callable[[np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating], float], tuple[np.ndarray[np.floating], np.ndarray[np.floating]]], optional): Function to compute the virtual image and its derivative with respect to `rho`. By default, `g_slide_g_prime`.

Notes

- The integration points and weights in the isoparametric space and along the normal direction are computed automatically.
- The `virtual_image` function must return a tuple (`g`, `g_prime`), where `g` is the virtual image and `g_prime` its derivative with respect to `rho`.

spline: bsplyne.b_spline.BSpline

ctrl_pts: numpy.ndarray[numpy.floating]

virtual_image: Callable[[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], float], tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]]

h: float

xi: numpy.ndarray[numpy.floating]

dxi: numpy.ndarray[numpy.floating]

eta: numpy.ndarray[numpy.floating]

deta: numpy.ndarray[numpy.floating]

gamma: numpy.ndarray[numpy.floating]

dgamma: numpy.ndarray[numpy.floating]

Xv0: numpy.ndarray[numpy.floating]

Uv_p: scipy.sparse._matrix.spmatrix

wdetJs: numpy.ndarray[numpy.floating]

wdetJ: numpy.ndarray[numpy.floating]

saved_data: dict[str, numpy.ndarray[numpy.floating]]

```
def rigid_body_copy(self, new_ctrl_pts: numpy.ndarray[numpy.floating]):
```

• View Source

```
def make_intg_space(
 self,
 surf_dx: float,
 padding: Union[float, tuple[tuple[float, float], tuple[float, float]]],
 width_dx: float,
 h: float,
 eps: float = 1e-06
) -> tuple[tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floati
```

• View Source

Generate integration points and weights in the parametric space and along the normal direction for VIC computation.

This method returns arrays of integration points and their corresponding weights in the parametric directions (`xi` , `eta`) of the `BSpline` surface, and along the normal direction (`gamma`). Parametric points are distributed so that the mapped distance between them is close to `surf_dx` , with optional border padding specified by `padding` .

Along the normal direction (`gamma`), the interval `[-h, h]` is discretized using a midpoint rule with spacing close to `width_dx` . Two additional points are inserted symmetrically near zero at `-eps` and `+eps` to reduce potential invariance issues with respect to `rho` . Weights `dgamma` are adjusted so that the sum remains exactly `2*h` , preserving the midpoint rule.

Parameters

- **surf_dx** (float): Target mapped distance between integration points in the parametric directions.
- **padding** (Union[float, tuple[tuple[float, float], tuple[float, float]]]): Distance to ignore at the patch boundaries. If a float, the same value is applied to all boundaries. If a tuple of tuples, ((dist_xi_0, dist_xi_1), (dist_eta_0, dist_eta_1)), each value specifies the ignored distance at the corresponding boundary.
- **width_dx** (float): Target spacing between integration points along the normal direction.
- **h** (float): Half-width of the neighborhood along the normal direction.
- **eps** (float, optional): Small offset used to insert two additional points near zero (-eps and +eps). Default is 1e-6.

Returns

- **(xi, eta, gamma)** (tuple[np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating]]): Integration points in the parametric directions and along the normal direction.
- **(dxi, deta, dgamma)** (tuple[np.ndarray[np.floating], np.ndarray[np.floating], np.ndarray[np.floating]]): Integration weights corresponding to the points in `xi` , `eta` , and `gamma` .

Notes

- Parametric points are computed using `linspace_for_VIC_elem` .
- The normal direction uses a midpoint rule with uniform spacing, except near zero where ±eps points are inserted.
- The sum of `dgamma` always equals `2*h` .

```
def make_operators(
 self
) -> tuple[numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix, numpy.ndarray[numpy.floating], num
```

• View Source

Construct a linearized displacement operator and compute the initial position of a set of points in a normal neighborhood of the B-spline surface.

This operator allows exploring a tubular neighborhood of the surface by linearly extrapolating points along the local normal direction.

The displacement of a point at a signed normal distance γ from the surface is given by:

```
u_v(ξ, η, γ) = u(ξ, η) + γ * (Φ(ξ, η) × a₃(ξ, η))

Φ = φ₁ a₁ + φ₂ a₂

φ₁ =  du/dη · a₃ / ||a₁ × a₂||
φ₂ = -du/dξ · a₃ / ||a₁ × a₂||
```

where: - u is the displacement of the surface, - a_1, a_2 are the tangents to the surface, - $a_3 = (a_1 \times a_2) / ||a_1 \times a_2||$ is the surface normal (normalized), - γ is the signed distance along the normal.

The operator computed here linearly maps control point displacements to displacements of γ-sampled points along the normal.

Returns

- **Xv0** (np.ndarray[np.floating]): Flattened coordinates of the points in the reference configuration, located along the normal at a signed distance γ from the surface. Flattened array of shape (3 * n_points,)
- **Uv_p** (sps.spmatrix): Sparse linear operator that maps displacements at control points to displacements of the corresponding points in the normal neighborhood. Sparse matrix of shape (3 * n_points, 3 * n_ctrl_pts).
- **wdetJs** (np.ndarray[np.floating]): xi and eta quadrature weights scaled by the surface Jacobian determinant. Array of shape (n_surf_points,)
- **wdetJ** (np.ndarray[np.floating]): xi, eta and gamma quadrature weights scaled by the surface Jacobian determinant. Array of shape (n_points,)

def f_df_dX(
self,
X: numpy.ndarray[numpy.floating],
image: numpy.ndarray
) -> tuple[numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix]:

- View Source

Interpolate the grey level values of a voxel-based `image` at specified coordinates `X` and compute the derivative of the interpolation with respect to `X`.

Parameters

- **X** (np.ndarray[np.floating]): Coordinates of the evaluation points as continuous pixel indices, flattened as `[x_0, ..., x_n, y_0, ..., y_n, z_0, ..., z_n]`.
- **image** (np.ndarray): Voxel-based image (volume) in which the transition area is searched.

Returns

- **f** (np.ndarray[np.floating]): Array of grey levels of the `image` interpolated at the coordinates specified by `X`.
- **df_dX** (sps.spmatrix): Sparse matrix representing the derivative of the grey level interpolation with respect to the integration point coordinates `X`.

Notes

- The interpolation and its gradient are computed using the internal `TriLinearRegularGridInterpolator`.
- The returned `df_dX` is a sparse matrix with block-diagonal structure, where each block corresponds to the partial derivatives with respect to `x`, `y`, and `z` coordinates.

def g_dg_drho(
self,
rho: float
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]:

- View Source

Evaluate the virtual image and its derivative with respect to `rho` at the integration points in the isoparametric space and along the normal direction.

Parameters

- **rho** (float): Parameter of the virtual image.

Returns

- **g** (np.array of float): Virtual image at the integration points.
- **dg_drho** (np.array of float): Derivative of the virtual image with respect to `rho` at the integration points.

Notes

- The integration points are defined by the arrays `xi`, `eta`, and `gamma` in the isoparametric space and along the normal direction.
- The `virtual_image` function must return a tuple (`g`, `g_prime`), where `g` is the virtual image and `g_prime` its derivative with respect to `rho`.

```
def r_dr_dg_dr_df(
 self,
 g: numpy.ndarray[numpy.floating],
 f: numpy.ndarray[numpy.floating]
) -> tuple[numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix, scipy.sparse._matrix.spmatrix]:
```

- View Source

Compute the residual of the VIC problem and its derivatives with respect to the virtual image and the image at the integration points in the isoparametric space and along the normal direction.

Parameters

- **g** (np.ndarray[np.floating]): Virtual image evaluated at the integration points (in the isoparametric space and along the normal direction).
- **f** (np.ndarray[np.floating]): Image evaluated at the integration points (in the isoparametric space and along the normal direction).

Returns

- **r** (np.ndarray[np.floating]): Residual of the VIC problem, i.e., `g - f`, flattened as a 1D array.
- **dr_dg** (sps.spmatrix): Sparse identity matrix representing the derivative of the residual with respect to the virtual image evaluation (`g`).
- **dr_df** (sps.spmatrix): Sparse negative identity matrix representing the derivative of the residual with respect to the image evaluation (`f`).

Notes

- The input arrays `g` and `f` are reshaped internally to match the integration grid defined by (`xi`, `eta`, `gamma`).
- The returned derivatives are sparse matrices of shape (`n_points`, `n_points`), where `n_points` is the total number of integration points.

```
def E_dE_du_d2E_du2_dE_drho_d2E_drho2(
 self,
 u: numpy.ndarray[numpy.floating],
 rho: float,
 image: numpy.ndarray
) -> tuple[float, numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix, float, float]:
```

- View Source

Compute the VIC energy, its gradient, and Hessian with respect to the B-spline control point displacements (`u`) and the virtual image parameter (`rho`).

