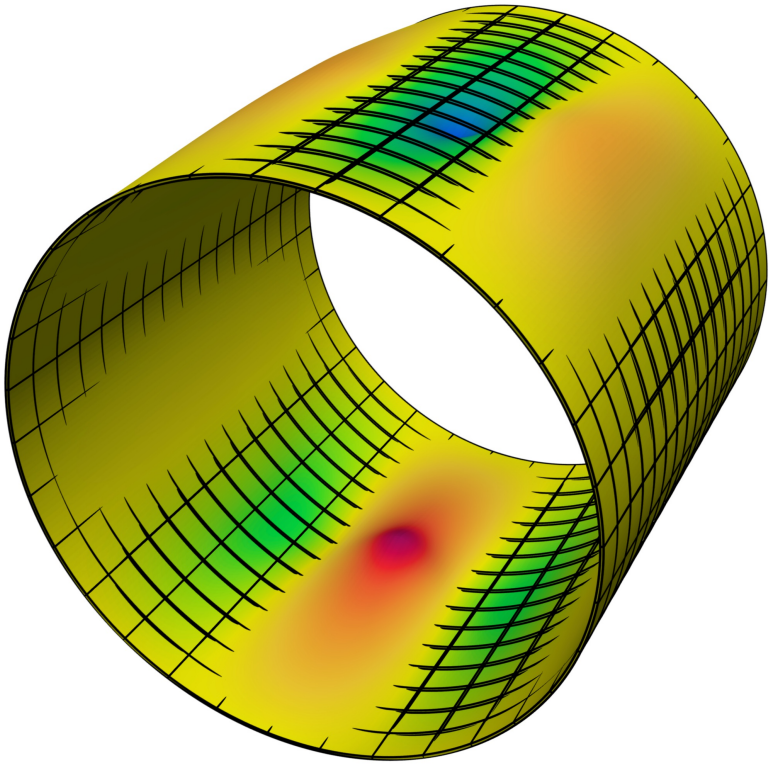
IGA_for_bsplyne

IGA_for_bsplyne



IGA_for_bsplyne is a Python library for 3D isogeometric analysis (IGA) applied to elasticity problems. This library builds upon the previous work from the bsplyne library and extends its capabilities for complex elasticity simulations.

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

Installation

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

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

Make sure to also install the required dependency bsplyne manually:

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

Additionally, ensure that scikit-sparse is installed (recommended installation via conda):

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

Main Modules

- **Dirichlet**
  Manages Dirichlet boundary conditions by applying an affine mapping (u = C @ dof + k).
  Key functions: `Dirichlet.eye()`, `Dirichlet.lock_disp_inds()`, `set_u_inds_vals()`, `u_du_ddof()`, `u()`, `dof_lsq()`

- **IGAPatch**

Constructs a 3D IGA patch to compute the stiffness matrix, right-hand side vector, and other operators for elasticity problems over a B-spline volume.
Key functions: `jacobian()`, `grad_N()`, `make_W()`, `stiffness()`, `rhs()`, `epsilon()`, `sigma()`, `sigma_eig()`, `von_mises()`, `save_paraview()`

- **ProblemIGA**
  Assembles the global system of equations, applies boundary conditions, and solves the elasticity problem across one or more patches.
  Key functions: `assembly_block()`, `lhs_rhs()`, `apply_dirichlet()`, `solve_from_lhs_rhs()`, `solve()`, `save_paraview()`

Examples

Several example scripts demonstrating the usage of **IGA_for_bsplyne** can be found in the `examples/` directory. These scripts cover different aspects of the library, including setting up boundary conditions, creating IGA patches, and solving elasticity problems.

Documentation

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

Contributing

Contributions are welcome!

- To report bugs or suggest improvements, please open an issue.
- For direct contributions, feel free to fork the repository and submit pull requests.

License

- View Source

IGA_for_bsplyne.IGAPatch

- View Source

class IGAPatch:

- View Source

IGAPatch class to compute linear elasticity operators on 3D B-spline volumes. This class computes the stiffness matrix and the right hand side on one B-spline patch.

