



treeIDW is a Python library for performing **Inverse Distance Weighting (IDW)** interpolation using an efficient **KD-tree-based selection strategy**.

It is designed to be easy to use for newcomers while offering fine-grained control and performance-oriented options for advanced users in numerical methods and spatial data analysis.

Key Features

- Efficient IDW interpolation using KD-tree nearest-neighbor selection
- Automatic exclusion of boundary nodes with negligible contribution
- Optimized numerical kernels powered by [numba](#)
- Scalable to large datasets (millions of interpolation points)
- Simple API with expert-level tunable parameters

Installation

treeIDW is available on **PyPI**.

```
pip install treeIDW
```

Development installation (from source)

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

Dependencies

The core dependencies are:

- [numpy](#)

- `scipy`
- `numba`

These are automatically installed when using `pip`.

Package Structure

- **`treeIDW.treeIDW`**
Core IDW interpolation engine.
Uses a KD-tree to select only boundary nodes with significant influence, improving both accuracy and performance.
 - **`treeIDW.helper_functions`**
Low-level, performance-critical routines for IDW weight computation.
Implemented with `numba`, including vectorized and parallelized variants.
 - **`treeIDW.weight_function`**
Definition of the IDW weight function.
The default implementation uses inverse squared distance, but custom weight laws can be implemented if needed.
-

Examples

Example scripts are provided in the `examples/` directory:

- **Graphical demonstration**
Interpolation of a rotating vector field inside a square domain.
 - **Large-scale computation**
Propagation of a scalar field from 1,000 boundary nodes to 1,000,000 internal points, highlighting scalability.
 - **Logo generation**
The generation process of the *treeIDW* logo itself, where the letters “IDW” are encoded as a vector field and interpolated on a 2D grid.
-

Documentation

- Online documentation: <https://dorian210.github.io/treeIDW/>
 - API reference is also available in the `docs/` directory of the repository.
-

Contributions

This project is currently not open to active development contributions.
However, bug reports and suggestions are welcome via the issue tracker.

License

This project is distributed under the **CeCILL License**.
See [LICENSE.txt](#) for details.

- [View Source](#)

treeIDW.weight_function



- [View Source](#)

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

```
def compute_weight(dist_squared: float) -> float:
```

- [View Source](#)

Compute the IDW weight. Can be changed but is usually $1/\text{dist_squared}$.

Parameters

- **dist_squared** (float): Squared distance between the interior node and the boundary node.

Returns

- **weight** (float): IDW weight.



- [View Source](#)

```
def treeIDW(  
    boundary_nodes: numpy.ndarray[numpy.floating],  
    boundary_field: numpy.ndarray[numpy.floating],  
    internal_nodes: numpy.ndarray[numpy.floating],  
    neglectible_treshold: float = 0.2,  
    bisect_rtol: float = 0.001,  
    parallel: bool = False  
) -> numpy.ndarray[numpy.floating]:
```

- [View Source](#)

Performs IDW interpolation using a KD-tree to select relevant boundary nodes for each internal node. Only boundary nodes with significant weights are included in the interpolation.

Parameters

- **boundary_nodes** (np.ndarray[np.floating]): Boundary nodes where the data is known. Must be of shape (`n_interp` , `space_dim`).
- **boundary_field** (np.ndarray[np.floating]): Known data. Must be of shape (`n_interp` , `field_dim`).
- **internal_nodes** (np.ndarray[np.floating]): Internal nodes where the interpolator is evaluated. Must be of shape (`n_eval` , `space_dim`).
- **neglectible_treshold** (float, optional): Relative weight threshold below which boundary nodes are ignored, by default 0.2
- **bisect_rtol** (float, optional): Relative tolerance for bisection convergence, by default 1e-3
- **parallel** (bool, optional): Whether to use parallel implementation, by default False

Returns

- **internal_field** (np.ndarray[np.floating]): Interpolated data. Should be of shape (`n_eval` , `field_dim`).



- [View Source](#)

```
@nb.njit(cache=True)
def inv_dist_weight(
    boundary_nodes: numpy.ndarray[numpy.floating],
    boundary_field: numpy.ndarray[numpy.floating],
    internal_nodes: numpy.ndarray[numpy.floating],
    relevant_nodes_inds_flat: numpy.ndarray[numpy.integer],
    relevant_nodes_inds_sizes: numpy.ndarray[numpy.integer]
) -> numpy.ndarray[numpy.floating]:
```

- [View Source](#)

Performs the Inverse Distance Weighting interpolation method using only provided boundary nodes indices in the weighted sum. This function is designed to be used after selecting the relevant nodes through a KD-tree method.