This method evaluates the energy functional by integrating the squared residual between the virtual image and the interpolated voxel-based `image` over the B-spline surface space and along the normal direction. It also computes the first and second derivatives of the energy with respect to both `u` and `rho`.

Parameters

- **u** (np.ndarray[np.floating]): Displacements of the B-spline control points in the isoparametric space. Should be a 1D array of length `3 * n_ctrl_pts`.
- **rho** (float): Virtual image parameter.
- **image** (np.ndarray): Voxel-based image (volume) on which the surface fitting is performed.

Returns

- **E** (float): Value of the VIC energy for the current parameters.
- **dE_du** (np.ndarray[np.floating]): Gradient of the energy with respect to the control point displacements (`u`). 1D array of length `3 * n_ctrl_pts`.
- **d2E_du2** (sps.spmatrix): Sparse Hessian matrix of the energy with respect to the control point displacements (`u`). Shape: (`3 * n_ctrl_pts`, `3 * n_ctrl_pts`).
- **dE_drho** (float): First derivative (scalar) of the energy with respect to the virtual image parameter (`rho`).
- **d2E_drho2** (float): Second derivative (scalar) of the energy with respect to the virtual image parameter (`rho`).

Notes

- The energy is computed as `0.5 * sum(r**2 * wdetJ)`, where `r` is the residual between the virtual image and the interpolated image, and `wdetJ` are the quadrature weights scaled by the Jacobian determinant to ensure integrating on the surface.
- The gradient and Hessian are assembled using the chain rule, leveraging the derivatives of the residual with respect to both the B-spline control point displacements and the virtual image parameter.
- Intermediate results (`r`, `f`, `g`, `rho`) are stored as attributes (`last_saved_r`, `last_saved_f`, `last_saved_g`, `last_saved_rho`) for potential later use.

```
def make_image_energies(
 mesh: volVIC.Mesh.Mesh,
 h: float,
 width_dx: float = 0.5,
 surf_dx: float = 1,
 alpha: Union[float, tuple[tuple[float, float], tuple[float, float]]] = 0.0,
 virtual_image: Callable[[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[numpy.f
 verbose: bool = True,
 disable_parallel: bool = False
) -> list[VirtualImageCorrelationEnergyElem]:
```

- View Source

Initialize surface-based Virtual Image Correlation (VIC) energy elements for a multipatch B-spline mesh.

This function constructs one `VirtualImageCorrelationEnergyElem` instance per patch of the input mesh. For each patch, it computes the operators required for surface-based VIC, initializes the integration points and weights, and stores all auxiliary data needed for subsequent global assembly.

If `mesh` is a `MeshLattice`, the computation is performed **only on the reference cell**, assuming periodic repetition of identical spline definitions across the lattice. The resulting energy elements can then be reused for all lattice cells.

Parameters

- **mesh** (Mesh or MeshLattice): B-spline multipatch mesh on which the VIC energy elements are built. For a `MeshLattice`, only the reference cell is processed.
- **h** (float): Regularization or smoothing parameter passed to the virtual image function.
- **width_dx** (float, optional): Target spacing used for the discretization along the surface width direction. Default is

`0.5`.

- **surf_dx** (float, optional): Target spacing used for the discretization along the surface parametric directions. Default is `1`.
- **alpha** (Union[float, tuple[tuple[float, float], tuple[float, float]]], optional): Ignored distance on the patch boundaries in parametric space. If a float, the same value is applied to all boundaries. If a tuple of tuples, ((`xi_min`, `xi_max`), (`eta_min`, `eta_max`)), values are specified independently for each boundary. Default is `0.0`.
- **virtual_image** (callable, optional): Virtual image function defining the image intensity and its spatial derivatives. It must have the signature `(xi, eta, gamma, rho) -> (I, dI_drho)`, where `I` is the image value and `dI_drho` its derivative wrt `rho`. Default is `g_slide()`.
- **verbose** (bool, optional): If `True`, prints progress and diagnostic information during the construction of the energy elements. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution of patch-wise computations. Default is `False`.

Returns

- **energies** (list of VirtualImageCorrelationEnergyElem): List of initialized VIC energy elements, one per patch.

Notes

- The computation is patch-wise and can be executed in parallel.
- Intended for use in surface-based Virtual Image Correlation workflows.

def plot_last_profile(
image_energies: list[VirtualImageCorrelationEnergyElem]
):

- View Source

Plot mean graylevel profiles and inter-/intra-patch standard deviations along the normal direction of the surface.

This function post-processes a list of `VirtualImageCorrelationEnergyElem` objects and visualizes the graylevel profiles stored in their `saved_data` from the **last VIC evaluation**. Statistics are computed patch-wise and then aggregated over the surface using area-weighted averages.

For each value of the normal coordinate `gamma`, the function computes:

- the global (area-weighted) mean profiles,
- the inter-patch standard deviation (variability between patches),
- the intra-patch standard deviation (variability within patches).

Two stacked plots are produced:

- **Top**: inter-patch standard deviation,
- **Bottom**: intra-patch standard deviation.

Parameters

- **image_energies** (list of VirtualImageCorrelationEnergyElem): List of initialized VIC energy elements containing saved quantities from the last image correlation evaluation. Each element must provide the following attributes:
  - `gamma`: 1D array of normal coordinates,
  - `wdetJs`: integration weights,
  - `saved_data` with keys `"last_saved_g"`, `"last_saved_f"`, and `"last_saved_r"`.

Notes

- Patch contributions are weighted by their surface area.
- `g` denotes the reference graylevel profile : the virtual image.
- `f` denotes the observed graylevel profile in the surface neighborhood.
- `r` denotes the graylevel residual.

```
def compute_image_energy_operators(
    image_energies: list[VirtualImageCorrelationEnergyElem],
    mesh: volVIC.Mesh.Mesh,
    image: numpy.ndarray,
    u_field: numpy.ndarray[numpy.floating],
    rho: float,
    verbose: bool = True,
    disable_parallel: bool = False
) -> tuple[float, numpy.ndarray[numpy.floating], scipy.sparse._matrix.spmatrix, float, float]:
```

- View Source

Evaluate image energy and assemble first- and second-order operators for a multipatch B-spline mesh.

This function evaluates the surface-based Virtual Image Correlation (VIC) energy on each patch of the mesh and assembles the resulting contributions into global quantities. For each patch, the following quantities are computed:

- image energy,
- gradient with respect to the displacement field,
- Gauss-Newton approximation of the Hessian,
- first and second derivatives with respect to the scalar parameter `rho`.

Patch-wise computations are performed independently and can be executed in parallel. Patch-level operators are then assembled into global vectors and matrices using the mesh connectivity.

Parameters

- **image_energies** (list of VirtualImageCorrelationEnergyElem): List of initialized VIC energy elements, one per patch.
- **mesh** (Mesh): Multipatch B-spline mesh used to assemble global operators.
- **image** (np.ndarray): 3D graylevel-based image containing the features to be fitted. The Virtual Image Correlation (VIC) energy defines a cost function that quantifies how well the deformed mesh matches these image features.
- **u_field** (np.ndarray[np.floating]): Global displacement field defined at the unique control points of the mesh. It is internally expanded to patch-wise (separated) representations.
- **rho** (float): Scalar parameter passed to the virtual image model.
- **verbose** (bool, optional): If `True`, prints progress information during patch-wise evaluation. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution of patch-wise computations. Default is `False`.

Returns

- **E** (float): Total image energy summed over all patches.
- **grad** (np.ndarray[np.floating]): Assembled global gradient of the image energy with respect to the displacement field.
- **H** (scipy.sparse.spmatrix): Assembled global Gauss-Newton Hessian of the image energy with respect to the displacement field.
- **dE_drho** (float): First derivative of the total image energy with respect to `rho`.
- **d2E_drho2** (float): Second derivative of the total image energy with respect to `rho`.

Notes

- The Hessian corresponds to a Gauss-Newton approximation.
- Patch-level auxiliary data stored in `image_energy.saved_data` are updated during evaluation and can be used for post-processing.
- Global assembly is handled through the mesh connectivity.

```
def compute_distance_field_patch(
 image_energy,
 f: numpy.ndarray[numpy.floating],
 rho: float,
 max_iter: int = 20
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Calculate the distance between the target profile and the real profiles in the image using Gauss-Newton optimization method.

Parameters

- **f** (np.ndarray[np.floating]): Real image profiles to compare with the target profile.
- **rho** (float): Parameter of the virtual image.
- **eps** (float, optional): Tolerance for convergence, by default 1e-3.
- **max_iter** (int, optional): Maximum number of iterations, by default 20.

Returns

- **d** (np.ndarray[np.floating]): Displacement field representing the difference between real and target profiles.