Attributes

- **spline** (BSpline): B-spline volume object used as the patch. Contains the methods to compte the shape functions.
- **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.
- **xi** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate xi.
- **dxi** (np.ndarray[np.floating]): Corresponding weights of xi.
- **eta** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate eta.
- **deta** (np.ndarray[np.floating]): Corresponding weights of eta.
- **zeta** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate zeta.
- **dzeta** (np.ndarray[np.floating]): Corresponding weights of zeta.
- **F_N** (np.ndarray[np.floating]): Surfacic forces applied on the corresponding side of the patch.

```
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 the IGAPatch with the given parameters.

Parameters

- **spline** (BSpline): B-spline volume used as the patch.
- **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.
- **F_N** (np.ndarray[np.floating], optional): Surfacic forces applied on the corresponding side of the patch. Its shape should be (3(param), 2(side), 3(phy)). By default np.zeros((3, 2, 3), dtype='float').

spline: bsplyne.b_spline.BSpline

ctrl_pts: numpy.ndarray[numpy.floating]

E: float

nu: float

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

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

Calculate the Jacobian matrix (derivative of the mesh wrt its isoparametric space), its inverse, and its determinant.

Parameters

- **dN_dXI** (tuple[sps.spmatrix, sps.spmatrix, sps.spmatrix]): Tuple of sparse matrices representing the derivatives of shape functions wrt the isoparametric space. Contains dN_dxi, dN_deta and dN_dzeta.

Returns

- **J** (np.ndarray[np.floating]): Jacobian matrix, with shape (3(phy), 3(param), nb_intg_pts).
- **Jinv** (np.ndarray[np.floating]): Inverse of the Jacobian matrix, with shape (3(param), 3(phy), nb_intg_pts).
- **detJ** (np.ndarray[np.floating]): Determinant of the Jacobian matrix, with shape (nb_intg_pts,).

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

Calculate the gradient of shape functions with respect to physical space.

Parameters

- **Jinv** (np.ndarray[np.floating]): Inverse of the Jacobian matrix, with shape (3(phy), 3(param), nb_intg_pts).
- **dN_dXI** (tuple[sps.spmatrix, sps.spmatrix, sps.spmatrix]): Tuple of sparse matrices representing the derivatives of shape functions with respect to the isoparametric space.

Returns

- **dN_dX** (np.ndarray[np.object_]): Gradient of shape functions with respect to physical space. Numpy array of shape (3(phy),) containing `sps.spmatrix` objects of shape (nb_intg_pts, nb_ctrl_pts).

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

- View Source

Compute the measure for intagrating in the physical space as (abs(det(dX_dXI))*dxi*deta*dzeta). This is the element-wise product of the absolute value of detJ and the Kronecker product of dxi, deta, and dzeta.

Parameters

- **detJ** (np.ndarray[np.floating]): Array containing the determinant values. Its shape is (nb_intg_pts,).

Returns

- **W** (np.ndarray[np.floating]): Array containing the measure for intagrating in the physical space. Its shape is (nb_intg_pts,).

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

Calculate the stiffness matrix for the IGAPatch.

Returns

- **K** (sps.spmatrix): Stiffness matrix computed based on the given parameters and operations.

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

Calculate the right-hand side (rhs) vector for the IGAPatch.

Returns

- **rhs** (np.ndarray[np.floating]): The computed rhs vector based on the given parameters and operations.

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

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

Calculate the strain tensor (epsilon) for the IGAPatch based on the displacement field `U` and the isoparametric coordinates `XI`.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

Returns

- **eps** (np.ndarray[np.floating]): Strain tensor epsilon in voight notation computed as a numpy array of shape (6, nb_param_pts).

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

Calculate the stress tensor (sigma) for the IGAPatch based on the displacement field `U` and the isoparametric coordinates `XI`.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

Returns

- **sig** (np.ndarray[np.floating]): Stress tensor sigma in voight notation computed as a numpy array of shape (6, nb_param_pts).

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

Calculate the eigenvalues of the stress tensor for the IGAPatch based on the displacement field `U` and the isoparametric coordinates `XI`.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

Returns

- **sig_eig** (np.ndarray[np.floating]): Eigenvalues of the stress tensor computed as a numpy array of shape (nb_param_pts, 3).

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

Calculate the von Mises stress for the IGAPatch based on the displacement field `U` and the isoparametric coordinates `XI`.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

Returns

- **vm** (np.ndarray[np.floating]): Von Mises stress computed as a numpy array of shape (nb_param_pts,).

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

Save data for visualization in ParaView.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **path** (str): Path to save the data.
- **name** (str): Name of the saved data.
- **n_eval_per_elem** (int, optional): Number of evaluations per element, by default 10.

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

Make data fields for visualization in ParaView.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).

def compute_epsilon(patch_obj, u_patch, spline, XI): • View Source

def compute_sigma(patch_obj, u_patch, spline, XI): • View Source

def compute_sigma_eig(patch_obj, u_patch, spline, XI): • View Source

def compute_von_mises(patch_obj, u_patch, spline, XI): • View Source

class IGAPatchDensity(IGAPatch): • View Source

IGAPatch class to compute linear elasticity operators on 3D B-spline volumes. This class computes the stiffness matrix and the right hand side on one B-spline patch.

Attributes

- **spline** (BSpline): B-spline volume object used as the patch. Contains the methods to compte the shape functions.
- **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.
- **xi** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate xi.
- **dxi** (np.ndarray[np.floating]): Corresponding weights of xi.
- **eta** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate eta.
- **deta** (np.ndarray[np.floating]): Corresponding weights of eta.
- **zeta** (np.ndarray[np.floating]): Discretization of the isoparametric coordinate zeta.
- **dzeta** (np.ndarray[np.floating]): Corresponding weights of zeta.
- **F_N** (np.ndarray[np.floating]): Surfacic forces applied on the corresponding side of the patch.

