Merge StarDist Masks

Niklas Netter

Jul 28, 2023

CONTENTS

1 Features 3

2 Requirements 5

3 Usage 7

4 Installation 9

5 Contributing 11

6 License 13

7 Issues 15

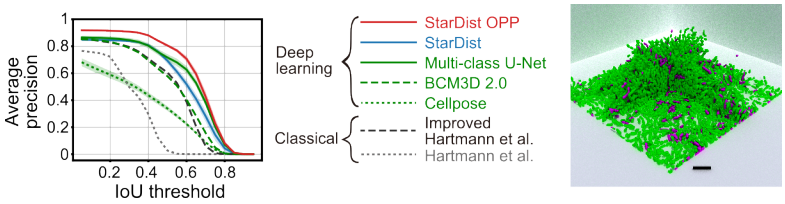
8 How to cite 17

9 Credits 19

Python Module Index 37

Index 39

This repository contains the python package for the new StarDist post-processing step *StarDist OPP*. *StarDist OPP* allows to use StarDist segmentation on non-star-convex objects. In our paper, we show that *StarDist OPP* outperforms other methods in instance segmentation tasks for three-dimensional microbial biofilms. Check it out for more information.

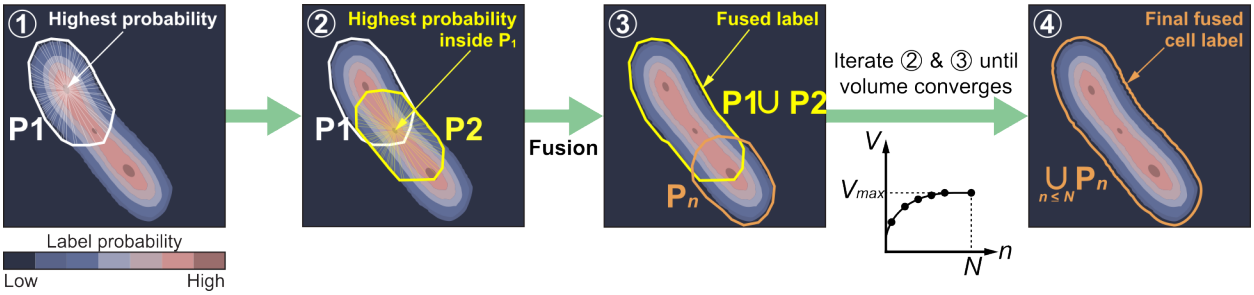


CHAPTER

ONE

FEATURES

- *StarDist OPP* merges masks together - hence the repository name



- *StarDist OPP* works in 2D and 3D
- In 2D, *StarDist OPP* works also on big and winding objects

CHAPTER

TWO

REQUIREMENTS

- A StarDist installation.

CHAPTER

THREE

USAGE

Please see the EXAMPLE in Usage for details or check out the tutorial of our napari plugin to directly use *StarDist OPP* on your data.

CHAPTER

FOUR

INSTALLATION

You can install *StarDist OPP* via pip from PyPI:

```
$ pip install merge-stardist-masks
```

CHAPTER

FIVE

CONTRIBUTING

Contributions are very welcome. To learn more, see the Contributor Guide.

CHAPTER

SIX

LICENSE

Distributed under the terms of the MIT license, *StarDist OPP* is free and open source software.