```
def compute_distance_field(
 image_energies: list[VirtualImageCorrelationEnergyElem],
 saved_data: list[dict[str, numpy.ndarray[numpy.floating]]] = None,
 verbose: bool = True,
 disable_parallel: bool = False
) -> list[numpy.ndarray[numpy.floating]]:
```

- View Source

Compute the distance field between real image profiles and target virtual profiles for a set of VIC energy elements.

This function evaluates, for each patch, the displacement field `d` that minimizes the difference between the real profiles stored in the image and the virtual profiles defined by the energy element. A Gauss-Newton iterative method is used along the normal direction (gamma) to compute the per-patch distance field.

Parameters

- **image_energies** (list of VirtualImageCorrelationEnergyElem): List of VIC energy elements, one per patch.
- **saved_data** (list of dict, optional): List of dictionaries containing the saved image profiles for each patch. Expected keys include `'last_saved_f'` and `'last_saved_rho'`. If `None`, uses the `saved_data` attribute from each energy element.
- **verbose** (bool, optional): If `True`, prints progress information. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel execution. Default is `False`.
- **max_iter** (int, optional): Maximum number of Gauss-Newton iterations per patch. Default is `20`.

Returns

- **distances** (list of np.ndarray): List of per-patch displacement fields representing the distance between real and target profiles along the normal direction.

Notes

- The displacement is clamped to the range [-h, h], where `h` is half width of search for the virtual image corellation.
- Computation is patch-wise and can be parallelized using `parallel_blocks`.

volVIC.image_utils

- View Source

@nb.njit
def hist(img: numpy.ndarray[numpy.uint16]): - View Source

Compute the gray-level histogram of an unsigned integer image.

This function counts the number of occurrences of each gray level in the input image. The histogram size is determined from the full range of the image data type (e.g. 0–65535 for `uint16`).

Parameters

- **img** (np.ndarray[np.uint16]): Input grayscale image with unsigned integer values.

Returns

- **histogram** (np.ndarray[np.uint64]): Histogram array where `histogram[g]` is the number of pixels with gray level `g`. Histogram array of size `np.iinfo(img.dtype).max + 1 = 65_536`.

Notes

- Implemented with explicit loops for compatibility with Numba `njit`.
- The histogram covers the full dynamic range of the input dtype, even if some gray levels are not present in the image.

@nb.njit
def otsu_threshold(histogram: numpy.ndarray[numpy.uint64]) -> tuple[float, float]: - View Source

Estimate foreground and background gray levels using Otsu's method.

This function applies Otsu's criterion to a gray-level histogram to find the threshold that maximizes the inter-class variance. It then returns representative mean gray levels for the foreground and background classes.

Parameters

- **histogram** (np.ndarray[np.uint64]): Gray-level histogram of an image, where each entry represents the number of pixels at a given gray level.

Returns

- **fg_value** (float): Mean gray level of the foreground class.
- **bg_value** (float): Mean gray level of the background class.

Notes

- The method assumes a bimodal histogram.
- Returned values are floating-point means, not necessarily integer gray levels.
- The foreground is defined as the class with the higher mean gray level.

def interp_fg_bg(histogram: numpy.ndarray[numpy.uint64]) -> tuple[float, float]: - View Source

Estimate foreground and background gray levels by histogram interpolation.

This method analyzes the gray-level histogram of an image and estimates representative background and foreground gray levels by:

1. Cropping the histogram to its significant support.
2. Applying a low-pass filtering in the log-histogram domain by progressively truncating the FFT spectrum.
3. Identifying major modes (peaks) and separating them using valleys.
4. Refining the peak locations by interpolation to obtain sub-bin estimates of foreground and background gray levels.

The approach is designed to be robust to noise and small secondary peaks, and is intended for bimodal or quasi-bimodal histograms commonly encountered in volumetric image correlation problems.

Parameters

- **histogram** (np.ndarray[np.uint64]): Gray-level histogram of an image, where each entry represents the number of pixels at a given gray level.

Returns

- **fg_value** (float): Estimated foreground gray level.
- **bg_value** (float): Estimated background gray level.

Notes

- The histogram is first cropped to exclude bins with negligible population (below 1% of the average bin count).
- A low-frequency approximation of the log-histogram is constructed using a limited number of Fourier coefficients, controlled by a maximum number of sign changes in its derivative.
- Foreground and background modes are identified as the dominant peaks in two regions separated by a valley in the smoothed histogram.
- Sub-bin peak locations are obtained by differentiating and rooting an interpolated histogram representation.
- This method is heuristic in nature but often more stable than Otsu's thresholding for unbalanced class distributions.

```
def find_fg_bg(
    img: numpy.ndarray[numpy.uint16],
    method: Literal['otsu', 'interp'] = 'otsu',
    save_file: Optional[str] = None,
    verbose: bool = True
) -> tuple[float, float]:
```

- View Source

Estimate background and foreground gray levels from a grayscale image histogram.

This function analyzes the gray-level histogram of an unsigned integer image (typically `uint16`) and estimates representative background (`bg`) and foreground (`fg`) gray levels using either Otsu's method or a histogram interpolation strategy.

Optionally, a histogram visualization can be displayed and saved, showing the estimated background and foreground levels.

Parameters

- **img** (np.ndarray[np.uint16]): Input grayscale image with unsigned integer values (typically `uint16`).
- **method** ({"otsu", "interp"}, optional): Method used to separate foreground and background gray levels:
  - `"otsu"`: Otsu thresholding applied to the histogram.
  - `"interp"`: Histogram-based interpolation heuristic. Default is `"otsu"`.
- **save_file** (str or None, optional): Filename used to save the histogram plot (matplotlib compatible formats accepted). If `None`, the plot is not saved. Default is `None`.
- **verbose** (bool, optional): If `True`, prints a short summary of the estimation and displays the histogram plot. Default is `True`.

Returns

- **fg_value** (float): Estimated foreground gray level.
- **bg_value** (float): Estimated background gray level.

Notes

- The histogram is internally cropped to remove near-empty bins before visualization (the full histogram is used for estimation).
- The returned values are floats, even though the input image is integer-valued.

def find_sigma_hat(image: numpy.ndarray[numpy.uint16], fg: float, bg: float) -> float: • View Source

Estimate the standard deviation of noise in a CT image by analyzing voxels confidently assigned to a single phase.

This function computes the unbiased second moment estimate of the noise standard deviation, σ, by considering only voxels whose intensities are either below the background gray level (`bg` , corresponding to air) or above the foreground gray level (`fg` , corresponding to solid material). Voxels with intensities in the ambiguous range [`bg` , `fg`] are excluded to avoid partial volume effects. For each selected voxel, the deviation from its nominal phase value is calculated, and σ is estimated as the square root of the mean squared deviation.

Parameters

- **image** (np.ndarray[np.uint16]): The input CT image as a `np.ndarray` of `np.uint16` , where voxel intensities represent X-ray absorption.
- **fg** (float): The foreground gray level (nominal value for solid material).
- **bg** (float): The background gray level (nominal value for air).

Returns

- **sigma_hat** (float): The estimated standard deviation of noise, computed from the clean phase voxels.

Notes

- Only voxels with intensities `< bg` (air) or `> fg` (material) are used for the estimation.
- Voxels in the range [`bg` , `fg`] are excluded to avoid bias from partial volume effects.
- The estimate is based on the second moment of deviations from the nominal phase values, following the folded normal distribution model.

volVIC.integration_space_image

- View Source

```
@nb.njit(nb.float64[:](nb.float64, nb.float64, nb.int64, nb.float64[:, :], nb.int64, nb.int64, nb.float64[:], nb.float64[:], nb.int64, nb.int64, nb.int64, nb.int64), cache=True)
def get_tangent(
 xi,
 eta,
 axis,
 ctrl_pts,
 p_xi,
 p_eta,
 knot_xi,
 knot_eta,
 m_xi,
 m_eta,
 n_xi,
 n_eta
):
```

- View Source