```
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 the IGAPatch with the given parameters.

Parameters

- **spline** (BSpline): B-spline volume used as the patch.
- **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.
- **F_N** (np.ndarray[np.floating], optional): Surfacic forces applied on the corresponding side of the patch. Its shape should be (3(param), 2(side), 3(phy)). By default np.zeros((3, 2, 3), dtype='float').

d

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

Calculate the stiffness matrix for the IGAPatch.

Returns

- **K** (sps.spmatrix): Stiffness matrix computed based on the given parameters and operations.

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

Calculate the stress tensor (sigma) for the IGAPatch based on the displacement field `U` and the isoparametric coordinates `XI`.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).
- **XI** (list[np.ndarray[np.floating]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

Returns

- **sig** (np.ndarray[np.floating]): Stress tensor sigma in voight notation computed as a numpy array of shape (6, nb_param_pts).

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

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

Make data fields for visualization in ParaView.

Parameters

- **U** (np.ndarray[np.floating]): Displacement field as a numpy array of shape (3(phy), nb_ctrl_pts).

Inherited Members

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

def compute_density(patch_obj, spline, XI): • View Source

IGA_for_bsplyne.ProblemIGA

- View Source

class ProblemIGA: - View Source

ProblemIGA class to compute linear elasticity on 3D multipatch B-spline volumes. This class computes the stiffness matrix and the right hand side and solves the linear problem.

Attributes

- **patches** (list[IGAPatch]): List of IGAPatch objects representing the patches for the ProblemIGA.
- **connectivity** (MultiPatchBSplineConnectivity): MultiPatchBSplineConnectivity object defining the connectivity information.
- **dirichlet** (Dirichlet): Dirichlet object specifying the Dirichlet boundary conditions.

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

Initialize the ProblemIGA class with the provided patches, connectivity, and dirichlet.

Parameters

- **patches** (list[IGAPatch]): List of IGAPatch objects representing the patches for the ProblemIGA.
- **connectivity** (MultiPatchBSplineConnectivity): MultiPatchBSplineConnectivity object defining the connectivity information.
- **dirichlet** (Dirichlet): Dirichlet object specifying the Dirichlet boundary conditions.

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 left-hand side (lhs) matrix and right-hand side (rhs) vector for the linear system of equations.

Parameters

- **verbose** (bool, optional): If True, prints progress messages during the assembly process, by default False.
- **disable_parallel** (bool, optional): Wether to disable parallel exectution. By default, False.

