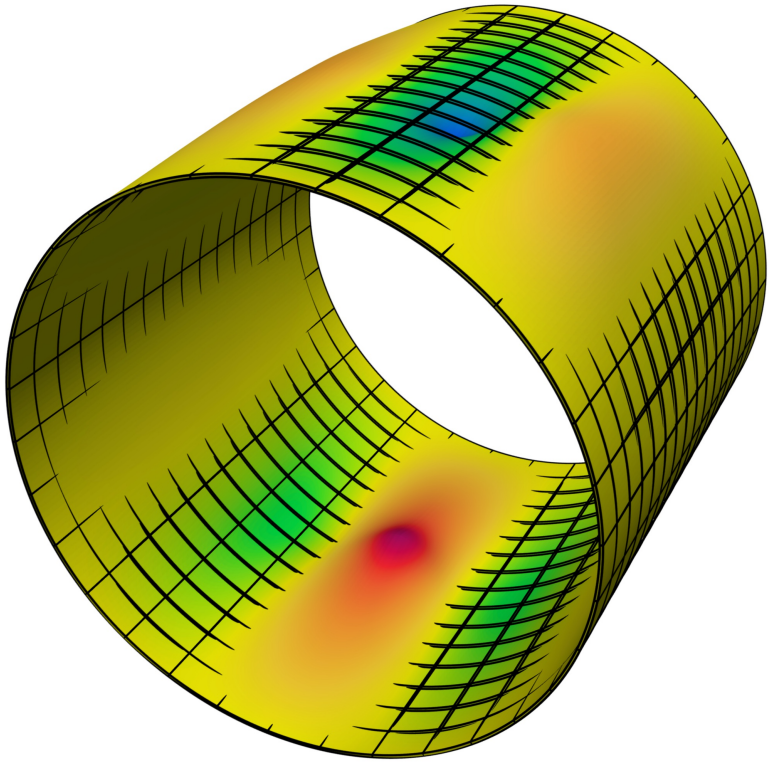
IGA_for_bsplyne

IGA_for_bsplyne



IGA_for_bsplyne is a high-performance Python library for 3D Isogeometric Analysis (IGA) applied to linear elasticity. It bridges the gap between Computer-Aided Design (CAD) and Finite Element Analysis (FEA) by using B-Spline basis functions directly as the approximation space for structural mechanics.

This library extends the geometric foundations of bsplyne to provide a complete multipatch IGA solver pipeline.

Installation

1. High-Performance Solvers (Optional but Recommended)

For large-scale 3D problems, **IGA_for_bsplyne** leverages `SuiteSparse` (via `scikit-sparse` and `sparseqr`) to achieve significant speedups.

- **Recommended Installation (Conda)**: Conda is highly recommended as it automatically manages the complex non-Python dependencies (SuiteSparse). Run these commands in order:

  ```
  conda install -c conda-forge scikit-spars
  e
  pip install sparseqr
  ```

  Note: `sparseqr` is not on Conda, but it will successfully link to the SuiteSparse libraries installed by scikit-sparse.

- **Alternative (System Package Manager)**: If you are not using Conda, you must install SuiteSparse manually. **Note:** This method is not recommended for most users as it often leads to complex compilation issues during the `pip` phase:

  - **macOS**: `brew install suitesparse && pip install scikit-sparse sparseqr`
  - **Ubuntu/Debian**: `sudo apt-get install libsuitesparse-dev && pip install scikit-sparse sparseqr`

- **Fallback**: If these packages are not detected, the library automatically falls back to standard `scipy.sparse` solvers and an internal Sparse QR implementation, ensuring full compatibility at the cost of performance.

2. Install the Library

Standard Installation

Install the latest stable version directly from PyPI:

```
pip install IGA-for-bsplyne
```

Development Installation

If you wish to modify the code or contribute to the project, install the library in editable mode:

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

Note: The core dependency bsplyne is handled automatically by pip.

Core Architecture

The library is organized into three specialized layers:

🛠 Constraint Management: `Dirichlet`

Handles complex boundary conditions via an affine mapping approach: **u = Cd + k**.

- **Static condensation**: Automatically reduces the system size by removing constrained DOFs.
- **Advanced Relations**: Supports slave-reference dependencies and Rigid Body constraints.
- **Key tools**: `DirichletConstraintHandler` for automated multi-point constraint assembly.

Local Physics: `IGAPatch`

The bridge between B-Spline geometry and continuum mechanics.

- **Operators**: High-performance assembly of Stiffness matrix (**K**) and RHS vector (**f**).
- **Post-Processing**: Integrated calculation of Strains, Stresses, and Von Mises invariants.
- **Digital Twin & Immersed Models**: Supports `IGAPatchDensity` to model heterogeneous materials (e.g., from **CT-scan imagery**) by interpolating volume fractions directly into the B-Spline basis.

🌐 Global Solver: `ProblemIGA`

Orchestrates the multipatch environment and global resolution.

- **Connectivity**: Seamlessly "welds" patches together using `MultiPatchBSplineConnectivity`.
- **Solvers**: Choice between Direct (Cholesky) and Iterative (PCG) solvers.
- **Visualization**: Native ParaView (.pvd/.vtu) export.

Quick Start

```
from IGA_for_bsplyne.IGAPatch import IGAPatch
from IGA_for_bsplyne.ProblemIGA import ProblemIGA

# Define your patch
patch = IGAPatch(spline, ctrl_pts, E=70e9, nu=0.33)

# Assemble and solve
prob = ProblemIGA([patch], connectivity, dirichlet)
u = prob.solve()

# Export results
prob.save_paraview(u, path="./", name="results")
```

Tutorials & Examples

A comprehensive hands-on guide is available in `examples/tutorial/`:

1. **Affine Mappings** (01-03)
2. **Local Integration & Post-Processing** (04-06)

3. **Immersed Models & Density** (07)
4. **Multipatch Solving** (08-09)

Documentation

The full API documentation is available in the `docs/` directory or via the Online Portal.

License

This project is licensed under the **CeCILL License**.

- View Source

IGA_for_bsplyne.IGAPatch

- View Source

class IGAPatch:

- View Source

Local representation of a 3D B-spline patch for linear elasticity.

This class provides the building blocks to compute linear elasticity operators (stiffness matrix, right-hand side, strain, stress, von Mises stress) for a single B-spline volume. It is intended **solely as a component of a multipatch problem** and is not designed to be used as a standalone solver.

Typically, IGAPatch objects are managed by a `ProblemIGA` instance, which assembles multiple patches, enforces Dirichlet constraints, and solves the global linear elasticity problem.

Features provided by IGAPatch include: - computation of the Jacobian and its determinant, - gradients of shape functions in physical space, - linear elastic stiffness matrix assembly for one patch, - evaluation of prescribed surface forces, - post-processing quantities such as displacement, strain, stress, and von Mises stress, - export of results to visualization tools (e.g., ParaView).

Attributes

- **spline** (BSpline): The 3D B-spline volume representing the patch geometry and shape functions.
- **ctrl_pts** (np.ndarray[np.floating]): Array of control points of shape (3, n_xi, n_eta, n_zeta) defining the patch geometry.
- **xi** (np.ndarray[np.floating]): Isoparametric integration points along the xi direction.
- **dxi** (np.ndarray[np.floating]): Corresponding quadrature weights for xi.
- **eta** (np.ndarray[np.floating]): Isoparametric integration points along the eta direction.
- **deta** (np.ndarray[np.floating]): Corresponding quadrature weights for eta.
- **zeta** (np.ndarray[np.floating]): Isoparametric integration points along the zeta direction.
- **dzeta** (np.ndarray[np.floating]): Corresponding quadrature weights for zeta.
- **F_N** (np.ndarray[np.floating]): Prescribed surface forces, with shape (3 (direction), 2 (side: front/back), 3 (physical components)). Used for computing the right-hand side vector.
- **H** (np.ndarray[np.floating]): Constitutive matrix in Voigt notation (6x6) for linear isotropic elasticity. Defined from `E` and `nu`.