CHAPTER

SEVEN

ISSUES

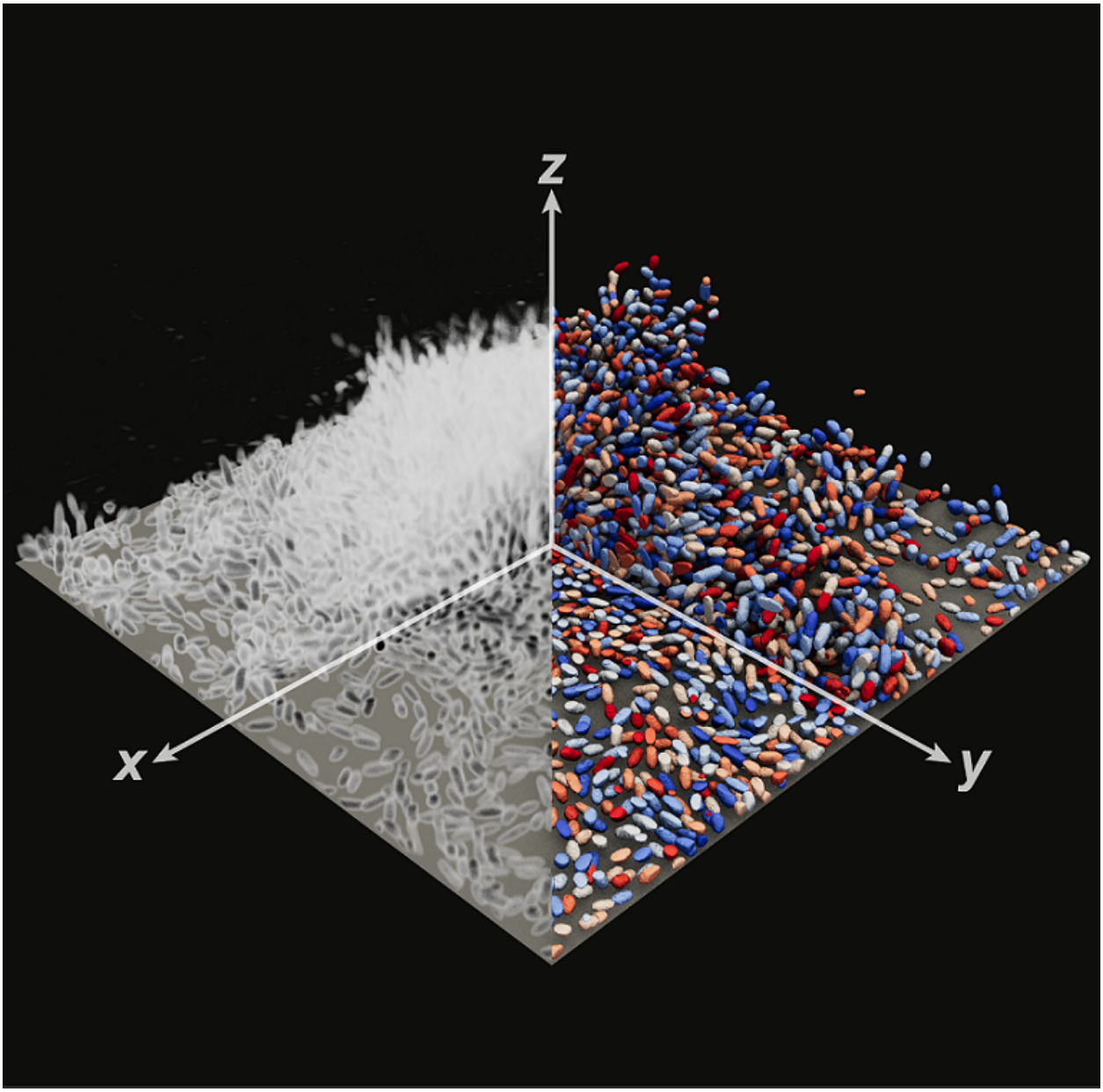
If you encounter any problems, please file an issue along with a detailed description.

CHAPTER

EIGHT

HOW TO CITE

```
@article{https://doi.org/10.1111/mmi.15064,
author = {Jelli, Eric and Ohmura, Takuya and Netter, Niklas and Abt, Martin and Jiménez-
↪Siebert, Eva and Neuhaus, Konstantin and Rode, Daniel K. H. and Nadell, Carey D. and␣
↪Drescher, Knut},
title = {Single-cell segmentation in bacterial biofilms with an optimized deep learning␣
↪method enables tracking of cell lineages and measurements of growth rates},
journal = {Molecular Microbiology},
volume = {119},
number = {6},
pages = {659-676},
keywords = {3D segmentation, biofilm, deep learning, image analysis, image cytometry,␣
↪Vibrio cholerae},
doi = {https://doi.org/10.1111/mmi.15064},
url = {https://onlinelibrary.wiley.com/doi/abs/10.1111/mmi.15064},
eprint = {https://onlinelibrary.wiley.com/doi/pdf/10.1111/mmi.15064},
abstract = {Abstract Bacteria often grow into matrix-encased three-dimensional (3D)␣
↪biofilm communities, which can be imaged at cellular resolution using confocal␣
↪microscopy. From these 3D images, measurements of single-cell properties with high␣
↪spatiotemporal resolution are required to investigate cellular heterogeneity and␣
↪dynamical processes inside biofilms. However, the required measurements rely on the␣
↪automated segmentation of bacterial cells in 3D images, which is a technical challenge.
↪ To improve the accuracy of single-cell segmentation in 3D biofilms, we first␣
↪evaluated recent classical and deep learning segmentation algorithms. We then extended␣
↪StarDist, a state-of-the-art deep learning algorithm, by optimizing the post-
↪processing for bacteria, which resulted in the most accurate segmentation results for␣
↪biofilms among all investigated algorithms. To generate the large 3D training dataset␣
↪required for deep learning, we developed an iterative process of automated␣
↪segmentation followed by semi-manual correction, resulting in >18,000 annotated Vibrio␣
↪cholerae cells in 3D images. We demonstrate that this large training dataset and the␣
↪neural network with optimized post-processing yield accurate segmentation results for␣
↪biofilms of different species and on biofilm images from different microscopes.␣
↪Finally, we used the accurate single-cell segmentation results to track cell lineages␣
↪in biofilms and to perform spatiotemporal measurements of single-cell growth rates␣
↪during biofilm development.},
year = {2023}
}
```


z
x
y

CHAPTER

NINE

CREDITS

This project was generated from @cjolowicz's Hypermodern Python Cookiecutter template.