Returns

- **lhs, rhs** (tuple[sps.spmatrix, np.ndarray[np.floating]]): The assembled sparse left-hand side matrix and right-hand side vector.

```
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 system of equations.

Parameters

- **lhs** (sps.spmatrix): The left-hand side sparse matrix of the system.
- **rhs** (np.ndarray[np.floating]): The right-hand side vector of the system.
- **verbose** (bool, optional): If True, prints progress messages, by default False.

Returns

- **lhs, rhs** (tuple[sps.spmatrix, np.ndarray[np.floating]]): The modified left-hand side matrix and right-hand side vector after applying Dirichlet boundary conditions.

```
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 the linear system defined by the left-hand side (lhs) matrix and right-hand side (rhs) vector.

Parameters

- **lhs** (sps.spmatrix): The left-hand side sparse matrix of the system.
- **rhs** (np.ndarray[np.floating]): The right-hand side vector of the system.
- **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, by default True.

Returns

- **dof** (np.ndarray[np.floating]): The solution vector representing the degrees of freedom.

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

- View Source

Solve the linear system for the ProblemIGA class using ProcessPoolExecutor with block splitting.

Parameters

- **iterative_solve** (bool, optional): Whether to use an iterative solver or not, default is False. The iterative solver is a sparse conjugate gradient with a diagonal preconditioner. The direct solver is a Cholesky sparse solver. 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

Returns

- **u** (np.ndarray[np.floating]): Solution vector representing the computed displacements in packed notation. Shape : (3(phy), nb_unique_nodes)

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

- View Source

Save the computed displacements and related fields to Paraview format for visualization.

Parameters

- **u** (np.ndarray[np.floating]): Displacement field in packed notation as a numpy array of shape (3(phy), nb_unique_nodes).
- **path** (str): Path to save the Paraview files.
- **name** (str): Name of the Paraview files.
- **n_eval_per_elem** (int, optional): Number of evaluations per element, default is 10.

IGA_for_bsplyne.Dirichlet

- View Source

class Dirichlet: - View Source

A class to handle Dirichlet boundary conditions (BC) for a problem using an affine mapping.

The Dirichlet class provides methods to apply Dirichlet BCs by mapping degrees of freedom (dof) to displacements (u) using the relation `u = C @ dof + k`. It supports creating instances with identity mappings, locking specific displacement indices, and computing dof from displacements via least squares approximation.

Attributes

- **C** (sps.csc_matrix): The matrix used in the affine mapping from `dof` to `u`.
- **k** (np.ndarray[np.floating]): The vector used in the affine mapping from `dof` to `u`.

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

Initializes a Dirichlet instance with the given matrix and vector for affine mapping.

Parameters

- **C** (sps.spmatrix of float): The matrix used to map degrees of freedom `dof` to displacements `u`.
- **k** (np.ndarray[np.floating]): The vector used to map degrees of freedom `dof` to displacements `u`.

C: scipy.sparse._csc.csc_matrix

k: numpy.ndarray[numpy.floating]

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

Create a `Dirichlet` instance with an identity mapping, where no degrees of freedom `dof` are locked. Sets `C` to the identity matrix and `k` to a zero-filled vector.

Parameters

- **size** (int): Size of the `dof` and `u` vectors.

Returns

- **dirichlet** (Dirichlet): The identity `Dirichlet` instance.

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

Creates a `Dirichlet` instance with specified displacement `u` indices locked to given values.

Parameters

- **inds** (np.ndarray[np.integer]): Indices of the displacement field `u` to be locked.
- **k** (np.ndarray[np.floating]): Values to lock the specified indices of `u` to. After this, `u[inds]` are set to `k[inds]` while other compontents are left free.

Returns

- **dirichlet** (Dirichlet): A `Dirichlet` instance with specified displacements locked.

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

Locks specified indices of the displacement field `u` to given values by modifying the matrix `C` and vector `k` accordingly. This involves zeroing out the specified rows in `C` and adjusting `k` to reflect the locked values.

Parameters

- **inds** (np.ndarray[np.integer]): Indices of the displacement field `u` to be locked.
- **vals** (np.ndarray[np.floating]): Values to lock the specified indices of `u` to.

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

This function modifies the sparse matrix `C` and the vector `k` to enforce reference-slave constraints in an optimization problem. The goal is to eliminate the degrees of freedom (DOFs) associated with slave nodes, keeping only the reference DOFs. The relation `u = C@dof + k` is updated so that slave DOFs are expressed as linear combinations of reference DOFs, reducing the problem's size while maintaining the imposed constraints.