```
@nb.njit(nb.float64(nb.float64, nb.float64, nb.float64, nb.float64[:, :], nb.int64, nb.int64, nb.float64[:], nb.float64[:], nb.int64, nb.int64, nb.int64, nb.int64), cache=True)
def get_dxi(
 xi,
 eta,
 dist,
 ctrl_pts,
 p_xi,
 p_eta,
 knot_xi,
 knot_eta,
 m_xi,
 m_eta,
 n_xi,
 n_eta
):
```

- View Source

```
@nb.njit(nb.float64(nb.float64, nb.float64, nb.float64, nb.float64[:, :], nb.int64, nb.int64, nb.float64[:], nb.float64[:],
nb.int64, nb.int64, nb.int64, nb.int64), cache=True)
def get_deta(
 xi,
 eta,
 dist,
 ctrl_pts,
 p_xi,
 p_eta,
 knot_xi,
 knot_eta,
 m_xi,
 m_eta,
 n_xi,
 n_eta
):
```

- View Source

```
@nb.njit(nb.types.Tuple((nb.types.UniTuple(nb.float64[:], 2), nb.types.UniTuple(nb.float64[:], 2)))
(nb.types.UniTuple(nb.types.UniTuple(nb.float64, 2), 2), nb.int64, nb.int64, nb.float64[:], nb.float64[:],
nb.types.UniTuple(nb.float64, 2), nb.types.UniTuple(nb.float64, 2), nb.float64[:, :], nb.float64), cache=True)
def linspace_for_VIC_elem_numba(
 alpha,
 p_xi,
 p_eta,
 knot_xi,
 knot_eta,
 span_xi,
 span_eta,
 ctrl_pts,
 dx
):
```

- View Source

```
def linspace_for_VIC_elem(
 spline: bsplyne.b_spline.BSpline,
 ctrl_pts: numpy.ndarray[numpy.floating],
 dist: float = 1.0,
 alpha: Union[float, tuple[tuple[float, float], tuple[float, float]]] = 0.0
) -> tuple[tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]], tuple[numpy.ndarray[numpy.floa
```

- View Source

Compute integration points and weights in the parametric space of a `BSpline` surface, such that the mapped distance between points is close to `dist`.

This function generates arrays of integration points and their corresponding weights (step sizes) in both parametric directions (`xi` and `eta`) of a `BSpline` surface. The integration points are distributed so that the mapped distance between them, after transformation by the `BSpline`, is approximately `dist`. The ignored border distance can be set independently for each boundary using `alpha`. The computation is performed at the center of the other parametric span for each direction.

Parameters

- **spline** (BSpline): The `BSpline` surface object defining the parametric space and mapping.
- **ctrl_pts** (np.ndarray[np.floating]): The control points of the `BSpline` surface, as a `numpy` array of shape `(3, N_xi, N_eta)`.

- **dist** (float, optional): The target distance between integration points after mapping through the `BSpline`. By default, `1`.
- **alpha** (Union[float, tuple[tuple[float, float], tuple[float, float]]], optional): The distance to ignore on the border of the patch in each parametric direction. If a `float`, the same value is used for all boundaries. If a `tuple of tuples`, ((`dist_xi_0`, `dist_xi_1`), (`dist_eta_0`, `dist_eta_1`)), where each value specifies the ignored distance at the corresponding boundary. By default, `0`.

Returns

- **(xi, eta)** (tuple[np.ndarray[np.floating], np.ndarray[np.floating]]): Tuple of `numpy` arrays containing the integration points in the `xi` and `eta` isoparametric directions.
- **(dxi, deta)** (tuple[np.ndarray[np.floating], np.ndarray[np.floating]]): Tuple of `numpy` arrays containing the integration weights (step sizes) in the `xi` and `eta` directions.

Notes

- The integration points are distributed so that the mapped distance between them is approximately `dist`.
- The ignored border distance can be set independently for each boundary using `alpha`.
- The function internally calls a `numba`-accelerated implementation for performance.
- For a surface (2D), returns ((`xi` points, `eta` points), (`xi` weights, `eta` weights)).

volVIC.marching_cubes

- View Source

tri_table = show

tri_table_lens = show

edge_vertex_offsets = show

edge_to_vertices = show

key_type = UniTuple(int32, 3)

@nb.njit(cache=True)
def marching_cubes_nb(volume: numpy.ndarray, threshold: float): • View Source

def marching_cubes(volume: numpy.ndarray, threshold: float) -> meshio._mesh.Mesh: • View Source

Extract an isosurface from a 3D scalar field using the Marching Cubes algorithm and return it as a `meshio.Mesh` object.

This function calls `marching_cubes_nb`, a Numba-compiled routine that performs the core computations, including vertex deduplication and triangle generation.

Parameters

- **volume** (np.ndarray): 3D scalar field array with shape (Z, Y, X). Values of any type comparable to `threshold` can be processed without copying the data.
- **threshold** (float): Iso-value used to extract the surface. Voxels with values >= `threshold` are considered inside.

Returns

- **io.Mesh**: Triangle mesh where `points` is an array of shape (N, 3) and `cells` contains a `"triangle"` entry with indices into `points`.

Notes

- The function avoids memory-expensive copies of the input volume.
- Vertex and face generation is delegated to a Numba-compiled routine.
- To prevent the creation of zero-area triangles, vertex coordinates are slightly offset (by 1e-5) from voxel centers.

volVIC.C1_triplets

- View Source

```
@nb.njit(cache=True)
def recover_nodes_couples_from_unique_inds(
 unique_nodes_inds: numpy.ndarray[numpy.integer]
) -> numpy.ndarray[numpy.integer]:
```

- View Source

Recover all unordered pairs of node indices sharing the same unique node ID.

This function is intended for multipatch B-spline lattices, where multiple nodes may correspond to the same unique node (shared across patches). It returns all combinations of indices `(i, j)` such that both nodes correspond to the same unique node ID.

Parameters

- **unique_nodes_inds** (np.ndarray of int): Array of length `n_nodes` mapping each mesh node to its unique node ID.

Returns

- **np.ndarray of int, shape (n_couples, 2)**: Array of node index pairs `(i, j)` with `i < j`, for all nodes sharing the same unique node ID.

Notes

- The output pairs are sorted by group but not globally by node index.
- Each unique unordered pair is returned exactly once.

```
def get_all_triplets(
 unique_nodes_inds: numpy.ndarray[numpy.integer],
 shape_by_patch: numpy.ndarray[numpy.integer]
) -> numpy.ndarray[numpy.integer]:
```

- View Source

Compute triplets of nodes `(A, B, C)` for C1 continuity constraints.

For lattice or multipatch meshes, each triplet corresponds to three nodes that form a C1 constraint (smoothness across patches). This function handles corner cases and boundary nodes, ensuring only valid triplets are returned.

Parameters

- **unique_nodes_inds** (np.ndarray of int): Array mapping each mesh node to its unique node ID.
- **shape_by_patch** (np.ndarray of int, shape (n_patches, 2)): Number of nodes along each parametric direction per patch.

Returns

- **np.ndarray of int, shape (3, n_triplets)**: Array of triplets of unique node indices. Each column corresponds to a triplet (A, B, C) forming a candidate C1 continuity constraint.

Notes

- Handles corner and edge nodes correctly, discarding invalid triplets.
- Ensures that each triplet respects mesh topology and lattice boundaries.

```
def make_C1_eqs(
 mesh: volVIC.Mesh.Mesh,
 C1_inds: Union[NoneType, Literal['auto', 'none', 'all'], numpy.ndarray[numpy.integer]] = None,
 threshold: float = 0.1,
 field_size: int = 3,
 verbose: bool = True
) -> scipy.sparse._matrix.spmatrix:
```

- View Source

Generate sparse C1 continuity equations for a multipatch or lattice mesh.

Constructs a sparse matrix encoding C1 continuity constraints between triplets of nodes in the mesh. Each row corresponds to one triplet `(A, B, C)` with the equation `A - 2*B + C = 0` for enforcing smoothness.

Parameters

- **mesh** (Mesh): Multipatch B-spline mesh providing:
  - unique control points (`mesh.unique_ctrl_pts`)
  - patch topology (`mesh.connectivity`)
- **C1_inds** (None, {"auto", "none", "all"} or ndarray of int, optional): Selection mode for C1 triplets:
  - `None` or `"auto"` : automatically select triplets based on geometry. Triplets already close to C1 continuity are preserved.
  - `"none"` : do not generate any C1 constraint.
  - `"all"` : enforce C1 continuity on all valid triplets.
  - `ndarray` :
    - shape `(n_nodes,)` : select triplets whose middle node `B` belongs to the given unique node indices.
    - shape `(3, n_triplets)` : explicit `(A, B, C)` triplets.
- **threshold** (float, optional): Relative geometric tolerance used in `"auto"` mode.

  For each triplet `(A, B, C)` , the quantity::