9.1 Usage

merge_stardist_masks.naive_fusion.**naive_fusion**(*dists: ndarray[Any, dtype[float64]]*, *probs: ndarray[Any, dtype[float64]]*, *rays: Rays_Base | None = None*, *prob_thresh: float = 0.5*, *grid: Tuple[int, ...] = (2, 2, 2)*, *no_slicing: bool = False*, *max_full_overlaps: int = 2*, *erase_probs_at_full_overlap: bool = False*, *show_overlaps: bool = False*, *respect_probs: bool = False*) → ndarray[Any, dtype[uint16]] | ndarray[Any, dtype[int32]]

Merge overlapping masks given by dists, probs, rays.

Performs a naive iterative scheme to merge the masks that a StarDist network has calculated to generate a label image. This function can perform 2D and 3D segmentation. For 3D segmentation *rays* has to be passed from the StarDist model.

Parameters

- **dists** (*ndarray[Any, dtype[float64]]*) – 3- or 4-dimensional array representing distances of each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x, n_rays)`, for 3D predictions it is `(len_z, len_y, len_x, n_rays)`.
- **probs** (*ndarray[Any, dtype[float64]]*) – 2- or 3-dimensional array representing the probabilities for each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x)`, for 3D predictions it is `(len_z, len_y, len_x)`.
- **rays** (*Rays_Base | None*) – For 3D predictions *rays* must be set otherwise a `ValueError` is raised. It should be the `Rays_Base` instance used by the StarDist model.
- **prob_thresh** (*float*) – Only masks with probability above this threshold are considered.
- **grid** (*Tuple[int, ...]*) – Should be of length 2 for 2D and of length 3 for 3D segmentation. This is the grid information about the subsampling occuring in the StarDist model.
- **no_slicing** (*bool*) – For very big and winded objects this should be set to `True`. However, this might result in longer calculation times.
- **max_full_overlaps** (*int*) – Maximum no. of full overlaps before current object is treated as finished.
- **erase_probs_at_full_overlap** (*bool*) – If set to `True` probs are set to -1 whenever a full overlap is detected.

- **show_overlaps** (*bool*) – If set to true, overlaps are set to -1.
- **respect_probs** (*bool*) – If set to true, overlapping elements are overwritten by considering their probabilities. Only works when show_overlaps is 'false'.

Returns

The label image with uint16 labels. For 2D, the shape is `(len_y * grid[0], len_x * grid[1])` and for 3D it is `(len_z * grid[0], len_y * grid[1], len_z * grid[2])`.

Raises

- **ValueError** – If *rays* is `None` and 3D inputs are given or when `probs.ndim != len(grid)`. # noqa: DAR402 ValueError
- **NotImplementedError** – If grid is anisotropic and respect_probs is set to true.

Return type

ndarray[*Any*, *dtype*[*uint16*]] | *ndarray*[*Any*, *dtype*[*int32*]]

Example

```
>>> from merge_stardist_masks.naive_fusion import naive_fusion
>>> from stardist.rays3d import rays_from_json
>>> probs, dists = model.predict(img)  # model is a 3D StarDist model
>>> rays = rays_from_json(model.config.rays_json)
>>> lbl = naive_fusion(dists, probs, rays, grid=model.config.grid)
```

9.2 napari plugin

9.2.1 Installation

Make sure to have a running napari installation.

Via pip

In the environment with your napari installation run:

```
$ pip install merge-stardist-masks
```

Within napari

Within napari go to `Plugins -> Install/Uninstall Plugins...` and search for *napari-merge-stardist-masks* in the lower list. Then click on the blue *install* button.

After installation

Make sure to restart napari after the installation. If you do not find the plugins, go to `Plugins -> Install/Uninstall Plugins...` and toggle the checkboxes in the upper list for *stardist-napari* and *napari-merge-stardist-masks*.

