



**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-sparse
```

## Main Modules

- **Dirichlet**

Manages Dirichlet boundary conditions by applying an affine mapping ( $u = C @ \text{dof} + k$ ).

*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

This project is licensed under the [CeCILL License](#).

- [View Source](#)



## class IGAPatch:

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 compute the shape functions.
- **ctrl\_pts** (npt.NDArray[np.float\_]): Control points defining the patch geometry.
- **E** (float): Young's modulus of the material.
- **nu** (float): Poisson's ratio of the material.
- **xi** (npt.NDArray[np.float\_]): Discretization of the isoparametric coordinate xi.
- **dxi** (npt.NDArray[np.float\_]): Corresponding weights of xi.
- **eta** (npt.NDArray[np.float\_]): Discretization of the isoparametric coordinate eta.
- **deta** (npt.NDArray[np.float\_]): Corresponding weights of eta.
- **zeta** (npt.NDArray[np.float\_]): Discretization of the isoparametric coordinate zeta.
- **dzeta** (npt.NDArray[np.float\_]): Corresponding weights of zeta.
- **F\_N** (npt.NDArray[np.float\_]): Surface forces applied on the corresponding side of the patch.

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

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

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

Initialize the IGAPatch with the given parameters.

### Parameters

- **spline** (BSpline): B-spline volume used as the patch.
- **ctrl\_pts** (npt.NDArray[np.float\_]): Control points defining the patch geometry.
- **E** (float): Young's modulus of the material.
- **nu** (float): Poisson's ratio of the material.
- **F\_N** (npt.NDArray[np.float\_], optional): Surface 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[typing.Any, numpy.dtype[numpy.float64]]

**E:** float

**nu:** float

**xi:** numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**dxI**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**eta**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**deta**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**zeta**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**dzeta**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**F\_N**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

**H**

```
def jacobian(
    self,
    dN_dXI: tuple
) -> tuple[numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]], numpy.ndarray[typing.Any, numpy.dtype[r
```

• [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** (npt.NDArray[np.float\_]): Jacobian matrix, with shape (3(phy), 3(param), nb\_intg\_pts).
- **Jinv** (npt.NDArray[np.float\_]): Inverse of the Jacobian matrix, with shape (3(param), 3(phy), nb\_intg\_pts).
- **detJ** (npt.NDArray[np.float\_]): Determinant of the Jacobian matrix, with shape (nb\_intg\_pts,).

```
def grad_N(
    self,
    Jinv: numpy.ndarray,
    dN_dXI: tuple
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.object_]]:
```

• [View Source](#)

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

#### Parameters

- **Jinv** (npt.NDArray[np.float\_]): 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** (npt.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.ndarray[typing.Any, numpy.dtype[numpy.float64]]:
```

• [View Source](#)

Compute the measure for integrating 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** (npt.NDArray[np.float\_]): Array containing the determinant values. Its shape is (nb\_intg\_pts,).

## Returns

- **W** (npt.NDArray[np.float\_]): Array containing the measure for integrating in the physical space. Its shape is (nb\_intg\_pts,).

**def stiffness(self) -> scipy.sparse\_base.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[typing.Any, numpy.dtype[numpy.float64]]:**

• [View Source](#)

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

## Returns

- **rhs** (npt.NDArray[np.float\_]): 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,  
XI: list  
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:**

• [View Source](#)

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

## Parameters

- **U** (npt.NDArray[np.float\_]): Displacement field as a numpy array of shape (3(phy), nb\_ctrl\_pts).
- **XI** (list[npt.NDArray[np.float\_]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

## Returns

- **eps** (npt.NDArray[np.float\_]): Strain tensor epsilon in voight notation computed as a numpy array of shape (6, nb\_param\_pts).

**def sigma(  
self,  
U: numpy.ndarray,  
XI: list  
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:**

• [View Source](#)

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

## Parameters

- **U** (npt.NDArray[np.float\_]): Displacement field as a numpy array of shape (3(phy), nb\_ctrl\_pts).
- **XI** (list[npt.NDArray[np.float\_]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

## Returns

- **sig** (npt.NDArray[np.float\_]): Stress tensor sigma in voight notation computed as a numpy array of shape (6, nb\_param\_pts).

```
def sigma_eig(
    self,
    U: numpy.ndarray,
    XI: list
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:
```

• [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** (npt.NDArray[np.float\_]): Displacement field as a numpy array of shape (3(phy), nb\_ctrl\_pts).
- **XI** (list[npt.NDArray[np.float\_]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

#### Returns

- **sig\_eig** (npt.NDArray[np.float\_]): Eigenvalues of the stress tensor computed as a numpy array of shape (nb\_param\_pts, 3).

```
def von_mises(
    self,
    U: numpy.ndarray,
    XI: list
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:
```

• [View Source](#)

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

#### Parameters

- **U** (npt.NDArray[np.float\_]): Displacement field as a numpy array of shape (3(phy), nb\_ctrl\_pts).
- **XI** (list[npt.NDArray[np.float\_]]): List of isoparametric coordinates for each direction xi, eta, and zeta.