Parameters

- **boundary_nodes** (np.ndarray[np.floating]): Boundary nodes where the data is known. Must be of shape (`n_interp` , `space_dim`).
- **boundary_field** (np.ndarray[np.floating]): Known data. Must be of shape (`n_interp` , `field_dim`).
- **internal_nodes** (np.ndarray[np.floating]): Internal nodes where the interpolator is evaluated. Must be of shape (`n_eval` , `space_dim`).
- **relevant_nodes_inds_flat** (np.ndarray[np.integer]): Boundary nodes indices that weight in the IDW interpolator for each internal node. Must contain `n_eval` concatenated lists of indices between 0 and `n_interp` excluded.
- **relevant_nodes_inds_sizes** (np.ndarray[np.integer]): Sizes of each of the `n_eval` lists concatenated in `relevant_nodes_inds_flat` .

Returns

- **internal_field** (np.ndarray[np.floating]): Interpolated data. Should be of shape (`n_eval` , `field_dim`).

```
@nb.njit(parallel=True, cache=True)
def inv_dist_weight_parallel(
    boundary_nodes: numpy.ndarray[numpy.floating],
    boundary_field: numpy.ndarray[numpy.floating],
    internal_nodes: numpy.ndarray[numpy.floating],
    relevant_nodes_inds_flat: numpy.ndarray[numpy.integer],
    relevant_nodes_inds_sizes: numpy.ndarray[numpy.integer]
) -> numpy.ndarray[numpy.floating]:
```

- [View Source](#)

Performs the Inverse Distance Weighting interpolation method using only provided boundary nodes indices in the weighted sum. This function is designed to be used after selecting the relevant nodes through a KD-tree method.

Parameters

- **boundary_nodes** (np.ndarray[np.floating]): Boundary nodes where the data is known. Must be of shape (`n_interp` , `space_dim`).
- **boundary_field** (np.ndarray[np.floating]): Known data. Must be of shape (`n_interp` , `field_dim`).
- **internal_nodes** (np.ndarray[np.floating]): Internal nodes where the interpolator is evaluated. Must be of shape (`n_eval` , `space_dim`).
- **relevant_nodes_inds_flat** (np.ndarray[np.integer]): Boundary nodes indices that weight in the IDW interpolator for each internal node. Must contain `n_eval` concatenated lists of indices between 0 and `n_interp` excluded.
- **relevant_nodes_inds_sizes** (np.ndarray[np.integer]): Sizes of each of the `n_eval` lists concatenated in `relevant_nodes_inds_flat` .

Returns

- **internal_field** (np.ndarray[np.floating]): Interpolated data. Should be of shape (`n_eval` , `field_dim`).

```
@nb.njit(cache=True)
def bisect_weight_elem(
    dist_squared_a: float,
    dist_squared_b: float,
    weight_treshold: float,
    rtol: float
) -> float:
```

• [View Source](#)

Compute the squared distance that correspond to a specific weight threshold using a bisection approach.

Parameters

- **dist_squared_a** (float): Left boundary of the interval containing the looked for weight treshold preimage.
- **dist_squared_b** (float): Right boundary of the interval containing the looked for weight treshold preimage.
- **weight_treshold** (float): Image of the looked for squared distance by the weight function of the IDW algorithm.
- **rtol** (float, optional): The relative tolerance is used to compute the necessary number of bisection iterations, by default 1e-3

Returns

- **dist_squared_c** (float): Weight treshold preimage found through bisection.

bisect_weight = <numba._GUFunc 'bisect_weight'>

Vectorized version of bisect_weight_elem that finds squared distances corresponding to weight thresholds.

Parameters

- **dist_squared_a** (float): Left boundary of interval containing weight threshold preimage
- **dist_squared_b** (float): Right boundary of interval containing weight threshold preimage
- **weight_treshold** (float): Target weight threshold value
- **rtol** (float): Relative tolerance for bisection convergence
- **out** (ndarray): Output array for storing computed squared distance

bisect_weight_parallel = <ufunc 'bisect_weight_parallel'>

Vectorized version of bisect_weight_elem that finds squared distances corresponding to weight thresholds.

Parallelized version of the function.

Parameters

- **dist_squared_a** (float): Left boundary of interval containing weight threshold preimage
- **dist_squared_b** (float): Right boundary of interval containing weight threshold preimage
- **weight_treshold** (float): Target weight threshold value
- **rtol** (float): Relative tolerance for bisection convergence
- **out** (ndarray): Output array for storing computed squared distance

compute_weight_vectorized = <numba._GUFunc 'compute_weight_vectorized'>

Vectorized version of compute_weight function for IDW calculations.

Parameters

- **dist_squared** (float): Squared distance between interior and boundary nodes.
- **out** (ndarray): Output array containing the computed IDW weights.

compute_weight_vectorized_parallel = <ufunc 'compute_weight_vectorized_parallel'>

Vectorized version of compute_weight function for IDW calculations. Parallelized version of the function.

Parameters

- **dist_squared** (float): Squared distance between interior and boundary nodes.
- **out** (ndarray): Output array containing the computed IDW weights.