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.
- **coefs** (Union[np.ndarray[float], None], optional): 2D array of coefficients defining the linear relationship between references and slaves. If None, the coefficients are set so that the slaves are the average of the references. By default None.

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

Computes the displacement field `u` and its derivative with respect to the degrees of freedom `dof`. The displacement field is calculated as `u = C @ dof + k`, and its derivative is `C`.

Parameters

- **dof** (np.ndarray[np.floating]): The degrees of freedom of the problem, representing the input to the affine mapping.

Returns

- **u, du_ddof** (tuple[np.ndarray[np.floating], sps.csc_matrix]): A tuple containing the displacement field `u` and its derivative with respect to `dof`.

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

Computes the displacement field `u` using the affine mapping `u = C @ dof + k`.

Parameters

- **dof** (np.ndarray[np.floating]): The degrees of freedom of the problem, representing the input to the affine mapping.

Returns

- **u** (np.ndarray[np.floating]): The computed displacement field `u`.

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

Computes the degrees of freedom `dof` from the displacement field `u` using a least squares approximation. This method performs a least squares 'inversion' of the affine mapping `u = C @ dof + k`. It solves the linear problem `C.T @ C @ dof = C.T @ (u - k)` for `dof`.

Parameters

- **u** (np.ndarray[np.floating]): The displacement field from which to compute the degrees of freedom.

Returns

- **dof** (np.ndarray[np.floating]): The computed degrees of freedom corresponding to the given displacement field.

@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

Manage linear Dirichlet constraints for variational or optimization problems.

This class accumulates linear equations of the form `D @ u = c` representing Dirichlet boundary conditions or linear relationships between degrees of freedom (DOFs), and computes a reduced basis representation that parametrizes the set of admissible solutions.

Specifically, it computes matrices `C` and `k` such that any vector `u` satisfying `D @ u = c` can be written as:

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

where `dof` is a reduced vector of free parameters.

Attributes

- **nb_dofs_init** (int): Number of DOFs in the original unconstrained system, before adding any reference DOFs.
- **lhs** (sps.spmatrix): Accumulated constraint matrix D (left-hand side of the Dirichlet conditions).
- **rhs** (np.ndarray[np.floating]): Accumulated right-hand side vector c of the Dirichlet conditions.

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

Add linear constraint equations of the form D_new @ u = c_new.

Appends the given equations to the existing Dirichlet constraint system. If the number of columns in `lhs` matches the initial DOF count, it is automatically extended with zero-padding to match the current DOF count (in case reference DOFs have been added).

Parameters

- **lhs** (sps.spmatrix): Constraint matrix D_new of shape (n_eqs, nb_dofs) to be added to the system.
- **rhs** (np.ndarray[np.floating]): Right-hand side values c_new of shape (n_eqs,) corresponding to the constraint.

Raises

- **ValueError**: If the number of columns in `lhs` does not match the initial or current DOF count.

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

Extend the system by adding new reference DOFs.

This increases the number of columns in the constraint matrix by `nb_dofs`, initializing them with zeros in all existing constraint equations.

Parameters

- **nb_dofs** (int): Number of new reference DOFs to append at the end of the current DOF vector.

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

Add new reference DOFs and define their influence on existing DOFs via a behavioral relation.

This method appends new reference DOFs and enforces their relationship to existing DOFs (called "slaves") through a linear behavior matrix `behavior_mat`. The resulting constraints take the form:

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

and are added to the global constraint system.

Parameters

- **behavior_mat** (sps.spmatrix): Matrix of shape `(n_slaves, n_ref_dofs)` defining how each reference DOF influences the corresponding slave DOFs.
- **slave_inds** (np.ndarray[np.integer]): Indices of the slave DOFs. Must have length `n_slaves` and must be in the same order as the rows of `behavior_mat`.

Raises

- **AssertionError**: If the number of rows in `behavior_mat` does not match the size of `slave_inds`.

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

Add a reference node and impose a rigid body motion constraint on a set of slave nodes.

This method introduces new reference degrees of freedom (DOFs) corresponding to a reference node located at `ref_point`, and constrains the displacements of a set of slave nodes (given by `nodes_inds` and `nodes_positions`) to follow a rigid body motion defined by the reference node. The rigid body motion includes both translation and rotation about the reference point.