Notes

- All integration points are generated using Gaussian quadrature with a number of points determined from the spline degree.
- This class **does not handle multipatch connectivity or global constraints**. Use `ProblemIGA` to assemble and solve multipatch problems.
- Post-processing quantities can be evaluated at the integration points or exported to visualization tools like ParaView.

```
IGAPatch(
 spline: bsplyne.b_spline.BSpline,
 ctrl_pts: numpy.ndarray[numpy.floating],
 E: float,
 nu: float,
 F_N: numpy.ndarray[numpy.floating] = array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

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

- View Source

Initialize a local `IGAPatch` for linear elasticity computations.

This constructor sets up all necessary information for assembling the stiffness matrix and right-hand side for a single B-spline patch. Note that `IGAPatch` objects are intended to be used as components of a `ProblemIGA` multipatch object, and are **not standalone solvers**.

Parameters

- **spline** (BSpline): The 3D B-spline volume defining the patch geometry and the shape functions. Provides methods to compute basis functions and derivatives.
- **ctrl_pts** (np.ndarray[np.floating]): Array of control points defining the patch geometry. Shape should be (3, n_xi, n_eta, n_zeta) corresponding to (physical_dim, n_ctrl_pts_xi, n_ctrl_pts_eta, n_ctrl_pts_zeta).
- **E** (float): Young's modulus of the material.
- **nu** (float): Poisson's ratio of the material.
- **F_N** (np.ndarray[np.floating], optional): Prescribed surface forces applied on the patch boundaries. Shape should be (3 (direction: x, y, z), 2 (side: front/back), 3 (physical components)), representing forces at the integration points on each face. Default is zero (no surface forces).

Attributes set

H : np.ndarray[np.floating] 6×6 constitutive matrix in Voigt notation, built from `E` and `nu` . xi, eta, zeta : np.ndarray[np.floating] Isoparametric coordinates for Gaussian quadrature in each direction. dxi, deta, dzeta : np.ndarray[np.floating] Quadrature weights associated with the isoparametric coordinates. F_N : np.ndarray[np.floating] Prescribed surface forces, stored for use in the rhs computation.

Notes

- Quadrature points and weights are automatically generated using `spline.gauss_legendre_for_integration` with `degree+1` points.
- The constitutive matrix `H` assumes linear isotropic elasticity in Voigt notation.
- This patch only defines the local operators; global assembly and enforcement of boundary conditions are handled by `ProblemIGA` .

spline: bsplyne.b_spline.BSpline

ctrl_pts: numpy.ndarray[numpy.floating]

xi: numpy.ndarray[numpy.floating]

dxi: numpy.ndarray[numpy.floating]

eta: numpy.ndarray[numpy.floating]

deta: numpy.ndarray[numpy.floating]

zeta: numpy.ndarray[numpy.floating]

dzeta: numpy.ndarray[numpy.floating]

F_N: numpy.ndarray[numpy.floating]

H: numpy.ndarray[numpy.floating]

def jacobian(
 self,
 dN_dXI: tuple[scipy.sparse._matrix.spmatrix, scipy.sparse._matrix.spmatrix, scipy.sparse._matrix.spmatrix]
) -> tuple[numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating], numpy.ndarray[numpy.floating]]:

- View Source

Compute the Jacobian matrix of the mapping from parametric to physical space, its inverse, and its determinant at all integration points.

The Jacobian describes how the parametric (xi, eta, zeta) coordinates are mapped to physical coordinates using the patch's control points.

Parameters

- **dN_dXI** (tuple[sps.spmatrix, sps.spmatrix, sps.spmatrix]): Derivatives of the shape functions with respect to the parametric coordinates. Tuple of sparse matrices `(dN_dxi, dN_deta, dN_dzeta)`, each of shape (n_intg_pts, n_ctrl_pts).

Returns

- **J** (np.ndarray[np.floating]): Jacobian matrices at each integration point, shape `(3, 3, n_intg_pts)` corresponding to (physical_dim, param_dim, n_intg_pts).
- **Jinv** (np.ndarray[np.floating]): Inverse of the Jacobian matrices, shape `(3, 3, n_intg_pts)` corresponding to (param_dim, physical_dim, n_intg_pts).
- **detJ** (np.ndarray[np.floating]): Determinant of the Jacobian at each integration point, shape `(n_intg_pts,)`.

Notes

- `J[:, :, i]` gives the 3×3 Jacobian matrix at the i-th integration point.
- `Jinv[:, :, i]` is the inverse of that matrix, used to compute derivatives with respect to physical coordinates.
- `detJ[i]` is the determinant of `J[:, :, i]`, used for integration weights.

def grad_N(
 self,
 Jinv: numpy.ndarray[numpy.floating],
 dN_dXI: tuple[scipy.sparse._matrix.spmatrix, scipy.sparse._matrix.spmatrix, scipy.sparse._matrix.spmatrix]
) -> numpy.ndarray[numpy.object_]:

- View Source

Compute the gradient of the B-spline shape functions with respect to physical coordinates.

The gradient is obtained by applying the chain rule: dN/dX_i = $[J^{-1}]_{ij}$ @ dN/dXI_j for each integration point.

Parameters

- **Jinv** (np.ndarray[np.floating]): Inverse Jacobian matrices at all integration points, shape `(3, 3, n_intg_pts)` corresponding to (param_dim, physical_dim, n_intg_pts).
- **dN_dXI** (tuple[sps.spmatrix, sps.spmatrix, sps.spmatrix]): Derivatives of shape functions with respect to parametric coordinates `(dN_dxi, dN_deta, dN_dzeta)`. Each sparse matrix has shape `(n_intg_pts, n_ctrl_pts)`.

Returns

- **dN_dX** (np.ndarray[np.object_]): Gradients of the shape functions with respect to physical coordinates. Numpy array of shape `(3,)`, where each element is a `sps.spmatrix` of shape `(n_intg_pts, n_ctrl_pts)` corresponding to the derivative along each physical axis.

Notes

- `dN_dX[i]` gives the derivative of all shape functions along the i-th physical direction at all integration points.
- Each sparse matrix can be directly used in the assembly of the stiffness matrix or other integrals over the physical domain.
- The use of `np.object_` array allows each entry to hold a separate sparse matrix.

def make_W(
self,
detJ: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]: • View Source

Compute the integration weights in physical space for Gaussian quadrature.

The integration measure W at each quadrature point combines:

1. The local volume scaling from the Jacobian determinant `detJ`.
2. The parametric quadrature weights (`dxi`, `deta`, `dzeta`) in each direction.

Mathematically: W = |detJ| * dxi ⊗ deta ⊗ dzeta with * the pointwise multiplication and ⊗ the Kronecker product.

Parameters

- **detJ** (np.ndarray[np.floating]): Determinant of the Jacobian at each integration point. Shape: (n_intg_pts,)

Returns

- **W** (np.ndarray[np.floating]): Integration weights in physical space for each quadrature point. Shape: (n_intg_pts,)

def stiffness(self) -> scipy.sparse._matrix.spmatrix: • View Source

Compute the linear elasticity stiffness matrix for this IGAPatch.

This method constructs the 3D linear elasticity stiffness matrix using Gaussian quadrature in the parametric space of the B-spline patch. The computation follows standard isogeometric analysis (IGA) procedure:

1. Evaluate the derivatives of the B-spline shape functions with respect to the parametric coordinates (dN/dXI).
2. Compute the Jacobian, its inverse, and determinant at all integration points.
3. Transform shape function derivatives to the physical space (dN/dX).
4. Build the strain-displacement matrices B (Bxx, Byy, Bzz, Bxy, Byz, Bxz).
5. Assemble the stiffness contributions using the material elasticity tensor H and the integration weights W derived from the Jacobian.
6. Sum all contributions to obtain the global stiffness matrix `K`.

Returns

- **K** (sps.spmatrix): Sparse stiffness matrix of the patch. Shape: (3 * n_ctrl_pts, 3 * n_ctrl_pts), in physical coordinates.

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

Compute the right-hand side (load) vector for this `IGAPatch`.

This method assembles the contribution of the surface forces applied on the patch boundary. It uses Gaussian

quadrature along the patch faces where forces are defined. The procedure is:

1. Loop over each physical axis (x, y, z) and each side of the patch (0=lower, 1=upper).
2. Skip faces with zero surface forces.
3. For each face with a force: a. Select the isoparametric coordinate corresponding to the face. b. Compute the shape functions `N` and their derivatives along the face. c. Compute the surface measure `dS` using the cross product of tangent vectors to the surface and parametric quadrature weights. d. Integrate the contribution of the shape functions over the surface.
4. Sum all face contributions into the global right-hand side vector.

Returns

- **rhs** (np.ndarray[np.floating]): Right-hand side vector for the patch. Shape: (3 * n_ctrl_pts,), in physical coordinates.

def area_border(self, axis: int, front_side: bool) -> float: • View Source

Compute the total surface area of a patch face across a given axis.

The face is defined by fixing one parametric coordinate (`xi` , `eta` , or `zeta`) to either its front or back boundary. The surface area is computed by integrating the differential surface measure over the face using the Gaussian quadrature points and weights.

Parameters

- **axis** (int): Axis across which the face is oriented: 0 → xi, 1 → eta, 2 → zeta.
- **front_side** (bool): If True, select the "front" boundary across the given axis; otherwise, select the "back" boundary.

Returns

- **area** (float): Total surface area of the specified patch face in physical space.

def epsilon(
self,
U: numpy.ndarray[numpy.floating],
XI: list[numpy.ndarray[numpy.floating]]
) -> numpy.ndarray[numpy.floating]: • View Source

Compute the strain tensor in Voigt notation for the patch.

The strain tensor is computed as: eps = [ε_xx, ε_yy, ε_zz, ε_xy, ε_yz, ε_xz]^T using the displacement field `U` and the isoparametric coordinates `XI` . Small adjustments are applied at the parametric boundaries to avoid evaluation exactly on the control point spans.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field of shape (3, nb_ctrl_pts), with the first dimension corresponding to the physical coordinates (x, y, z).
- **XI** (list[np.ndarray[np.floating]]): List of arrays for the parametric coordinates in each direction [xi, eta, zeta].

Returns

- **eps** (np.ndarray[np.floating]): Strain tensor in Voigt notation, shape (6, n_param_pts), where n_param_pts = product of lengths of XI arrays.

def sigma(
self,
eps: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]: • View Source

Compute the stress tensor in Voigt notation for the patch.

The stress tensor is computed as: sigma = H @ eps where H is the linear elasticity matrix of the material.

Parameters

- **eps** (np.ndarray[np.floating]): Strain tensor in Voigt notation, shape (6, n_param_pts).

Returns

- **sig** (np.ndarray[np.floating]): Stress tensor in Voigt notation, shape (6, n_param_pts).