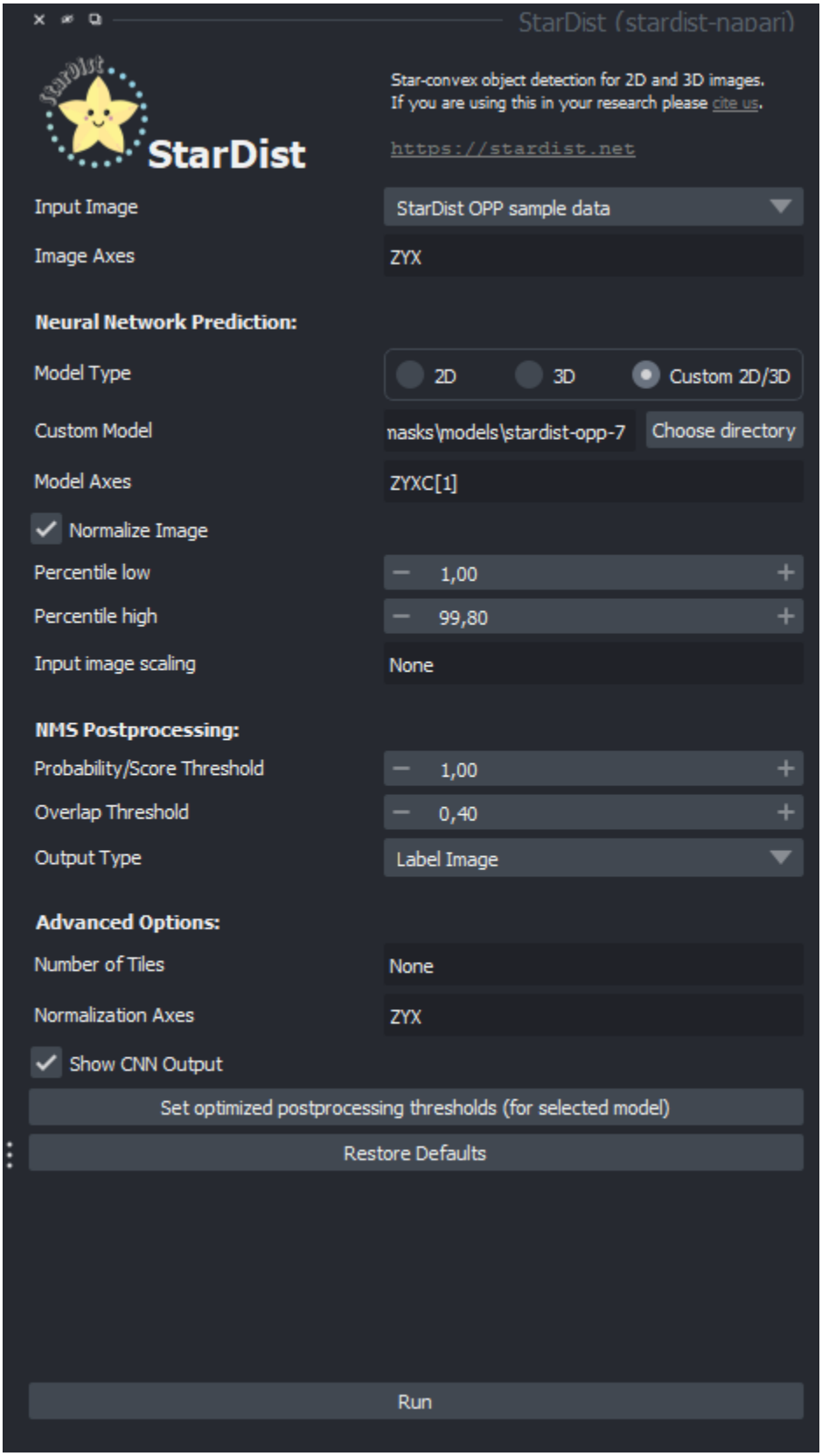
9.2.2 Usage

Preparations

Download one of the pre-trained StarDist models from here and unzip the file.

Run a segmentation

1. Load sample data with `File -> Open sample -> StarDist OPP sample data`
2. Click `Plugins -> StarDist OPP`. Two widgets will open, the StarDist plugin and this plugin.
3. All the parameters in the *StarDist plugin* should be correctly set already. Make sure that the axes in the field `Image Axes` are correct, for a 3D image it should be `ZYX`.
4. Select `Custom 2D/3D` in the field `Model Type` and choose the directory where you unzipped the pre-trained model in the `Custom Model` field. See the image below for the correct settings.

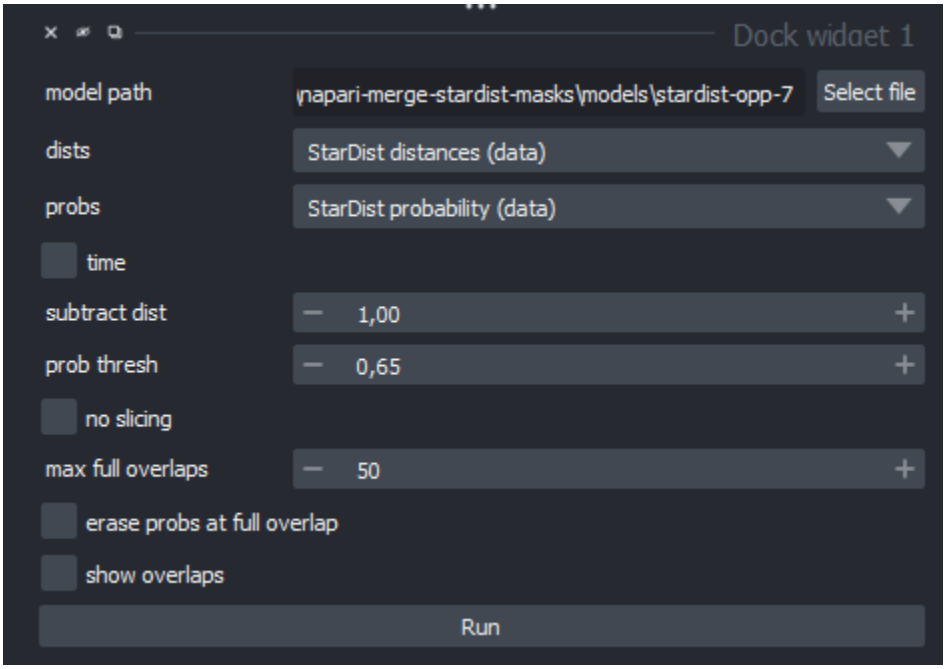


5. Hit Run. And wait until the CNN calculates the outputs. The outputs of the CNN are displayed once they are

calculated.