  ```
  rhs = ||A - 2 B + C||
  ```

  is compared to the upper bound::

  ```
  up_bound = ||B - A|| + ||C - B||
  ```

  The triplet is kept if::

  ```
  rhs <= threshold * up_bound
  ```

  Default is `0.1` (10%).

- **field_size** (int, optional): Size of the field on which C1 continuity is enforced:
  - `1` for scalar fields
  - `3` for vector fields
  - etc.

  The constraint matrix is expanded block-diagonally accordingly. Default is `3` .

- **verbose** (bool, optional): If `True` , print information about selected triplets and modes. Default is `True` .

Returns

- **scipy.sparse.spmatrix**: Sparse constraint matrix of shape::

  (n_triplets * field_size, n_unique_nodes * field_size)

Each block row encodes the equation `A - 2 B + C = 0`.

Raises

- **ValueError**: If `C1_inds` has an unsupported shape or an unknown mode is provided.

Notes

- Constraints are built on **unique control points**, not per-patch DOFs.
- Vector-valued fields are handled by block-diagonal duplication.
- Intended for C1 regularization in volume-based virtual image correlation (volVIC) and related inverse problems.

volVIC.solve

- View Source

```
def iteration(
    u_field: numpy.ndarray[numpy.floating],
    rho: float,
    mesh: volVIC.Mesh.Mesh,
    image_energies: Iterable[volVIC.virtual_image_correlation_energy.VirtualImageCorrelationEnergyElem],
    image: numpy.ndarray,
    membrane_K: scipy.sparse._matrix.spmatrix,
    membrane_weight: float,
    C: scipy.sparse._matrix.spmatrix,
    verbose: bool = True,
    disable_parallel: bool = False
):
```

- View Source

Perform one iteration of the Virtual Image Correlation (VIC) energy minimization algorithm.

This function updates the displacement field and the virtual image parameter by solving a linearized system derived from the total energy, which combines the image correlation energy and an intrapatch membrane regularization. Dirichlet and interpatch C^1 constraints are imposed through the constraint matrix `C`.

Parameters

- **u_field** (np.ndarray[np.floating]): Current displacement field defined at the mesh control points.
- **rho** (float): Current virtual image parameter controlling the gray-level transformation.
- **mesh** (Mesh): Geometric mesh to be deformed during the optimization.
- **image_energies** (Iterable[VirtualImageCorrelationEnergyElem]): Collection of patchwise virtual image correlation energy elements.
- **image** (np.ndarray): Real image (experimental or reference) to which the virtual image is fitted.
- **membrane_K** (scipy.sparse.spmatrix): Sparse matrix representing the intrapatch membrane stiffness operator.
- **membrane_weight** (float): Weight factor applied to the membrane regularization term.
- **C** (scipy.sparse.spmatrix): Constraint matrix enforcing Dirichlet and C^1 coupling conditions.
- **verbose** (bool, optional): If True, display intermediate energy values and diagnostic plots. Default is True.
- **disable_parallel** (bool, optional): If True, disable parallel computation during energy operator evaluation. Default is False.

Returns

- **du_field** (np.ndarray): Increment of the displacement field obtained from the linearized system.
- **drho** (float): Increment of the virtual image parameter corresponding to intensity adaptation.

Notes

The function assembles and solves the following linearized system:

```
H_tot @ Δu = -grad_tot
```

where grad_tot = C^T (grad + w_mem * K_mem * u) H_tot = C^T (H + w_mem * K_mem) C

The scalar parameter increment Δρ is computed independently as:

```
Δρ = - (∂E/∂ρ) / (∂²E/∂ρ²)
```

volVIC.Mesh

- View Source

def in_notebook() -> bool:

- View Source

class Mesh:

- View Source

Represents a multi-patch B-spline mesh.

This class manages a multi-patch B-spline mesh, including control points, connectivity, evaluation, field propagation, and visualization. It supports both 2D and 3D meshes.

Attributes

- **connectivity** (MultiPatchBSplineConnectivity): Multi-patch connectivity structure.
- **splines** (list[BSpline]): List of B-spline objects for each patch.
- **unique_ctrl_pts** (np.ndarray): Array of unique control points for the entire mesh.
- **ref_inds** (np.ndarray): Reference indices used for linking to submeshes.

Mesh(
splines: list[bsplyne.b_spline.BSpline],
ctrl_pts: Union[list[numpy.ndarray[numpy.floating]], numpy.ndarray[numpy.floating]],
connectivity: Optional[bsplyne.multi_patch_b_spline.MultiPatchBSplineConnectivity] = **None**,
ref_inds: Optional[numpy.ndarray[numpy.integer]] = **None**
)

- View Source

Initialize a multi-patch B-spline mesh.

This constructor sets up a multi-patch mesh with B-spline patches, control points, and optional connectivity and reference indices. The mesh can be initialized either with separated control points per patch or with a packed array of unique control points. If connectivity is not provided, it is inferred automatically from separated control points.

Parameters

- **splines** (list[BSpline]): List of `BSpline` objects for each patch.
- **ctrl_pts** (list[np.ndarray] or np.ndarray): Control points of the mesh.
  - If a list of arrays is provided, each array corresponds to a patch (shape: `(dim_phys, n1, ..., nd)`).
  - If a single array is provided, it should contain either the unique control points or the full concatenated points for all patches.
- **connectivity** (MultiPatchBSplineConnectivity, optional): Multi-patch connectivity object. If `None`, it is automatically inferred from separated control points.
- **ref_inds** (np.ndarray, optional): Reference indices mapping control points to submeshes or external data. If `None`, defaults to `np.arange(nb_unique_nodes)`.

Notes

- The mesh stores control points in a "unique" packed format (`self.unique_ctrl_pts`) internally, regardless of input format.
- If `ctrl_pts` is a single array but does not match the number of unique nodes, it is automatically packed using the provided or inferred connectivity.

connectivity: bsplyne.multi_patch_b_spline.MultiPatchBSplineConnectivity

splines: list[bsplyne.b_spline.BSpline]

unique_ctrl_pts: numpy.ndarray[numpy.floating]

ref_inds: numpy.ndarray[numpy.integer]

def show(self): • View Source

Display the mesh or the last plot screenshot.

- If a screenshot from the last `PyVista` plot exists, it is displayed via `Matplotlib`.
- Otherwise, prints a simple textual representation of the mesh.

def save(self, filename: str): • View Source

Serialize and save the mesh object to a file using pickle.

Parameters

- **filename** (str): Path to the file where the mesh will be saved.

@staticmethod
def load(filename: str) -> Mesh: • View Source

Load a mesh object previously saved with `save()`.

Parameters

- **filename** (str): Path to the file containing the pickled mesh.

Returns

- **Mesh**: The loaded mesh object.

def separated_to_unique(self, field, method=**None**): • View Source

Convert a field given per patch to the unique/global control points representation.

Parameters

- **field** (list[np.ndarray]): Field separated per patch.
- **method** (optional): Agglomeration method passed to `MultiPatchBSplineConnectivity.agglomerate()`.

Returns

- **np.ndarray**: Field in the unique/global representation.

def unique_to_separated(self, field): • View Source

Convert a field in the unique/global representation to a per-patch separated form.

Parameters

- **field** (np.ndarray): Field in unique/global representation.

Returns

- **list[np.ndarray]**: Field separated per patch.

def get_separated_ctrl_pts(self) -> list[numpy.ndarray[numpy.floating]]: • View Source

Return the control points as a list of arrays, one per patch.

Returns

- **list[np.ndarray]**: Control points separated per patch.

def set_separated_ctrl_pts(self, separated_ctrl_pts: list[numpy.ndarray[numpy.floating]]): • View Source

Set the mesh control points from a list of arrays (one per patch).

Parameters

- **separated_ctrl_pts** (list[np.ndarray]): Control points separated per patch.

def assemble_grads(self, terms: list[numpy.ndarray], field_size: int) -> numpy.ndarray: • View Source

Assemble patch-wise gradient vectors into a global gradient vector.

Each entry in `terms` corresponds to a gradient vector computed on a single patch. This method uses the mesh connectivity to map patch-local indices to global unique node indices and sums contributions from overlapping nodes.

Important: Assumes that the degrees of freedom are arranged per field, i.e., `[x_0, ..., x_n, y_0, ..., y_n, z_0, ..., z_n]`. It does not work if the DOFs are interleaved as `[x_0, y_0, z_0, x_1, y_1, z_1, ...]`.

Parameters

- **terms** (list of np.ndarray): Gradient vectors for each patch, typically computed from patch-local operations.
- **field_size** (int): Number of degrees of freedom per mesh node (e.g., 3 for 3D displacements).

Returns

- **np.ndarray**: Global gradient vector of size `field_size * nb_unique_nodes`.

Notes

- Uses `MultiPatchBSplineConnectivity.unique_field_indices()` to map patch-local degrees of freedom to global indices.

def assemble_hessians(
self,
terms: list[scipy.sparse._matrix.spmatrix],
field_size: int
) -> scipy.sparse._matrix.spmatrix: • View Source

Assemble patch-wise Hessian matrices into a global sparse Hessian.

Each matrix in `terms` corresponds to the Hessian on a single patch. This method maps the local patch indices to global unique node indices using the mesh connectivity and constructs a global COO sparse matrix by summing overlapping contributions.

Important: Assumes that the degrees of freedom are arranged per field, i.e., `[x_0, ..., x_n, y_0, ..., y_n, z_0, ..., z_n]`. It does not work if the DOFs are interleaved as `[x_0, y_0, z_0, x_1, y_1, z_1, ...]`.

Parameters

- **terms** (list of scipy.sparse.spmatrix): Hessian matrices for each patch, typically in CSR or CSC format.
- **field_size** (int): Number of degrees of freedom per mesh node (e.g., 3 for 3D displacements).

Returns

- **scipy.sparse.spmatrix**: Global sparse Hessian matrix of size `(field_size * nb_unique_nodes, field_size * nb_unique_nodes)`.

Notes

- Uses `MultiPatchBSplineConnectivity.unique_field_indices()` to map patch-local degrees of freedom to global indices.
- The returned matrix is in COO format, suitable for further assembly or conversion to CSR/CSC.

def extract_border(self) -> Mesh: • View Source

Extract the exterior boundary mesh from a multi-patch B-spline mesh.

This method identifies the nodes and patches lying on the exterior of the mesh and constructs a new `Mesh` object representing only these boundary elements. The reference indices are also updated to correspond to the subset of unique nodes retained.

Returns

- **Mesh**: A new `Mesh` instance representing the exterior boundaries of the mesh.

Notes

- The `ref_inds` of the returned mesh correspond to the indices of the original mesh's unique nodes included in the boundary.
- The global layout of degrees of freedom remains the same as the parent mesh.

def subset(self, patchs_to_keep: numpy.ndarray[numpy.integer]) -> Mesh: • View Source

Create a submesh containing only the specified patches.

The method constructs a new `Mesh` object consisting of the patches indexed by `patchs_to_keep`. The unique control points and reference indices are reduced accordingly to match the retained patches.

Parameters

- **patchs_to_keep** (np.ndarray of int): Array of patch indices to retain in the submesh.

Returns

- **Mesh**: A new `Mesh` instance containing only the specified patches and the associated control points.

Notes

- The `ref_inds` of the returned submesh correspond to the subset of the original mesh's reference indices included in the retained patches.
- The global layout of degrees of freedom is preserved.
- Useful for operations on a specific region or cell of a larger multipatch mesh.

def propagate_field(
self,
field_values: numpy.ndarray[numpy.floating],
indices: numpy.ndarray[numpy.integer],
disable_parallel: bool = **False**
) -> numpy.ndarray[numpy.floating]: • View Source

Propagate a field defined on a subset of nodes to the entire mesh using IDW.

This method uses inverse distance weighting (IDW) interpolation to estimate the values of a field at unknown nodes from known nodes. The known nodes are specified by `indices` and the field values at those nodes are provided through `field_values`.

Parameters

- **field_values** (np.ndarray of float): Field values at known nodes, shape `(dof, n_known)` or `(n_known,)`. Must be of `float` type for IDW interpolation.
- **indices** (np.ndarray of int): Indices of the known nodes in `self.unique_ctrl_pts`.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation in the IDW interpolation. Default is `False`.

Returns

- **np.ndarray of float**: Field values propagated to all nodes of the mesh, same dtype as input.

Notes

- `self.unique_ctrl_pts` must be float type for IDW to work correctly.
- Interpolation is performed only for nodes not in `indices`; known node values are preserved.