```
def sigma_eig(
 self,
 sig: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Compute the principal stresses (eigenvalues of the stress tensor) at the specified parametric points.

The stress tensor is first converted from Voigt notation to full 3x3 tensors before computing eigenvalues.

Parameters

- **sig** (np.ndarray[np.floating]): Stress tensor in Voigt notation, shape (6, n_param_pts).

Returns

- **sig_eig** (np.ndarray[np.floating]): Principal stresses (eigenvalues) of shape (n_param_pts, 3), sorted by magnitude.

```
def von_mises(
 self,
 sig_eig: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Compute the von Mises equivalent stress at the specified parametric points.

The von Mises stress is computed from the principal stresses (eigenvalues of the stress tensor) as a scalar measure of the deviatoric stress.

Parameters

- **sig_eig** (np.ndarray[np.floating]): Principal stresses (eigenvalues) of shape (n_param_pts, 3), sorted by magnitude.

Returns

- **vm** (np.ndarray[np.floating]): Von Mises stress at each parametric point, shape (n_param_pts,).

```
def save_paraview(
 self,
 U: numpy.ndarray[numpy.floating],
 path: str,
 name: str,
 n_eval_per_elem: int = 10
):
```

- View Source

Export the patch data to a ParaView-readable format.

The saved data includes the displacement field and derived fields (strain, stress, principal stresses, von Mises stress) for visualization.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field, shape (3, nb_ctrl_pts), with physical coordinates first.
- **path** (str): Directory path where the ParaView file will be saved.
- **name** (str): Base name of the saved file.
- **n_eval_per_elem** (int, optional): Number of evaluation points per element in each parametric direction (higher values produce smoother visualizations). Default is 10.

```
def make_paraview_fields(
    self,
    U: numpy.ndarray[numpy.floating],
    XI: list[numpy.ndarray[numpy.floating]]
) -> dict[str, numpy.ndarray[numpy.floating]]:
```

View Source

Generate fields for visualization in ParaView from the displacement field.

This function computes the following fields: - Displacement `U` - Strain tensor `epsilon` in Voigt notation - Stress tensor `sigma` in Voigt notation - Principal stresses `sigma_eig` - Von Mises stress `von_mises`

The resulting arrays are reshaped to match the expected ParaView format: - The first dimension is for potential time/step (here 1) - Subsequent dimensions correspond to physical or Voigt components, followed by the parametric grid shape defined by `XI` .

Parameters

- **U** (np.ndarray[np.floating]): Displacement field of shape (3, nb_ctrl_pts), with the first dimension corresponding to physical coordinates (x, y, z).
- **XI** (list[np.ndarray[np.floating]]): List of parametric coordinates for each direction [xi, eta, zeta]. Determines the evaluation grid for the derived fields.

Returns

- **fields** (dict[str, np.ndarray[np.floating]]): Dictionary of fields ready for ParaView, with the following shapes: - "U" : shape (1, 3, nxi, neta, nzeta) - "epsilon" : shape (1, 6, nxi, neta, nzeta) - "sigma" : shape (1, 6, nxi, neta, nzeta) - "sigma_eig" : shape (1, 3, nxi, neta, nzeta) - "von_mises" : shape (1, 1, nxi, neta, nzeta) where (nxi, neta, nzeta) = tuple(xi.size for xi in XI)

class IGAPatchDensity(IGAPatch):

View Source

Density-weighted representation of a 3D B-spline patch for linear elasticity.

This class extends `IGAPatch` by allowing a spatially varying material density. The density field is defined at the control points and interpolated over the parametric domain, scaling the constitutive matrix locally for both stiffness assembly and stress evaluation.

IGAPatchDensity is intended as a component of a multipatch problem and is not a standalone solver. Typically, objects of this class are managed by a `ProblemIGA` instance that assembles multiple patches, enforces Dirichlet constraints, and solves the global elasticity problem.

Features provided by IGAPatchDensity include: - computation of the Jacobian and its determinant, - gradients of shape functions in physical space, - density-weighted linear elastic stiffness matrix assembly, - evaluation of prescribed surface forces, - post-processing quantities such as displacement, strain, stress, principal stresses, von Mises stress, and local density, - export of results to visualization tools (e.g., ParaView).

Attributes

- **spline** (BSpline): The 3D B-spline volume representing the patch geometry and shape functions.
- **ctrl_pts** (np.ndarray[np.floating]): Array of control points of shape (3, n_xi, n_eta, n_zeta) defining the patch geometry.
- **xi** (np.ndarray[np.floating]): Isoparametric integration points along the xi direction.
- **dxi** (np.ndarray[np.floating]): Corresponding quadrature weights for xi.

- **eta** (np.ndarray[np.floating]): Isoparametric integration points along the eta direction.
- **deta** (np.ndarray[np.floating]): Corresponding quadrature weights for eta.
- **zeta** (np.ndarray[np.floating]): Isoparametric integration points along the zeta direction.
- **dzeta** (np.ndarray[np.floating]): Corresponding quadrature weights for zeta.
- **F_N** (np.ndarray[np.floating]): Prescribed surface forces, shape (3 (direction), 2 (side: front/back), 3 (physical components)), used for computing the right-hand side vector.
- **H** (np.ndarray[np.floating]): Constitutive matrix in Voigt notation (6x6) for linear isotropic elasticity. Defined from `E` and `nu`.
- **d** (np.ndarray[np.floating]): Density field expressed in the B-spline basis, used to scale the constitutive matrix locally. At evaluation, values are clipped to [0, 1].

Notes

- All integration points are generated using Gaussian quadrature with a number of points determined from the spline degree.
- The density field modifies the stiffness and stress at each integration point.
- This class does not handle multipatch connectivity or global constraints. Use `ProblemIGA` to assemble and solve multipatch problems.
- Post-processing quantities can be evaluated at the integration points or exported to visualization tools like ParaView.