6. In the *StarDist OPP* widget, select again the path to the unzipped pre-trained model in the field `model path`. Then select `StarDist distances (data)` and `StarDist probability (data)` for the `dists` and `probs` fields, respectively.

7. You can play around with the other fields. However, this might lead to errors. For 3D images, you should set `subtract dist` to `1.00` the other settings are already fine. See the following image for proper settings.



8. Hit `Run` in the *StarDist OPP* widget. The post-processing starts and might take some time (on our machine it takes ~10 minutes). Once the post-processing is done, the label image will be shown in the viewer.

9.3 Reference

9.3.1 merge_stardist_masks.naive_fusion

Naively merge all masks that have sufficient overlap and probability.

merge_stardist_masks.naive_fusion.**get_poly_list_to_label**(*shape: Tuple[int, ...]*, *rays: Rays_Base | None*) → merge_stardist_masks.naive_fusion.PolyToLabelSignature

Depending on len(shape) return different functions to calculate labels.

Parameters

- **shape** (*Tuple[int, ...]*) –
- **rays** (*Rays_Base | None*) –

Return type
merge_stardist_masks.naive_fusion.PolyToLabelSignature

merge_stardist_masks.naive_fusion.**get_poly_to_label**(*shape: Tuple[int, ...]*, *rays: Rays_Base | None*) → merge_stardist_masks.naive_fusion.PolyToLabelSignature

Depending on len(shape) return different functions to calculate labels.

Parameters

- **shape** (*Tuple[int, ...]*) –
- **rays** (*Rays_Base | None*) –

Return type
merge_stardist_masks.naive_fusion.PolyToLabelSignature

merge_stardist_masks.naive_fusion.**inflate_array**(*x: ndarray[Any, dtype[T]]*, *grid: Tuple[int, ...]*, *default_value: int | float = 0*) → ndarray[Any, dtype[T]]

Create new array with increased shape but old values of x.

Parameters

- **x** (*ndarray[Any, dtype[T]]*) –
- **grid** (*Tuple[int, ...]*) –
- **default_value** (*int | float*) –

Return type
ndarray[*Any*, *dtype*[*T*]]

merge_stardist_masks.naive_fusion.**mesh_from_shape**(*shape: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Convenience function to generate a mesh.

Parameters
shape (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**my_polygons_list_to_label**(*dists: numpy.typing.ArrayLike*, *points: numpy.typing.ArrayLike*, *shape: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Convenience funtion to pass 1-d arrays to polygons_to_label.

Parameters

- **dists** (*numpy.typing.ArrayLike*) –
- **points** (*numpy.typing.ArrayLike*) –
- **shape** (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**my_polygons_to_label**(*dists: numpy.typing.ArrayLike*, *points: numpy.typing.ArrayLike*, *shape: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Convenience funtion to pass 1-d arrays to polygons_to_label.

Parameters

- **dists** (*numpy.typing.ArrayLike*) –
- **points** (*numpy.typing.ArrayLike*) –
- **shape** (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**my_polyhedron_list_to_label**(*rays: Rays_Base*, *dists: numpy.typing.ArrayLike*, *points: numpy.typing.ArrayLike*, *shape: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Convenience funtion to pass 1-d arrays to polyhedron_to_label.

Parameters

- **rays** (*Rays_Base*) –
- **dists** (*numpy.typing.ArrayLike*) –
- **points** (*numpy.typing.ArrayLike*) –
- **shape** (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**my_polyhedron_to_label**(*rays: Rays_Base*, *dists: numpy.typing.ArrayLike*, *points: numpy.typing.ArrayLike*, *shape: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Convenience funtion to pass 1-d arrays to polyhedron_to_label.

Parameters

- **rays** (*Rays_Base*) –
- **dists** (*numpy.typing.ArrayLike*) –

- **points** (*numpy.typing.ArrayLike*) –
- **shape** (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**naive_fusion**(*dists: ndarray[Any, dtype[float64]]*, *probs: ndarray[Any, dtype[float64]]*, *rays: Rays_Base | None = None*, *prob_thresh: float = 0.5*, *grid: Tuple[int, ...] = (2, 2, 2)*, *no_slicing: bool = False*, *max_full_overlaps: int = 2*, *erase_probs_at_full_overlap: bool = False*, *show_overlaps: bool = False*, *respect_probs: bool = False*) → ndarray[Any, dtype[uint16]] | ndarray[Any, dtype[int32]]

Merge overlapping masks given by dists, probs, rays.