```
def propagate_field_from_submesh(
    self,
    submesh: Mesh,
    field_values: numpy.ndarray[numpy.floating],
    disable_parallel: bool = False
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Propagate a field from a submesh to the full mesh using IDW interpolation.

This method first identifies the nodes in the submesh that correspond to the full mesh, then propagates the field to all nodes of the full mesh using inverse distance weighting (IDW). The values at known nodes are preserved.

Parameters

- **submesh** (Mesh): Submesh containing the nodes where the field is defined.
- **field_values** (np.ndarray of float): Field values defined on the submesh nodes. Must be of `float` type for IDW.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation in the IDW interpolation. Default is `False`.

Returns

- **np.ndarray of float**: Field values propagated to all nodes of the full mesh.

Notes

- Both `field_values` and `self.unique_ctrl_pts` must be of float type.
- Interpolation is done only for nodes not in the submesh; known node values are preserved.

```
def get_orientation_field(
    self,
    n_eval_per_elem: int = 5,
    XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
    verbose: bool = True,
    disable_parallel: bool = False
) -> list[numpy.ndarray[numpy.floating]]:
```

- View Source

Compute the orientation field of a 3D multipatch B-spline mesh.

The orientation field is calculated as the scalar triple product of the local tangent vectors of each element. It indicates the local handedness of the mesh elements (positive for right-handed orientation, negative for left-handed).

Parameters

- **n_eval_per_elem** (int, optional): Number of evaluation points per element along each parametric direction. Default is `5`.
- **XI_list** (iterable of tuple of np.ndarray, optional): Predefined parametric coordinates for each patch. If `None`, a regular grid is generated using `n_eval_per_elem`.
- **verbose** (bool, optional): If `True`, prints progress messages. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation. Default is `False`.

Returns

- **list of np.ndarray of float**: A `list` (one entry per patch) containing the orientation scalar field for at each evaluation point.

Notes

- Only implemented for 3D meshes (`self.connectivity.npa == 3`).
- Uses parallel evaluation via `parallel_blocks` .

```
def correct_orientation(
 self,
 separated_fields: list = [],
 axis: int = 2,
 verbose: bool = True,
 disable_parallel: bool = False
):
```

- View Source

Ensure consistent orientation of all patches in a 3D multipatch B-spline mesh.

This method detects and corrects inverted patch orientations by evaluating the integrated orientation field over each patch. Patches with a negative signed orientation are flipped along the specified parametric axis.

The orientation of each patch is assessed by integrating the scalar triple product of the parametric tangent vectors using Gauss-Legendre quadrature. A negative integral indicates an inverted (left-handed) parametrization.

Parameters

- **separated_fields** (list, optional): List of fields defined in separated (per-patch) representation that must be flipped consistently with the control points. Each entry is modified before returning it. Default is an empty list.
- **axis** (int, optional): Parametric axis along which inverted patches are flipped (`0 -> xi` , `1 -> eta` , `2 -> zeta`). Default is `2` .
- **verbose** (bool, optional): If `True` , prints diagnostic information, including the number of flipped patches. Default is `True` .
- **disable_parallel** (bool, optional): If `True` , disables parallel computation. Default is `False` .

Returns

- **list**: Flipped version, consistently with the control points, of `separated_fields` .

Notes

- Only applicable to 3D meshes (`self.connectivity.npa == 3`).
- Patch orientation is determined by integrating the orientation field returned by `get_orientation_field()` .
- Flipping a patch involves:
  - Reversing the knot vector of the selected parametric axis.
  - Reversing the corresponding axis of the separated control points.
  - Reversing the same axis for all fields listed in `separated_fields` .
- The mesh connectivity (`unique_nodes_inds`) and `unique_ctrl_pts` are modified to reflect the updated orientation.
- This operation modifies the mesh in place.

```
def plot_orientation(
 self,
 n_eval_per_elem: Union[int, Iterable[int]] = 5,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
 verbose: bool = True,
 disable_parallel: bool = False,
 **kwargs
) -> pyvista.plotting.plotter.Plotter:
```

- View Source

Visualize the orientation (handedness) of the mesh elements.

This method computes the orientation field of the mesh and visualizes it as a boolean field indicating whether each evaluation point corresponds to a right-handed (`True`) or left-handed (`False`) parametrization.