```
IGAPatchDensity(
 spline: bsplyne.b_spline.BSpline,
 ctrl_pts: numpy.ndarray[numpy.floating],
 E: float,
 nu: float,
 d: numpy.ndarray[numpy.floating],
 F_N: numpy.ndarray[numpy.floating] = array([[[0., 0., 0.],
        [0., 0., 0.]],

       [[0., 0., 0.],
        [0., 0., 0.]],

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

- View Source

Initialize a density-weighted IGA patch.

This constructor extends `IGAPatch.__init__()` by introducing a control-point-based density field and by initializing Gaussian quadrature points for volumetric integration including the density field.

The density field is reshaped to match the control point grid.

Parameters

- **spline** (BSpline): B-spline volume defining the parametric domain.
- **ctrl_pts** (np.ndarray[np.floating]): Control points defining the patch geometry.
- **E** (float): Young's modulus of the material.
- **nu** (float): Poisson's ratio of the material.
- **d** (np.ndarray[np.floating]): Density field expressed in the B-spline basis. Evaluated density (N @ d) will be clipped to the range [0, 1].
- **F_N** (np.ndarray[np.floating], optional): Surfacic Neumann forces applied on the patch boundaries. Shape: (3, 2, 3). Default is zero.

d

def stiffness(self) -> scipy.sparse._matrix.spmatrix: • View Source

Assemble the density-weighted stiffness matrix.

This method follows the same formulation as `IGAPatch.stiffness()`, but the constitutive matrix is scaled pointwise by the interpolated density field.

At each integration point, the elasticity tensor is multiplied by the local density value, resulting in a spatially varying stiffness.

Returns

- **K** (sps.spmatrix): Sparse stiffness matrix of shape (3 * nb_ctrl_pts, 3 * nb_ctrl_pts).

See Also

`IGAPatch.stiffness` : Stiffness assembly without density weighting.
`IGAPatchDensity.density` : Evaluation of the density field at parametric points.

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

Evaluate the interpolated density field at parametric points.

The density is obtained by composition of the evaluated B-spline density field with a clipping operator to the interval [0, 1].

Parameters

- **XI** (list[np.ndarray[np.floating]]): Parametric coordinates [xi, eta, zeta] at which to evaluate the density.

Returns

- **density** (np.ndarray[np.floating]): Density values at the parametric points. Shape corresponds to the tensor-product grid defined by `XI`.

def sigma(
self,
eps: numpy.ndarray[numpy.floating],
density: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]: • View Source

Compute the density-weighted stress tensor in Voigt notation.

The stress is computed from the strain tensor using a linear elastic constitutive law, where the constitutive matrix is scaled pointwise by the provided density field.

Voight notation shear strains are internally converted to tensorial shear strains before applying the constitutive law.

Parameters

- **eps** (np.ndarray[np.floating]): Strain tensor in Voigt notation. Shape: (6, n_param_pts).
- **density** (np.ndarray[np.floating]): Density values at the same parametric points. Shape: (n_param_pts,).

Returns

- **sig** (np.ndarray[np.floating]): Stress tensor in Voigt notation. Shape: (6, n_param_pts).

See Also

`IGAPatch.epsilon` : Strain computation from the displacement field.

def make_paraview_fields(self, U: numpy.ndarray[numpy.floating], XI): • View Source

Generate ParaView fields including density-dependent quantities.

This method extends `IGAPatch.make_paraview_fields()` by adding the interpolated density field and by ensuring that stress-related quantities are computed using the density-weighted constitutive law.

The following fields are exported: - displacement `U` - strain tensor `epsilon` - density field `density` - stress tensor `sigma` - principal stresses `sigma_eig` - von Mises stress `von_mises`

Parameters

- **U** (np.ndarray[np.floating]): Displacement field of shape (3, nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): Parametric coordinates [xi, eta, zeta] defining the evaluation grid.

Returns

- **fields** (dict[str, np.ndarray[np.floating]]): Dictionary of fields formatted for ParaView export.

Inherited Members

IGAPatch spline, ctrl_pts, xi, dxi, eta, deta, zeta, dzeta, F_N, H, jacobian(), grad_N(), make_W(), rhs(), area_border(), epsilon(), sigma_eig(), von_mises(), save_paraview()

IGA_for_bsplyne.ProblemIGA

- View Source

class ProblemIGA: - View Source

`ProblemIGA` class for linear elasticity on 3D multipatch B-spline volumes.

This class assembles the global stiffness matrix and right-hand side vector from multiple `IGAPatch` or `IGAPatchDensity` objects, applies Dirichlet boundary conditions, and solves the linear system to compute the displacement field.

Attributes

- **patches** (list[IGAPatch]): List of patch objects representing the B-spline patches of the problem.
- **connectivity** (MultiPatchBSplineConnectivity): Connectivity information linking the patches to global degrees of freedom.
- **dirichlet** (Dirichlet): Object defining the Dirichlet boundary conditions applied to the system.

Notes

- The class handles multipatch assembly and global degrees of freedom mapping.
- Individual patch computations (stiffness, rhs, fields) are delegated to each patch.
- Post-processing quantities (strain, stress, von Mises) can be exported to ParaView using `save_paraview()`.
- Parallelization is used internally during assembly via `parallel_blocks`.

ProblemIGA(
 patches: list[IGA_for_bsplyne.IGAPatch.IGAPatch],
 connectivity: bsplyne.multi_patch_b_spline.MultiPatchBSplineConnectivity,
 dirichlet: IGA_for_bsplyne.Dirichlet.Dirichlet
) - View Source

Initialize a `ProblemIGA` instance.

This constructor sets up a multipatch linear elasticity problem by storing the patches, the connectivity mapping between them, and the Dirichlet boundary conditions. The object itself does not assemble or solve the system until `lhs_rhs()` or `solve()` is called.

Parameters

- **patches** (list[IGAPatch]): List of patch objects (instances of `IGAPatch` or its subclasses) defining the geometry, material, and local operators.
- **connectivity** (MultiPatchBSplineConnectivity): Connectivity object that maps local patch degrees of freedom to global degrees of freedom for assembly.
- **dirichlet** (Dirichlet): Dirichlet boundary condition object defining the constrained degrees of freedom and their prescribed values.

patches: list[IGA_for_bsplyne.IGAPatch.IGAPatch]

connectivity: bsplyne.multi_patch_b_spline.MultiPatchBSplineConnectivity

dirichlet: IGA_for_bsplyne.Dirichlet.Dirichlet

def lhs_rhs(
 self,
 verbose: bool = **False**,
 disable_parallel: bool = **False**
) -> tuple[scipy.sparse._matrix.spmatrix, numpy.ndarray[numpy.floating]]: - View Source

Assemble the global linear system for the multipatch problem.