Performs a naive iterative scheme to merge the masks that a StarDist network has calculated to generate a label image. This function can perform 2D and 3D segmentation. For 3D segmentation *rays* has to be passed from the StarDist model.

Parameters

- **dists** (*ndarray[Any, dtype[float64]]*) – 3- or 4-dimensional array representing distances of each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x, n_rays)`, for 3D predictions it is `(len_z, len_y, len_x, n_rays)`.
- **probs** (*ndarray[Any, dtype[float64]]*) – 2- or 3-dimensional array representing the probabilities for each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x)`, for 3D predictions it is `(len_z, len_y, len_x)`.
- **rays** (*Rays_Base | None*) – For 3D predictions *rays* must be set otherwise a `ValueError` is raised. It should be the `Rays_Base` instance used by the StarDist model.
- **prob_thresh** (*float*) – Only masks with probability above this threshold are considered.
- **grid** (*Tuple[int, ...]*) – Should be of length 2 for 2D and of length 3 for 3D segmentation. This is the grid information about the subsampling occuring in the StarDist model.
- **no_slicing** (*bool*) – For very big and winded objects this should be set to `True`. However, this might result in longer calculation times.
- **max_full_overlaps** (*int*) – Maximum no. of full overlaps before current object is treated as finished.
- **erase_probs_at_full_overlap** (*bool*) – If set to `True` probs are set to -1 whenever a full overlap is detected.
- **show_overlaps** (*bool*) – If set to true, overlaps are set to `-1`.
- **respect_probs** (*bool*) – If set to true, overlapping elements are overwritten by considering their probabilities. Only works when show_overlaps is 'false'.

Returns
The label image with uint16 labels. For 2D, the shape is `(len_y * grid[0], len_x * grid[1])` and for 3D it is `(len_z * grid[0], len_y * grid[1], len_z * grid[2])`.

Raises

- **ValueError** – If *rays* is `None` and 3D inputs are given or when `probs.ndim != len(grid)`. # noqa: DAR402 ValueError
- **NotImplementedError** – If grid is anisotropic and respect_probs is set to true.

Return type
ndarray[*Any*, *dtype*[*uint16*]] | *ndarray*[*Any*, *dtype*[*int32*]]

Example

```
>>> from merge_stardist_masks.naive_fusion import naive_fusion
>>> from stardist.rays3d import rays_from_json
>>> probs, dists = model.predict(img)  # model is a 3D StarDist model
>>> rays = rays_from_json(model.config.rays_json)
>>> lbl = naive_fusion(dists, probs, rays, grid=model.config.grid)
```

merge_stardist_masks.naive_fusion.**naive_fusion_anisotropic_grid**(*dists: ndarray[Any, dtype[float64]]*, *probs: ndarray[Any, dtype[float64]]*, *rays: Rays_Base | None = None*, *prob_thresh: float = 0.5*, *grid: Tuple[int, ...] = (2, 2, 2)*, *no_slicing: bool = False*, *max_full_overlaps: int = 2*, *erase_probs_at_full_overlap: bool = False*, *show_overlaps: bool = False*, *respect_probs: bool = False*) → ndarray[Any, dtype[uint16]] | ndarray[Any, dtype[int32]]

Merge overlapping masks given by dists, probs, rays for anisotropic grid.

Performs a naive iterative scheme to merge the masks that a StarDist network has calculated to generate a label image. This function can perform 2D and 3D segmentation. For 3D segmentation *rays* has to be passed from the StarDist model.

Parameters

- **dists** (*ndarray[Any, dtype[float64]]*) – 3- or 4-dimensional array representing distances of each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x, n_rays)`, for 3D predictions it is `(len_z, len_y, len_x, n_rays)`.
- **probs** (*ndarray[Any, dtype[float64]]*) – 2- or 3-dimensional array representing the probabilities for each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x)`, for 3D predictions it is `(len_z, len_y, len_x)`.
- **rays** (*Rays_Base | None*) – For 3D predictions *rays* must be set otherwise a `ValueError` is raised. It should be the `Rays_Base` instance used by the StarDist model.
- **prob_thresh** (*float*) – Only masks with probability above this threshold are considered.
- **grid** (*Tuple[int, ...]*) – Should be of length 2 for 2D and of length 3 for 3D segmentation. This is the grid information about the subsampling occuring in the StarDist model.
- **no_slicing** (*bool*) – For very big and winded objects this should be set to `True`. However, this might result in longer calculation times.
- **max_full_overlaps** (*int*) – Maximum no. of full overlaps before current object is treated as finished.
- **erase_probs_at_full_overlap** (*bool*) – If set to `True` probs are set to -1 whenever a full overlap is detected.

- **show_overlaps** (*bool*) – If set to true, overlaps are set to -1.
- **respect_probs** (*bool*) – If set to true, overlapping elements are overwritten by considering their probabilities. Only works when show_overlaps is 'false'.