Internally, the orientation field is evaluated using `get_orientation_field()` , then thresholded as `orientation > 0` and reshaped to comply with plotting requirements.

Parameters

- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element along each parametric direction. If an iterable is provided, it must match the parametric dimension. Default is `5` .
- **XI_list** (iterable of tuple of ndarray, optional): Parametric coordinates at which the orientation field is evaluated, one entry per patch. If `None` (default), a uniform sampling is used for each patch.
- **verbose** (bool, optional): If `True` , displays progress information. Default is `True` .
- **disable_parallel** (bool, optional): If `True` , disables parallel computation. Default is `False` .
- ****kwargs** (dict, optional): Additional keyword arguments forwarded to `Mesh.plot()` , such as `cmap` , `scalar_bar_args` , `show_scalar_bar` , etc.

Returns

- **pv.Plotter**: The PyVista plotter object used for visualization.

Notes

- The orientation field is visualized as a boolean scalar field, where `True` indicates a right-handed parametrization and `False` a left-handed one.
- The default colormap uses red for left-handed and green for right-handed orientations.
- This method is intended as a diagnostic tool, typically used before or after calling `correct_orientation()` .

```
def plot(
    self,
    n_eval_per_elem: Union[int, Iterable[int]] = 10,
    XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
    unique_field: Optional[numpy.ndarray] = None,
    separated_field: Union[list[numpy.ndarray], list[callable], NoneType] = None,
    interior_only: bool = True,
    plt_ctrl_mesh: bool = False,
    verbose: bool = True,
    disable_parallel: bool = False,
    pv_plotter: Optional[pyvista.plotting.plotter.Plotter] = None,
    show: bool = True,
    elem_sep_color: str = 'black',
    ctrl_poly_color: str = 'green',
    **pv_add_mesh_kwargs
):
```

- View Source

Visualize the B-spline multipatch mesh using `PyVista` .

This method evaluates the mesh geometry and optional scalar fields at parametric sampling points and renders them using `PyVista` . Fields can be provided either in a unique (global) or separated (per-patch) representation.

Parameters

- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element along each parametric direction. If an iterable is provided, it must match the parametric dimension. Default is `10` .
- **XI_list** (iterable of tuple of np.ndarray, optional): Explicit parametric coordinates for evaluation, provided per patch. If `None` , a regular grid based on `n_eval_per_elem` is used.
- **unique_field** (np.ndarray, optional): Field defined on the unique control points of the mesh. Mutually exclusive with `separated_field` .
- **separated_field** (list of np.ndarray or list of callable, optional): Field defined per patch, either as arrays

evaluated on parametric grids or as callables evaluated at runtime. Mutually exclusive with `unique_field`.
- **interior_only** (bool, optional): If `True`, only interior elements are displayed. Default is `True`.
- **plt_ctrl_mesh** (bool, optional): If `True`, overlays the control polygon mesh (only if `interior_only=False`). Default is `False`.
- **verbose** (bool, optional): If `True`, prints progress messages. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation. Default is `False`.
- **pv_plotter** (pv.Plotter or None, optional): Existing `PyVista` plotter to use. If `None`, a new one is created.
- **show** (bool, optional): If `True`, renders the scene immediately. Default is `True`.
- **elem_sep_color** (str, optional): Color used for element separators. Default is `'black'`.
- **ctrl_poly_color** (str, optional): Color used for the control polygon mesh. Default is `'green'`.
- ****pv_add_mesh_kwargs** (dict, optional): Additional keyword arguments forwarded to `pv.Plotter.add_mesh` (e.g. `cmap`, `clim`, `scalar_bar_args`, etc.).

Returns

- **pv.Plotter**: The `PyVista` plotter used for rendering.

Notes

- Exactly one of `unique_field` or `separated_field` may be provided.
- Fields passed as NumPy arrays for plotting must have the same number of dimensions (`ndim`) as the control points array. Scalar fields must therefore include a leading dimension of size 1.
- Fields are evaluated on-the-fly at visualization points, not at control points.
- If executed in a Jupyter notebook, a screenshot of the last plot is stored for display via `_repr_png_`.

```
def plot_in_image(
    self,
    image: numpy.ndarray,
    threshold: Optional[float] = None,
    threshold_method: Literal['otsu', 'interp'] = 'otsu',
    mode: Literal['marching cubes', 'voxels'] = 'marching cubes',
    n_eval_per_elem: Union[int, Iterable[int]] = 10,
    XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
    interior_only: bool = True,
    plt_ctrl_mesh: bool = False,
    verbose: bool = True,
    disable_parallel: bool = False,
    pv_plotter: Optional[pyvista.plotting.plotter.Plotter] = None,
    show: bool = True,
    image_show_edges: bool = False,
    image_line_width: float = 0.05,
    image_opacity: float = 0.85,
    image_edge_color: str = 'black',
    image_interior_color: str = 'white',
    **pv_add_mesh_kwargs
):
```

- View Source

Visualize the mesh together with an image-derived surface.

This method overlays the B-spline mesh visualization with a surface extracted from a volumetric image, using either marching cubes or voxel thresholding. It is typically used to assess geometric alignment between the mesh and image features.

Parameters

- **image** (np.ndarray): Input volumetric image of shape (n1, n2, n3).
- **threshold** (float or None, optional): Threshold value used to extract the image surface. If `None`, it is

automatically estimated using `threshold_method`.
- **threshold_method** ({"otsu", "interp"}, optional): Method used to estimate the threshold when `threshold` is `None`. Default is `"otsu"`.
- **mode** ({"marching cubes", "voxels"}, optional): Method used to extract the image surface. If `"marching cubes"`, the image must be of dtype `uint16`. Default is `"marching cubes"`.
- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element along each parametric direction. Default is `10`.
- **XI_list** (iterable of tuple of np.ndarray, optional): Explicit parametric coordinates for mesh evaluation.
- **interior_only** (bool, optional): If `True`, only interior elements of the mesh are displayed. Default is `True`.
- **plt_ctrl_mesh** (bool, optional): If `True`, overlays the control polygon mesh. Default is `False`.
- **verbose** (bool, optional): If `True`, prints progress messages. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel computation. Default is `False`.
- **pv_plotter** (pv.Plotter or None, optional): Existing PyVista plotter to use. If `None`, a new one is created.
- **show** (bool, optional): If `True`, renders the scene immediately. Default is `True`.
- **image_show_edges** (bool, optional): Whether to display edges of the image-derived surface. Default is `False`.
- **image_line_width** (float, optional): Line width used when displaying image edges. Default is `0.05`.
- **image_opacity** (float, optional): Opacity of the image-derived surface. Default is `0.85`.
- **image_edge_color** (str, optional): Color of image surface edges. Default is `'black'`.
- **image_interior_color** (str, optional): Color of the image-derived surface interior. Default is `'white'`.
- ****pv_add_mesh_kwargs** (dict, optional): Additional keyword arguments forwarded to `plot()`.

Returns

- **pv.Plotter**: The PyVista plotter used for visualization.

Notes

- The mesh is rendered first, then the image-derived surface is overlaid.
- This method does not modify the mesh or image data.

```
def ICP_rigid_body_transform(
    self,
    tgt_meshio: meshio._mesh.Mesh,
    n_eval_per_elem: Union[int, Iterable[int]] = 5,
    XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None,
    plot_after: bool = True,
    disable_parallel: bool = False
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], float]:
```

- View Source

Compute and apply a rigid body transformation aligning the mesh to a target mesh.

This method computes the rigid body transformation (rotation and translation) that best aligns the current mesh to a target mesh using an Iterative Closest Point (ICP) algorithm. The transformation is then directly applied to the control points of the mesh.

Parameters

- **tgt_meshio** (io.Mesh): Target mesh used as reference for the alignment.
- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element for each parametric dimension. If an integer is provided, the same value is used for all dimensions. If an iterable is provided, each value corresponds to one parametric direction. Default is `5`.
- **XI_list** (iterable of tuple of ndarray, optional): Parametric coordinates at which the geometry is evaluated to generate the source surface mesh. If provided, this overrides `n_eval_per_elem`.
- **plot_after** (bool, optional): If `True`, displays a visualization of the aligned source mesh and the target mesh after the ICP procedure. Default is `True`.
- **disable_parallel** (bool, optional): If `True`, disables parallel evaluation. Default is `False`.

Returns

- **Rmat** (np.ndarray of float, shape (3, 3)): Rotation matrix of the rigid body transformation.
- **tvec** (np.ndarray of float, shape (3,)): Translation vector of the rigid body transformation.
- **max_dist** (float): Maximum absolute distance between the aligned source mesh and the target mesh after transformation.