This method computes the global left-hand side (LHS) matrix and right-hand side (RHS) vector for the linear elasticity problem represented by the set of patches in the `ProblemIGA`.

The RHS is assembled from the individual patch contributions using the connectivity mapping. The LHS (stiffness matrix) is assembled by computing each patch's local stiffness matrix and mapping its entries to the global system.

Parameters

- **verbose** (bool, optional): If True, prints progress messages during assembly, by default False.
- **disable_parallel** (bool, optional): If True, disables parallel execution for LHS assembly, by default False.

Returns

- **lhs** (sps.spmatrix): The assembled global sparse stiffness matrix of shape (3 * nb_unique_nodes, 3 * nb_unique_nodes).
- **rhs** (np.ndarray[np.floating]): The assembled global right-hand side vector of shape (3 * nb_unique_nodes,).

Notes

- The factor 3 corresponds to the three physical displacement components (x, y, z) per node.
- The patch contributions are gathered and summed according to the connectivity mapping defined in `connectivity`.
- Parallelization is used by default for computing patch LHS and RHS blocks, and can be disabled via `disable_parallel`.

```
def apply_dirichlet(
    self,
    lhs: scipy.sparse._matrix.spmatrix,
    rhs: numpy.ndarray[numpy.floating],
    verbose: bool = False
) -> tuple[scipy.sparse._matrix.spmatrix, numpy.ndarray[numpy.floating]]:
```

View Source

Apply Dirichlet boundary conditions to the global linear system.

This method modifies the global LHS matrix and RHS vector to enforce prescribed Dirichlet conditions. The Dirichlet object (`dirichlet`) provides the selection matrix `C` and prescribed values `k` such that:

```
rhs = C.T @ (rhs - lhs @ k)
lhs = C.T @ lhs @ C
```

Parameters

- **lhs** (sps.spmatrix): Global sparse stiffness matrix of shape (3*nb_unique_nodes, 3*nb_unique_nodes).
- **rhs** (np.ndarray[np.floating]): Global right-hand side vector of shape (3*nb_unique_nodes,).
- **verbose** (bool, optional): If True, prints progress messages during application of Dirichlet conditions, by default False.

Returns

- **lhs** (sps.spmatrix): Reduced stiffness matrix with Dirichlet conditions applied.
- **rhs** (np.ndarray[np.floating]): Reduced RHS vector corresponding to the free degrees of freedom.

Notes

- The operation effectively reduces the system to the unconstrained degrees of freedom.
- Use `solve_from_lhs_rhs()` after applying Dirichlet conditions to obtain the solution.

```
def solve_from_lhs_rhs(
 self,
 lhs: scipy.sparse._matrix.spmatrix,
 rhs: numpy.ndarray[numpy.floating],
 iterative_solve: bool = False,
 verbose: bool = True
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Solve a linear system for the given left-hand side matrix and right-hand side vector.

This method solves the linear system `lhs @ dof = rhs` for the unknown degrees of freedom. It does not apply Dirichlet boundary conditions: the inputs should already be modified accordingly (e.g., via `apply_dirichlet()`).

Parameters

- **lhs** (sps.spmatrix): Sparse matrix representing the system after applying Dirichlet conditions. Can have arbitrary shape as long as it matches `rhs`.
- **rhs** (np.ndarray[np.floating]): Right-hand side vector, already accounting for Dirichlet conditions.
- **iterative_solve** (bool, optional): If True, use an iterative solver (conjugate gradient with diagonal preconditioner), otherwise use a direct solver (Cholesky factorization). Specs for 1_534_278 dof : - CG preconditioned with diagonal (`scipy.sparse.diags(1/lhs.diagonal())`): solved to 1e-5 tol in 131 min - CG preconditioned with AMG (`pyamg.smoothed_aggregation_solver(lhs).aspreconditioner()`): solved to 1e-5 tol in 225 min - Cholesky: solved in 32 min By default False.
- **verbose** (bool, optional): If True, print progress messages during the solving process. Default is True.

Returns

- **dof** (np.ndarray[np.floating]): Solution vector representing the degrees of freedom in the unconstrained/reduced system.

Notes

- The iterative solver prints residuals at each iteration if `verbose=True`.
- For large systems, iterative solvers can be more memory efficient, but may require tuning the preconditioner.
- Use `apply_dirichlet()` to handle Dirichlet constraints before calling this method.

```
def solve(self, iterative_solve=False) -> numpy.ndarray[numpy.floating]:
```

- View Source

Solve the linear elasticity problem for the multipatch IGA system.

This method assembles the global system, applies Dirichlet boundary conditions, and solves for the displacements. The solution returned includes all degrees of freedom in packed notation, with Dirichlet values restored.

Parameters

- **iterative_solve** (bool, optional): If True, solve the system using an iterative solver (preconditioned conjugate gradient), otherwise use a direct solver (Cholesky). Default is False. Exemple notes on performance for large systems (~1.5 million DOFs): - CG with diagonal preconditioner: 1e-5 tol in ~131 min - CG with AMG preconditioner: 1e-5 tol in ~225 min - Cholesky factorization: ~32 min

Returns

- **u** (np.ndarray[np.floating]): Displacement vector including all degrees of freedom, in packed notation. Shape: (3, nb_unique_nodes), with the first dimension corresponding to the physical coordinates (x, y, z).

See Also

`lhs_rhs` : assemble the global system.
`apply_dirichlet` : apply Dirichlet boundary conditions.
`solve_from_lhs_rhs` : solve a reduced linear system.

```
def save_paraview(
 self,
 u: numpy.ndarray[numpy.floating],
 path: str,
 name: str,
 n_eval_per_elem: int = 10,
 disable_parallel: bool = False
):
```

- View Source

Export displacements and derived fields to Paraview files for visualization.

This method separates the global displacement vector into patch-local fields, evaluates additional post-processing fields (strain, stress, von Mises, etc.) at a regular grid of points per element, and saves all data in a format readable by Paraview. The evaluation is done in parallel using `parallel_blocks` unless `disable_parallel` is True.

Parameters

- **u** (np.ndarray[np.floating]): Displacement field in packed notation, including Dirichlet values. Shape: (3, nb_unique_nodes), where the first dimension corresponds to physical coordinates (x, y, z).
- **path** (str): Directory path where the Paraview files will be saved.
- **name** (str): Base name for the Paraview files.
- **n_eval_per_elem** (int, optional): Number of evaluation points per element in each parametric direction. Default is 10.
- **disable_parallel** (bool, optional): If True, disables parallel evaluation of patch fields. Default is False.

Notes

- For each patch, the method evaluates fields at `n_eval_per_elem` points along each parametric direction and computes derived quantities via `IGAPatch.make_paraview_fields`.
- The output can include displacement, strain, stress, von Mises stress, and density if applicable (e.g., in `IGAPatchDensity`).
- The `connectivity` object handles assembling the patches into a global representation for Paraview.

See Also

`IGAPatch.make_paraview_fields`: compute local fields for each patch.
`MultiPatchBSplineConnectivity.save_paraview`: handles saving the assembled global data.

IGA_for_bsplyne.Dirichlet

- View Source

class Dirichlet: - View Source

Affine representation of linear Dirichlet constraints.

This class represents Dirichlet boundary conditions through an affine transformation between a reduced vector of free parameters (`dof`) and the full physical displacement vector (`u`):

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

where:

- `u` is the full displacement vector satisfying all imposed constraints.
- `dof` is the reduced vector of unconstrained degrees of freedom.
- `C` is a sparse matrix whose columns form a basis of admissible variations.
- `k` is a particular displacement vector satisfying the constraints.

This representation allows:

- elimination of constrained DOFs,
- support for general linear multi-point constraints.

Attributes

- **C** (sps.csc_matrix of shape (n_full_dofs, n_free_dofs)): Sparse matrix defining the linear mapping from reduced DOFs to full DOFs.
- **k** (np.ndarray of shape (n_full_dofs,)): Particular solution ensuring that `u` satisfies the Dirichlet constraints.

Dirichlet(C: scipy.sparse._matrix.spmatrix, k: numpy.ndarray[numpy.floating]) - View Source

Initialize an affine Dirichlet constraint representation.

Parameters

- **C** (sps.spmatrix): Sparse matrix of shape (n_full_dofs, n_free_dofs) defining the linear mapping from reduced DOFs to full DOFs.
- **k** (np.ndarray[np.floating]): Vector of shape (n_full_dofs,) defining the affine offset.

Notes

The matrix `C` is internally converted to CSC format for efficient matrix-vector products.

C: scipy.sparse._csc.csc_matrix

k: numpy.ndarray[numpy.floating]

@classmethod
def eye(cls, size: int): - View Source

Create an unconstrained Dirichlet mapping.

This method returns a `Dirichlet` object corresponding to the identity transformation:

```
u = dof
```

i.e. no Dirichlet constraints are applied.

Parameters

- **size** (int): Number of degrees of freedom.

Returns

- **dirichlet** (Dirichlet): Identity `Dirichlet` object with:
  - C = identity(size)
  - k = 0

```
@classmethod
def lock_disp_inds(
 cls,
 inds: numpy.ndarray[numpy.integer],
 k: numpy.ndarray[numpy.floating]
):
```

- View Source

Create Dirichlet constraints by prescribing displacement values at selected DOFs.

This method enforces pointwise Dirichlet conditions of the form:

```
u[i] = k[i]    for i in inds
```

All other DOFs remain unconstrained.

Parameters

- **inds** (np.ndarray[np.integer]): Indices of the displacement vector `u` to be constrained.
- **k** (np.ndarray[np.floating]): Target displacement vector of shape (n_full_dofs,). Only the values at `inds` are enforced.

Returns

- **dirichlet** (Dirichlet): `Dirichlet` instance representing the prescribed displacement constraints.

Notes

Constrained DOFs are removed from the reduced DOF vector. The resulting number of reduced DOFs will therefore be smaller than `k.size`.

```
def set_u_inds_vals(
 self,
 inds: numpy.ndarray[numpy.integer],
 vals: numpy.ndarray[numpy.floating]
):
```

- View Source

Impose pointwise Dirichlet conditions by prescribing displacement values at selected full DOFs.

This method enforces constraints of the form:

```
u[i] = vals[j]     for i = inds[j]
```

by modifying the affine mapping:

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

The modification removes the corresponding admissible variations in `C` and updates `k` so that the prescribed values are always satisfied. As a result, constrained DOFs are eliminated from the reduced DOF vector.

Parameters

- **inds** (np.ndarray[np.integer]): Indices of the full displacement vector `u` to constrain.
- **vals** (np.ndarray[np.floating]): Prescribed displacement values associated with `inds`. Must have the same length as `inds`.

Notes

- Rows of `C` corresponding to constrained DOFs are zeroed.
- Columns of `C` that become unused are removed, which may reduce the number of reduced DOFs.
- The affine offset `k` is updated to enforce the prescribed values.
- This operation modifies the current Dirichlet object in place.

```
def slave_reference_linear_relation(
 self,
 slaves: numpy.ndarray[int],
 references: numpy.ndarray[int],
 coefs: Optional[numpy.ndarray[float]] = None
):
```

- View Source

Impose linear multi-point constraints between full DOFs.

This method enforces relations of the form:

```
u[slave_i] = sum_j coefs[i, j] * u[references[i, j]]
```

by modifying the affine mapping:

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

Slave DOFs are expressed as linear combinations of reference DOFs. The corresponding admissible variations are propagated into `C` and `k`, effectively eliminating slave DOFs from the reduced parameter vector while preserving the imposed constraints.

Parameters

- **slaves** (np.ndarray[int] of shape (n_slaves,)): Indices of DOFs that are constrained (slave DOFs).
- **references** (np.ndarray[int] of shape (n_slaves, n_refs)): Reference DOF indices controlling each slave DOF. Row `i` contains the references associated with `slaves[i]`.
- **coefs** (np.ndarray[float], optional, shape (n_slaves, n_refs)): Linear combination coefficients linking references to slaves. If None, slaves are defined as the average of their references.

Notes

- Slave DOFs are removed from the reduced DOF vector.
- The constraint propagation accounts for hierarchical dependencies between slave DOFs using a topological ordering.
- This operation is typically faster than using `DirichletConstraintHandler`, as it directly modifies the affine mapping without constructing intermediate constraint objects.
- Cyclic dependencies between slaves are not supported and will raise an error.
- This operation modifies the current Dirichlet object in place.

Examples

To impose u[i] = 0.5 * (u[j] + u[k]):

```
>>> slaves = np.array([i])
>>> references = np.array([[j, k]])
>>> coefs = np.array([[0.5, 0.5]])
>>> dirichlet.slave_reference_linear_relation(slaves, references, coefs
)
```

```
def u_du_ddof(
 self,
 dof: numpy.ndarray[numpy.floating]
) -> tuple[numpy.ndarray[numpy.floating], scipy.sparse._csc.csc_matrix]:
```

- View Source

Evaluate the displacement field and its derivative with respect to the reduced DOFs.

The displacement field is obtained from the affine mapping:

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

Since the mapping is affine, the derivative of `u` with respect to `dof` is constant and equal to `C`.

Parameters

- **dof** (np.ndarray[np.floating] of shape (n_dof,)): Reduced degrees of freedom.

Returns

- **u** (np.ndarray[np.floating] of shape (n_u,)): Displacement field.
- **du_ddof** (sps.csc_matrix of shape (n_u, n_dof)): Jacobian of `u` with respect to `dof`: the matrix `C`.

Notes

The returned derivative is independent of `dof` because the mapping is affine.

```
def u(
 self,
 dof: numpy.ndarray[numpy.floating]
) -> numpy.ndarray[numpy.floating]:
```

- View Source

Evaluate the displacement field from reduced DOFs.

The displacement field is computed using the affine mapping:

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

Parameters

- **dof** (np.ndarray[np.floating]): Reduced degrees of freedom. Can be either:
  - shape (n_dof,)
  - shape (..., n_dof) for batched evaluation

Returns

- **u** (np.ndarray[np.floating]): Displacement field with shape:
  - (n_u,) if `dof` is 1D
  - (..., n_u) if `dof` is batched

Notes

This method supports vectorized evaluation over multiple DOF vectors.

```
def dof_lsq(self, u: numpy.ndarray[numpy.floating]) -> numpy.ndarray[numpy.floating]:
```

- View Source

Compute reduced DOFs from a displacement field using a least-squares projection.

This method computes `dof` such that:

```
u ≈ C @ dof + k
```

by solving the normal equations:

```
(Cᵀ C) dof = Cᵀ (u - k)
```

Parameters

- **u** (np.ndarray[np.floating]): Displacement field. Can be either:
  - shape (n_u,)

- shape (..., n_u) for batched projection

Returns

- **dof** (np.ndarray[np.floating]): Reduced degrees of freedom with shape:
  - (n_dof,) if `u` is 1D
  - (..., n_dof) if `u` is batched

Notes

This operation performs a least-squares inversion of the affine mapping. If `C` does not have full column rank, the solution corresponds to a minimum-norm least-squares solution.

The system `(C^T C)` is solved using a sparse linear solver.