Returns

The label image with uint16 labels. For 2D, the shape is `(len_y * grid[0], len_x * grid[1])` and for 3D it is `(len_z * grid[0], len_y * grid[1], len_z * grid[2])`.

Raises

ValueError – If *rays* is `None` and 3D inputs are given or when `probs.ndim != len(grid)`. # noqa: DAR402 ValueError

Return type

ndarray[*Any*, *dtype*[*uint16*]] | *ndarray*[*Any*, *dtype*[*int32*]]

Example

```
>>> from merge_stardist_masks.naive_fusion import naive_fusion_anisotropic_grid
>>> from stardist.rays3d import rays_from_json
>>> probs, dists = model.predict(img)  # model is a 3D StarDist model
>>> rays = rays_from_json(model.config.rays_json)
>>> grid = model.config.grid
>>> lbl = naive_fusion_anisotropic_grid(dists, probs, rays, grid=grid)
```

merge_stardist_masks.naive_fusion.**naive_fusion_isotropic_grid**(*dists: ndarray[Any, dtype[float64]]*, *probs: ndarray[Any, dtype[float64]]*, *rays: Rays_Base | None = None*, *prob_thresh: float = 0.5*, *grid: int = 2*, *no_slicing: bool = False*, *max_full_overlaps: int = 2*, *erase_probs_at_full_overlap: bool = False*, *show_overlaps: bool = False*, *respect_probs: bool = False*) → ndarray[Any, dtype[uint16]] | ndarray[Any, dtype[int32]]

Merge overlapping masks given by dists, probs, rays.

Performs a naive iterative scheme to merge the masks that a StarDist network has calculated to generate a label image. This function can perform 2D and 3D segmentation. For 3D segmentation *rays* has to be passed from the StarDist model.

Parameters

- **dists** (*ndarray[Any, dtype[float64]]*) – 3- or 4-dimensional array representing distances of each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x, n_rays)`, for 3D predictions it is `(len_z, len_y, len_x, n_rays)`.
- **probs** (*ndarray[Any, dtype[float64]]*) – 2- or 3-dimensional array representing the probabilities for each mask as outputed by a StarDist model. For 2D predictions the shape is `(len_y, len_x)`, for 3D predictions it is `(len_z, len_y, len_x)`.
- **rays** (*Rays_Base | None*) – For 3D predictions *rays* must be set otherwise a `ValueError` is raised. It should be the `Rays_Base` instance used by the StarDist model.
- **prob_thresh** (*float*) – Only masks with probability above this threshold are considered.

- **grid** (*int*) – This is the grid information about the subsampling occuring in the StarDist model.
- **no_slicing** (*bool*) – For very big and winded objects this should be set to `True`. However, this might result in longer calculation times.
- **max_full_overlaps** (*int*) – Maximum no. of full overlaps before current object is treated as finished.
- **erase_probs_at_full_overlap** (*bool*) – If set to `True` probs are set to -1 whenever a full overlap is detected.
- **show_overlaps** (*bool*) – If set to true, overlaps are set to `-1`.
- **respect_probs** (*bool*) – If set to true, overlapping elements are overwritten by considering their probabilities. Only works when show_overlaps is 'false'.

Returns

The label image with uint16 labels. For 2D, the shape is `(len_y * grid[0], len_x * grid[1])` and for 3D it is `(len_z * grid[0], len_y * grid[1], len_z * grid[2])`.

Raises

ValueError – If *rays* is `None` and 3D inputs are given or when `probs.ndim != len(grid)`. # noqa: DAR402 ValueError

Return type

ndarray[*Any*, *dtype*[*uint16*]] | *ndarray*[*Any*, *dtype*[*int32*]]

Example

```
>>> from merge_stardist_masks.naive_fusion import naive_fusion_isotropic_grid
>>> from stardist.rays3d import rays_from_json
>>> probs, dists = model.predict(img)  # model is a 3D StarDist model
>>> rays = rays_from_json(model.config.rays_json)
>>> grid = model.config.grid[0]
>>> lbl = naive_fusion_isotropic_grid(dists, probs, rays, grid=grid)
```

merge_stardist_masks.naive_fusion.**no_slicing_slice_point**(*point: numpy.typing.ArrayLike*, *max_dist: int*) → Tuple[Tuple[slice, ...], ndarray[Any, dtype[int64]]]

Convenience function that returns the same point and tuple of slice(None).

Parameters

- **point** (*numpy.typing.ArrayLike*) –
- **max_dist** (*int*) –

Return type

Tuple[*Tuple*[slice, ...], *ndarray*[*Any*, *dtype*[*int64*]]]

merge_stardist_masks.naive_fusion.**paint_in_with_overlaps**(*paint_in: ndarray[Any, dtype[T]]*, *shape: ndarray[Any, dtype[bool_]]*, *paint_id: int*) → ndarray[Any, dtype[T]]

Set entries of array paint_in to paint_id or -1 if not free anymore.