Notes

- For 3D meshes, the ICP is performed on the extracted boundary surface.
- The transformation is applied as `Rmat @ x + tvec`.
- This method modifies the mesh in place by updating `self.unique_ctrl_pts`.

```
def distance_to_meshio(
 self,
 tgt_meshio: meshio._mesh.Mesh,
 n_eval_per_elem: Union[int, Iterable[int]] = 5,
 XI_list: Optional[Iterable[tuple[numpy.ndarray[numpy.floating], ...]]] = None
) -> meshio._mesh.Mesh:
```

- View Source

Compute the signed distance field to a target mesh.

This method evaluates the geometry of the mesh (or its boundary in 3D) and computes the implicit distance to the surface of a target mesh. The resulting distance field is returned as a `MeshIO` mesh.

Parameters

- **tgt_meshio** (io.Mesh): Target mesh defining the reference surface.
- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element for each parametric dimension. Default is `5`.
- **XI_list** (iterable of tuple of ndarray, optional): Parametric coordinates at which the geometry is evaluated. If provided, this overrides `n_eval_per_elem`.

Returns

- **io.Mesh**: `MeshIO` mesh containing the evaluated geometry and the associated implicit distance field.

```
def save_paraview(
 self,
 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',
 disable_parallel: bool = False
):
```

- View Source

Save the mesh geometry and associated fields as ParaView-compatible files.

This method is a convenience wrapper around `MultiPatchBSplineConnectivity.save_paraview`. It evaluates the multipatch B-spline geometry represented by the current mesh and exports several VTU files suitable for visualization in ParaView, with an optional PVD file for grouping time steps and mesh components.

The following visualization meshes are generated:

- Interior mesh representing the evaluated B-spline geometry
- Element borders mesh showing the patch discretization
- Control points mesh showing the control structure

Parameters

- **path** (str): Directory where the ParaView files will be written.
- **name** (str): Base name used for all output files.
- **n_step** (int, optional): Number of time steps to export. Default is `1`.
- **n_eval_per_elem** (int or iterable of int, optional): Number of evaluation points per element for each parametric dimension. If an integer is provided, the same value is used for all dimensions. If an iterable is provided, each value corresponds to one parametric direction. Default is `10`.
- **unique_fields** (dict, optional): Fields defined on unique control points to be exported. Keys are field names and values are numpy arrays. Callables are not supported here. Default is an empty dictionary.
- **separated_fields** (list of dict, optional): Fields defined in separated (per-patch) representation. Each list entry corresponds to one patch and contains a dictionary mapping field names to field values. See `MultiPatchBSplineConnectivity.save_paraview` for supported formats.
- **XI_list** (iterable of tuple of ndarray, optional): Parametric coordinates at which the geometry and fields are evaluated. If provided, this overrides `n_eval_per_elem`.
- **groups** (dict, optional): Dictionary describing ParaView file groups for PVD organization. If `None`, groups are created automatically.
- **make_pvd** (bool, optional): If `True`, generates a PVD file grouping all VTU files. Default is `True`.
- **verbose** (bool, optional): If `True`, prints progress information. Default is `True`.
- **fields_on_interior_only** (bool, "auto", or list of str, optional): Controls on which meshes fields are written:
  - `True`: interior mesh only
  - `False`: all meshes
  - `"auto"`: displacement-like fields are written on all meshes, others only on the interior mesh
  - list of str: names of fields to include on all meshes Default is `"auto"`.
- **disable_parallel** (bool, optional): If `True`, disables parallel evaluation. Default is `False`.

Returns

- **dict[str, dict[str, Union[str, int]]]**: Updated groups dictionary describing the generated ParaView files.

Notes

- This method don't modifies the internal state; it only performs evaluation and export.
- Geometry and fields are evaluated using the current control points and splines.
- For full details on supported field formats and file organization, see `MultiPatchBSplineConnectivity.save_paraview`.

class MeshLattice(Mesh): • View Source

Structured lattice mesh with an explicit reference B-spline cell.

This class represents a *fully instantiated* multipatch B-spline mesh that exhibits a regular lattice structure obtained by repeating a reference cell along three directions. The lattice structure is described by the repetition counts `(l, m, n)`.

The control points and connectivity describe the *entire lattice geometry*, exactly as in a standard `Mesh`. In contrast, the `BSpline` instances correspond to a single reference cell and are reused for all lattice cells.

`MeshLattice` therefore behaves as a standard `Mesh`, while additionally storing lattice metadata and providing utilities to extract or operate on the reference cell only.

This is a utility class, notably used to allow `VirtualImageCorrelationEnergyElem.make_image_energies()` and `membrane_stiffness.make_membrane_stifness()` to assemble operators on a single cell while exploiting lattice periodicity at the algebraic level.

Attributes

- **l** (int): Number of repetitions of the reference cell along the first lattice direction.
- **m** (int): Number of repetitions of the reference cell along the second lattice direction.
- **n** (int): Number of repetitions of the reference cell along the third lattice direction.

Notes

- The lattice geometry is assumed to be fully defined at construction time.
- No geometric replication is performed internally.
- All lattice cells share identical B-spline definitions.
- The total number of patches is `l * m * n * nb_patchs_cell`.

MeshLattice(
l: int,
m: int,
n: int,
splines_cell: list[bsplyne.b_spline.BSpline],
ctrl_pts: Union[list[numpy.ndarray[numpy.floating]], numpy.ndarray[numpy.floating]],
connectivity: Optional[bsplyne.multi_patch_b_spline.MultiPatchBSplineConnectivity] = **None**,
ref_inds: Optional[numpy.ndarray[numpy.integer]] = **None**
)

- View Source

Initialize a lattice mesh from a reference B-spline cell. See `Mesh.__init__()` for more details.

Parameters

- **l** (int): Number of repetitions along the first lattice direction.
- **m** (int): Number of repetitions along the second lattice direction.
- **n** (int): Number of repetitions along the third lattice direction.
- **splines_cell** (list of BSpline): B-spline patches defining a single reference cell.
- **ctrl_pts** (list of np.ndarray or np.ndarray): Control points of the full lattice structure, in separated or unique representation.
- **connectivity** (MultiPatchBSplineConnectivity or None, optional): Connectivity describing the multipatch topology of the full lattice structure. If `None` (default), it is inferred automatically.
- **ref_inds** (np.ndarray of int or None, optional): Optional reference indices associated with the control points.

l: int

m: int

n: int

@classmethod
def from_mesh(
cls,
mesh: Mesh,
l: int,
m: int,
n: int
) -> MeshLattice:

- View Source

Construct a lattice mesh from an existing mesh.

This method interprets the input `mesh` as a lattice composed of `l * m * n` identical cells and reconstructs a `MeshLattice` instance using the corresponding reference cell splines.

Parameters

- **mesh** (Mesh): Existing mesh representing a lattice geometry.
- **l** (int): Number of repetitions along the first lattice direction.
- **m** (int): Number of repetitions along the second lattice direction.
- **n** (int): Number of repetitions along the third lattice direction.

Returns

- **MeshLattice**: A lattice mesh sharing the control points, connectivity and reference indices of `mesh` .

Notes

- The reference cell is inferred from the first `mesh.connectivity.nb_patchs // (l * m * n)` patches.
- No geometry or connectivity modification is performed; the operation is purely structural.

def get_mesh_cell_one(self) -> Mesh:

- View Source

Extract the reference cell mesh corresponding to a single lattice cell.

This method returns a `Mesh` representing one reference cell of the lattice, including:

- the corresponding subset of splines,
- the associated control points,
- a reduced connectivity.

Returns

- **Mesh**: Mesh corresponding to a single lattice cell.

Notes

- The returned mesh is independent from the lattice structure but shares the same control point values.
- This is typically used for analysis or visualization of the unit cell.

def get_nb_patchs_cell(self) -> int:

- View Source

Return the number of patches in a single lattice cell.

Returns

- **int**: Number of B-spline patches defining the reference cell.

Inherited Members

Mesh connectivity, splines, unique_ctrl_pts, ref_inds, show(), save(), load(), separated_to_unique(), unique_to_separated(), get_separated_ctrl_pts(), set_separated_ctrl_pts(), assemble_grads(), assemble_hessians(), extract_border(), subset(), propagate_field(), propagate_field_from_submesh(), get_orientation_field(), correct_orientation(), plot_orientation(), plot(), plot_in_image(), ICP_rigid_body_transform(), distance_to_meshio(), save_paraview()