```
@nb.njit(cache=True)
def slave_reference_linear_relation_sort(
 slaves: numpy.ndarray[int],
 references: numpy.ndarray[int]
) -> numpy.ndarray[int]:
```

- View Source

Sorts the slave nodes based on reference indices to respect hierarchical dependencies (each slave is processed after its references).

Parameters

- **slaves** (np.ndarray[int]): Array of slave indices.
- **references** (np.ndarray[int]): 2D array where each row contains the reference indices controlling a slave.

Returns

- **sorted_slaves** (np.ndarray[int]): Array of slave indices sorted based on dependencies.

```
@nb.njit(cache=True)
def slave_reference_linear_relation_inner(
 indices: numpy.ndarray[int],
 indptr: numpy.ndarray[int],
 data: numpy.ndarray[float],
 k: numpy.ndarray[float],
 slaves: numpy.ndarray[int],
 references: numpy.ndarray[int],
 coefs: numpy.ndarray[float],
 sorted_slaves: numpy.ndarray[int]
) -> tuple[numpy.ndarray[int], numpy.ndarray[int], numpy.ndarray[float], numpy.ndarray[float]]:
```

- View Source

Applies slave-reference relations directly to CSR matrix arrays.

Parameters

- **indices** (np.ndarray[int]): Column indices of CSR matrix.
- **indptr** (np.ndarray[int]): Row pointers of CSR matrix.
- **data** (np.ndarray[float]): Nonzero values of CSR matrix.
- **k** (np.ndarray[float]): Vector to be updated.
- **slaves** (np.ndarray[int]): Array of slave indices.
- **references** (np.ndarray[int]): 2D array where each row contains the reference indices controlling a slave.
- **coefs** (np.ndarray[float]): 2D array of coefficients defining the linear relationship between references and slaves.
- **sorted_slaves** (np.ndarray[int]): Array of slave indices sorted in topological order.

Returns

- **rows** (np.ndarray[int]): Updated row indices of COO matrix.

- **cols** (np.ndarray[int]): Updated column indices of COO matrix.
- **data** (np.ndarray[float]): Updated nonzero values of COO matrix.
- **k** (np.ndarray[float]): Updated vector.

class DirichletConstraintHandler: • View Source

Accumulate affine linear constraints and construct the associated Dirichlet mapping.

This class is designed to impose linear affine constraints of the form:

```
D @ u = c
```

where `u` is the full displacement (or state) vector. Constraints are progressively accumulated and, once fully specified, converted into an affine parametrization of the admissible solution space:

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

where:

- `C` is a basis of the nullspace of `D` (i.e. `D @ C = 0`) obtained by QR decomposition,
- `k` is a particular solution satisfying the constraints, obtained through the QR decomposition which helps solve:

  D k = c

- `dof` is a reduced vector of unconstrained degrees of freedom.

The main purpose of this class is to provide a flexible and robust interface to define constraints before constructing a `Dirichlet` object representing the reduced parametrization.

Typical workflow

1. Create a handler with the number of physical DOFs (`u`).
2. Add constraint equations using helper methods.
3. **Ensure that all reference DOFs (e.g., translations or rotations introduced for rigid-body relations) are fully constrained before computing `C` and `k`.**
4. Build the reduced representation `(C, k)` or directly create a `Dirichlet` object.

This separation allows constraints to be assembled incrementally and validated before generating the final affine mapping.

Attributes

- **nb_dofs_init** (int): Number of DOFs in the original unconstrained system (physical DOFs `u`).
- **lhs** (sps.spmatrix): Accumulated constraint matrix `D`.
- **rhs** (np.ndarray[np.floating]): Accumulated right-hand side vector `c`.

Notes

- The constraint system may include additional reference DOFs introduced to express kinematic relations or rigid-body behaviors.
- **All reference DOFs must be fully constrained before computing `C` and `k`;** otherwise, DOFs that lie in the kernel of `D` cannot be controlled and imposed values (e.g., prescribed translations) may not appear in the resulting solution `k`.
- The resulting affine mapping guarantees that any generated vector `u` satisfies all imposed constraints.

DirichletConstraintHandler(nb_dofs_init: int) • View Source

Initialize a Dirichlet constraint handler.

Parameters

- **nb_dofs_init** (int): The number of initial degrees of freedom in the unconstrained system. This value is used to size the initial constraint matrix and manage later extensions with reference DOFs.

nb_dofs_init: int

lhs: scipy.sparse._matrix.spmatrix

rhs: numpy.ndarray[numpy.floating]

def copy(self) -> DirichletConstraintHandler: • View Source

Create a deep copy of this DirichletConstraintHandler instance.

Returns

- **DirichletConstraintHandler**: A new instance with the same initial number of DOFs, constraint matrix, and right-hand side vector. All internal data is copied, so modifications to the returned handler do not affect the original.

def add_eqs(
self,
lhs: scipy.sparse._matrix.spmatrix,
rhs: numpy.ndarray[numpy.floating]
): • View Source

Append linear affine constraint equations to the global constraint system.

Adds equations of the form:

```
lhs @ u = rhs
```

to the accumulated constraint system `D @ u = c`.

If `lhs` is expressed only in terms of the initial physical DOFs (`nb_dofs_init` columns), it is automatically extended with zero-padding to match the current number of DOFs (e.g. after reference DOFs have been added).

Parameters

- **lhs** (sps.spmatrix of shape (n_eqs, n_dofs)): Constraint matrix defining the left-hand side of the equations. The number of columns must match either:
  - the initial number of DOFs (`nb_dofs_init`), or
  - the current number of DOFs in the handler.
- **rhs** (np.ndarray[np.floating] of shape (n_eqs,)): Right-hand side values associated with the constraint equations.

Raises

- **ValueError**: If the number of columns of `lhs` is incompatible with the current constraint system.

Notes

Constraints are appended to the existing system and are not simplified or checked for redundancy.

def add_ref_dofs(self, nb_dofs: int): • View Source

Append additional reference DOFs to the constraint system.

This method increases the size of the global DOF vector by adding `nb_dofs` new reference DOFs. These DOFs are introduced without imposing any constraint, i.e. they appear with zero coefficients in all existing equations.

Parameters

- **nb_dofs** (int): Number of reference DOFs to append to the global DOF vector.

Notes

Reference DOFs are typically used to express kinematic relations, rigid body motions, or other auxiliary constraint parametrizations.

def add_ref_dofs_with_behavior(
self,
behavior_mat: scipy.sparse._matrix.spmatrix,
slave_inds: numpy.ndarray[numpy.integer]
):

- View Source

Introduce reference DOFs and constrain slave DOFs through a linear relation.

This method appends new reference DOFs and enforces a linear behavioral relation linking these reference DOFs to existing DOFs (called *slave DOFs*). The imposed constraints take the form:

```
behavior_mat @ ref_dofs - u[slave_inds] = 0
```

Parameters

- **behavior_mat** (sps.spmatrix of shape (n_slaves, n_ref_dofs)): Linear operator defining how each reference DOF contributes to the corresponding slave DOFs.
- **slave_inds** (np.ndarray[np.integer] of shape (n_slaves,)): Indices of slave DOFs that are controlled by the reference DOFs. The ordering must match the rows of `behavior_mat`.

Raises

- **AssertionError**: If the number of slave DOFs is inconsistent with the number of rows of `behavior_mat`.

Notes

This method first adds the reference DOFs to the global system and then appends the corresponding constraint equations.

def add_rigid_body_constraint(
self,
ref_point: numpy.ndarray[numpy.floating],
slaves_inds: numpy.ndarray[numpy.integer],
slaves_positions: numpy.ndarray[numpy.floating]
):

- View Source

Constrain slave nodes to follow a rigid body motion defined by a reference point.

This method introduces six reference DOFs representing a rigid body motion: three rotations and three translations (in this order) around a reference point. The displacement of each slave node is constrained to follow the rigid body motion:

```
u(X) = θ × (X - X_ref) + t
```

where `θ` is the rotation vector and `t` is the translation vector.

Parameters

- **ref_point** (np.ndarray[np.floating] of shape (3,)): Reference point defining the center of rotation.
- **slaves_inds** (np.ndarray[np.integer] of shape (3, n_nodes)): DOF indices of the slave nodes. Each column contains the x, y, z DOF indices of a slave node.
- **slaves_positions** (np.ndarray[np.floating] of shape (3, n_nodes)): Physical coordinates of the slave nodes.

Notes

- Six reference DOFs (θx, θy, θz, tx, ty, tz) are added to represent the rigid body motion.

- The constraint is expressed as a linear relation between the reference DOFs and the slave displacements.
- This method is commonly used to impose rigid connections, master node formulations, or reference frame constraints.