The imposed constraint ensures that the displacement of each slave node is a linear combination of the reference node's translation and rotation, enforcing a rigid connection between the reference and the slaves.

Parameters

- **ref_point** (np.ndarray[np.floating]): Reference point (origin for rotation and translation), array of shape (3,) representing (x, y, z).
- **slaves_inds** (np.ndarray[np.integer]): Indices of the degrees of freedom of the slave displacements. Shape (3, n), where n is the number of slave nodes; each column contains the x, y, z DOF indices for a node.
- **slaves_positions** (np.ndarray[np.floating]): Initial positions of the slave nodes. Shape (3, n), where each column contains the (x, y, z) coordinates of a slave node in the physical space.

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

- View Source

Add pointwise Dirichlet conditions by prescribing values at specific DOFs.

This is a convenience method for adding equations of the form:

```
u[i] = v
```

for given indices `i` and corresponding values `v`.

Parameters

- **inds** (np.ndarray[np.integer]): Indices of the DOFs to constrain.
- **vals** (np.ndarray[np.floating], optional): Values to prescribe at the corresponding indices. If None, zeros are used. Must have the same size as `inds`.

Raises

- **AssertionError**: If `vals` is provided and does not match the shape of `inds`.

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

- View Source

Construct the transformation (C, k) that enforces the Dirichlet constraints.

Solves the linear constraint system D @ u = c by computing:

- a basis `C` for 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 be written as:

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

where `dof` is a reduced set of unconstrained degrees of freedom.

Returns

- **C** (sps.spmatrix): Basis of the nullspace of D, of shape (n_full_dofs, n_free_dofs), such that D @ C = 0.

- **k** (np.ndarray[np.floating]): Particular solution of shape (n_full_dofs,) such that D @ k = c.

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

Compute and return the reduced transformation matrices `(C_u, k_u)` for the physical degrees of freedom.

This method extracts only the first `nb_dofs_init` rows of the full constraint transformation matrices `(C, k)` produced by `make_C_k()`, yielding a pair `(C_u, k_u)` such that the physical solution vector `u_phys = C_u @ dof_ext + k_u` satisfies all imposed Dirichlet constraints for the original system.

Returns

- **tuple[sps.spmatrix, np.ndarray]**: A tuple `(C_u, k_u)` where:
  - `C_u` (`sps.spmatrix`): The reduced nullspace basis matrix of shape (`nb_dofs_init` , n_free_dofs).
  - `k_u` (`np.ndarray`): The reduced particular solution vector of shape (`nb_dofs_init` ,).

Notes

- Only the rows corresponding to the initial (physical) degrees of freedom are returned.
- The full transformation matrices may include additional reference DOFs, which are omitted here.

def create_dirichlet(self): • View Source

def get_ref_multipliers_from_internal_residual(self, K_u_minus_f): • View Source

Compute the Lagrange multipliers associated with reference point constraints from the internal residual vector of the mechanical problem.

This method reconstructs the multipliers λ enforcing the constraints linked to reference degrees of freedom (DOFs) using the internal residual `K_u_minus_f = K @ u - f`. The derivation relies on the relation:

```
C_ref.T @ λ = - C_u.T @ (K @ u - f),
```

where the transformation matrix C = [C_u; C_ref] maps reduced DOFs to the full set (physical + reference) while satisfying all constraint equations (built via a QR decomposition in `make_C_k`).

Solving for λ in a least-squares sense yields:

```
λ = - (C_ref @ C_ref.T)^{-1} @ C_ref @ C_u.T @ (K @ u - f).
```

Parameters

- **K_u_minus_f** (np.ndarray): The internal residual vector (K @ u - f) of size `(nb_dofs_init,)`, expressed only for the physical DOFs.

Returns

- **np.ndarray**: The Lagrange multipliers λ associated with the reference point constraints.

Notes

- The result corresponds to the reaction forces (or generalized forces) transmitted by the reference DOFs onto the system, ensuring equilibrium.
- The internal residual `K @ u - f` must be assembled consistently with the stiffness matrix `K` and the load vector `f`.
- This method assumes that `C_ref @ C_ref.T` is invertible, which is guaranteed if the reference constraints are linearly independent.