#### Returns

- **vm** (npt.NDArray[np.float\_]): Von Mises stress computed as a numpy array of shape (nb\_param\_pts,).

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

• [View Source](#)

Save data for visualization in ParaView.

#### Parameters

- **U** (npt.NDArray[np.float\_]): 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.



- [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,  
    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 assembly\_block(self, block):**

- [View Source](#)

Process a block of patches, accumulating contributions to rhs and lhs. Each block has its own progress bar.

```
def lhs_rhs(  
    self,  
    verbose: bool = False  
) -> tuple[scipy.sparse._base.spmatrix, numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]]:
```

- [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.

### Returns

- **lhs, rhs** (tuple[sps.spmatrix, npt.NDArray[np.float\_]]): The assembled sparse left-hand side matrix and right-hand side vector.

```
def apply_dirichlet(
    self,
    lhs: scipy.sparse._base.spmatrix,
    rhs: numpy.ndarray,
    verbose: bool = False
) -> tuple[scipy.sparse._base.spmatrix, numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]]: • 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** (npt.NDArray[np.float\_]): 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, npt.NDArray[np.float\_]]): 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._base.spmatrix,
    rhs: numpy.ndarray,
    iterative_solve: bool = False,
    verbose: bool = True
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]: • 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** (npt.NDArray[np.float\_]): 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** (npt.NDArray[np.float\_]): The solution vector representing the degrees of freedom.

```
def solve(
    self,
    iterative_solve=False
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]: • 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** (npt.NDArray[np.float\_]): Solution vector representing the computed displacements in packed notation.  
Shape : (3(phy), nb\_unique\_nodes)

```
def save_paraview(
    self,
    u: numpy.ndarray,
    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** (npt.NDArray[np.float\_]): 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.



- [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** (npt.NDArray[np.float\_]): The vector used in the affine mapping from **dof** to **u**.

**Dirichlet**(C: scipy.sparse.\_base.spmatrix, k: numpy.ndarray)

- [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** (npt.NDArray[np.float\_]): The vector used to map degrees of freedom **dof** to displacements **u**.

**C**: scipy.sparse.\_csc.csc\_matrix

**k**: numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]

@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, k: numpy.ndarray):

- [View Source](#)

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

### Parameters

- **inds** (npt.NDArray[np.int\_]): Indices of the displacement field **u** to be locked.
- **k** (npt.NDArray[np.float\_]): Values to lock the specified indices of **u** to. After this, **u[inds]** are set to **k[inds]** while other components are left free.

### Returns

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

**def set\_u\_inds\_vals**(self, inds: numpy.ndarray, vals: numpy.ndarray):

- [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** (npt.NDArray[np.int\_]): Indices of the displacement field **u** to be locked.
- **vals** (npt.NDArray[np.float\_]): Values to lock the specified indices of **u** to.

```
def slave_reference_linear_relation(
    self,
    slaves: numpy.ndarray,
    references: numpy.ndarray,
    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  $\mathbf{u} = \mathbf{C}@\mathbf{dof} + \mathbf{k}$  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
) -> tuple[numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]], 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  $\mathbf{u} = \mathbf{C} @ \mathbf{dof} + \mathbf{k}$ , and its derivative is **C**.

## Parameters

- **dof** (npt.NDArray[np.float\_]): The degrees of freedom of the problem, representing the input to the affine mapping.

## Returns

- **u, du\_ddof** (tuple[npt.NDArray[np.float\_], sps.csc\_matrix]): A tuple containing the displacement field **u** and its derivative with respect to **dof**.

```
def u(
    self,
    dof: numpy.ndarray
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:
```

• [View Source](#)

Computes the displacement field **u** using the affine mapping  $\mathbf{u} = \mathbf{C} @ \mathbf{dof} + \mathbf{k}$ .

## Parameters

- **dof** (npt.NDArray[np.float\_]): The degrees of freedom of the problem, representing the input to the affine mapping.

## Returns

- **u** (npt.NDArray[np.float\_]): The computed displacement field **u**.

```
def dof_lsq(
    self,
    u: numpy.ndarray
) -> numpy.ndarray[typing.Any, numpy.dtype[numpy.float64]]:
```

• [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** (npt.NDArray[np.float\_]): The displacement field from which to compute the degrees of freedom.

#### Returns

- **dof** (npt.NDArray[np.float\_]): The computed degrees of freedom corresponding to the given displacement field.

```
@nb.njit(cache=True)
```

```
def slave_reference_linear_relation_sort(slaves: numpy.ndarray, references: numpy.ndarray
) -> 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,
    indptr: numpy.ndarray,
    data: numpy.ndarray,
    k: numpy.ndarray,
    slaves: numpy.ndarray,
    references: numpy.ndarray,
    coefs: numpy.ndarray,
    sorted_slaves: numpy.ndarray
) -> 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.