```
def add_eqs_from_inds_vals(
 self,
 inds: numpy.ndarray[numpy.integer],
 vals: numpy.ndarray[numpy.floating] = None
):
```

- View Source

Impose pointwise Dirichlet conditions on selected DOFs.

This is a convenience method that adds constraint equations of the form:

```
u[inds] = vals
```

Parameters

- **inds** (np.ndarray[np.integer] of shape (n_eqs,)): Indices of DOFs to constrain.
- **vals** (np.ndarray[np.floating] of shape (n_eqs,), optional): Prescribed values associated with each constrained DOF. If None, zero values are imposed.

Raises

- **AssertionError**: If `vals` is provided and its size differs from `inds`.

Notes

This method is equivalent to adding rows of the identity matrix to the constraint operator.

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

- View Source

Construct the affine transformation (C, k) enforcing all Dirichlet constraints.

This method solves the linear constraint system:

```
D @ u = c
```

by computing:

- a matrix `C` forming a basis of the nullspace of `D` (i.e., D @ C = 0),
- a particular solution `k` such that D @ k = c.

Any admissible vector `u` satisfying the constraints can then be written as:

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

where `dof` is the reduced vector of unconstrained degrees of freedom.

The nullspace basis and the particular solution are obtained through a sparse QR factorization of D^T.

Returns

- **C** (sps.spmatrix): Sparse matrix of shape (n_full_dofs, n_free_dofs) whose columns form a basis of the nullspace of `D`.
- **k** (np.ndarray[np.floating]): Vector of shape (n_full_dofs,) representing a particular solution satisfying the constraints.

Notes

- The transformation (C, k) defines an affine parametrization of the admissible displacement space.

- The reduced DOF vector `dof` corresponds to the coordinates of `u` in the nullspace basis.
- The QR factorization is performed on D^T to efficiently extract both the nullspace basis and the particular solution.

def get_reduced_Ck(self) -> tuple[scipy.sparse._matrix.spmatrix, numpy.ndarray]: • View Source

Extract the affine constraint transformation restricted to physical DOFs.

This method computes the full transformation (C, k) using `self.make_C_k()` and returns only the rows associated with the initial (physical) degrees of freedom. The resulting pair (C_u, k_u) defines:

```
u_phys = C_u @ dof + k_u
```

where `u_phys` satisfies all imposed Dirichlet constraints.

Returns

- **C_u** (sps.spmatrix): Reduced nullspace basis matrix of shape (nb_dofs_init, n_free_dofs).
- **k_u** (np.ndarray): Reduced particular solution vector of shape (nb_dofs_init,).

Notes

- The full transformation (C, k) may include auxiliary reference DOFs introduced by multi-point or hierarchical constraints.
- Only the rows corresponding to physical DOFs are returned.
- A warning is emitted if reference DOFs remain unconstrained, which may indicate missing boundary specifications.

def create_dirichlet(self): • View Source

Build a `Dirichlet` object representing the reduced constraint transformation.

This is a convenience wrapper around `get_reduced_Ck()` that constructs a `Dirichlet` object directly from the reduced affine mapping:

```
u_phys = C_u @ dof + k_u
```

Returns

- **dirichlet** (Dirichlet): A `Dirichlet` instance encapsulating the reduced transformation matrices.

def get_ref_multipliers_from_internal_residual(self, K_u_minus_f): • View Source

Recover Lagrange multipliers associated with reference DOF constraints.

This method reconstructs the Lagrange multipliers λ corresponding to reference DOFs using the internal residual vector of the mechanical system:

```
r = K @ u - f
```

The derivation relies on the partition of the transformation matrix:

```
C = [C_u;
     C_ref
]
```

where `C_u` maps reduced DOFs to physical DOFs and `C_ref` maps them to reference DOFs.

The multipliers satisfy:

```
C_ref.T @ λ = - C_u.T @ r
```

and are obtained via the least-squares solution:

$$\lambda = -(C_{ref} @ C_{ref}.T)^{-1} @ C_{ref} @ C_u.T @ r$$

Parameters

- **K_u_minus_f** (np.ndarray): Internal residual vector of shape (nb_dofs_init,), corresponding to K @ u - f restricted to physical DOFs.

Returns

- **lamb** (np.ndarray): Vector of Lagrange multipliers associated with reference DOF constraints.

Notes

- The multipliers correspond to reaction forces or generalized constraint forces transmitted through reference DOFs.
- The residual must be assembled consistently with the stiffness matrix and load vector used to compute the displacement field.
- The matrix C_ref @ C_ref.T is assumed to be invertible, which holds when reference constraints are linearly independent.

IGA_for_bsplyne.fallback_sparse_qr

- View Source

```
@nb.njit
def get_givens_params(a, b):
```

- View Source

Computes the Givens rotation parameters (c, s) such that the rotation matrix applied to [a, b]^T results in [r, 0]^T.

```
@nb.njit
def apply_rotation_sparse(
 idx_j,
 dat_j,
 idx_i,
 dat_i,
 c,
 s,
 start_col,
 end_col,
 w_j,
 w_i,
 active
):
```

- View Source

Applies a Givens rotation to two sparse rows.

Parameters:

- idx_j, dat_j: Indices and data of the j-th row.
- idx_i, dat_i: Indices and data of the i-th row.
- c, s: Rotation parameters.
- start_col, end_col: Column range to apply the rotation.
- w_j, w_i: Pre-allocated dense work vectors for processing.
- active: Pre-allocated boolean mask to track non-zero entries.

```
@nb.njit
def to_csr(rows_idx, rows_dat, n_rows):
```

- View Source

Efficiently converts typed lists of sparse rows into a flat CSR structure. Uses pre-allocation to ensure O(nnz) performance.

```
@nb.njit(nb.types.UniTuple(nb.types.Tuple((nb.types.float64[::1], nb.types.int32[::1], nb.types.int32[::1])), 2)
(nb.types.float64[:], nb.types.int32[:], nb.types.int32[:], nb.types.UniTuple(nb.types.int64, 2)), cache=True)
def sparse_qr_numba(data, indices, indptr, shape):
```

- View Source

Core Sparse QR decomposition using Givens rotations.

Iterates through columns to eliminate sub-diagonal elements while maintaining a sparse representation using Numba typed lists. Constructs Q^T (stored in Q_idx/Q_dat) and R.

```
def my_qr_sparse(A):
```

- View Source

Computes the Sparse QR decomposition of matrix A. Uses column reordering to minimize fill-in.

Returns:

- Q: Orthogonal matrix (m x m)

- R_perm: Upper triangular matrix (m x n)
- permutation: Column permutation array
- rank: Numerical rank of the matrix

IGA_for_bsplyne.solvers

- View Source

def solve_sparse(A, b): - View Source

Solve Ax=b using Cholmod if available, otherwise fallback to spsolve.

def qr_sparse(A): - View Source

Compute QR decomposition using sparseqr if available, otherwise fallback to my_qr_sparse.