Parameters

- **paint_in** (*ndarray[Any, dtype[T]]*) –

- **shape** (*ndarray[Any, dtype[bool_]]*) –
- **paint_id** (*int*) –

Return type
ndarray[*Any*, *dtype*[*T*]]

merge_stardist_masks.naive_fusion.**paint_in_without_overlaps**(*paint_in: ndarray[Any, dtype[T]]*, *shape: ndarray[Any, dtype[bool_]]*, *paint_id: int*) → ndarray[Any, dtype[T]]

Set entries of array to paint_id according to boolean values in shape.

Parameters

- **paint_in** (*ndarray[Any, dtype[T]]*) –
- **shape** (*ndarray[Any, dtype[bool_]]*) –
- **paint_id** (*int*) –

Return type
ndarray[*Any*, *dtype*[*T*]]

merge_stardist_masks.naive_fusion.**paint_in_without_overlaps_check_probs**(*paint_in: ndarray[Any, dtype[T]]*, *shape: ndarray[Any, dtype[bool_]]*, *old_probs: ndarray[Any, dtype[float32]]*, *new_probs: ndarray[Any, dtype[float32]]*, *paint_id: int*) → Tuple[ndarray[Any, dtype[T]], ndarray[Any, dtype[float32]]]

Set and overwrite entries of array to paint_id respecting their probabilities.

Parameters

- **paint_in** (*ndarray[Any, dtype[T]]*) –
- **shape** (*ndarray[Any, dtype[bool_]]*) –
- **old_probs** (*ndarray[Any, dtype[float32]]*) –
- **new_probs** (*ndarray[Any, dtype[float32]]*) –
- **paint_id** (*int*) –

Return type
Tuple[*ndarray*[*Any*, *dtype*[*T*]], *ndarray*[*Any*, *dtype*[*float32*]]]

merge_stardist_masks.naive_fusion.**points_from_grid**(*shape: Tuple[int, ...]*, *grid: Tuple[int, ...]*) → ndarray[Any, dtype[int64]]

Generate array giving out points for indices.

Parameters

- **shape** (*Tuple[int, ...]*) –
- **grid** (*Tuple[int, ...]*) –

Return type
ndarray[*Any*, *dtype*[*int64*]]

merge_stardist_masks.naive_fusion.**poly_list_with_probs**(*dists_: numpy.typing.ArrayLike*, *points_: numpy.typing.ArrayLike*, *probs_: numpy.typing.ArrayLike*, *shape: Tuple[int, ...]*, *poly_list_func: merge_stardist_masks.naive_fusion.PolyToLabelSignature*) → Tuple[ndarray[Any, dtype[int64]], ndarray[Any, dtype[float32]]]

Return labels and according probabilities.

Parameters

- **dists_** (*numpy.typing.ArrayLike*) –
- **points_** (*numpy.typing.ArrayLike*) –
- **probs_** (*numpy.typing.ArrayLike*) –
- **shape** (*Tuple[int, ...]*) –
- **poly_list_func** (*merge_stardist_masks.naive_fusion.PolyToLabelSignature*) –

Return type
Tuple[*ndarray*[*Any*, *dtype*[*int64*]], *ndarray*[*Any*, *dtype*[*float32*]]]

merge_stardist_masks.naive_fusion.**slice_point**(*point: numpy.typing.ArrayLike*, *max_dist: int*) → Tuple[Tuple[slice, ...], ndarray[Any, dtype[int64]]]

Calculate the extents of a slice for a given point and its coordinates within.

Parameters

- **point** (*numpy.typing.ArrayLike*) –
- **max_dist** (*int*) –

Return type
Tuple[*Tuple*[slice, ...], *ndarray*[*Any*, *dtype*[*int64*]]]

9.4 Contributor Guide

Thank you for your interest in improving this project. This project is open-source under the MIT license and welcomes contributions in the form of bug reports, feature requests, and pull requests.

Here is a list of important resources for contributors:

- Source Code
- Documentation
- Issue Tracker
- Code of Conduct

9.4.1 How to report a bug

Report bugs on the Issue Tracker.

When filing an issue, make sure to answer these questions:

- Which operating system and Python version are you using?
- Which version of this project are you using?
- What did you do?
- What did you expect to see?
- What did you see instead?

The best way to get your bug fixed is to provide a test case, and/or steps to reproduce the issue.

9.4.2 How to request a feature

Request features on the Issue Tracker.

9.4.3 How to set up your development environment

You need Python 3.7+ and the following tools:

- Poetry
- Nox
- nox-poetry

Install the package with development requirements:

```
$ poetry install
```

You can now run an interactive Python session, or the command-line interface:

```
$ poetry run python
$ poetry run merge-stardist-masks
```

9.4.4 How to test the project

Run the full test suite:

```
$ nox
```

List the available Nox sessions:

```
$ nox --list-sessions
```

You can also run a specific Nox session. For example, invoke the unit test suite like this:

```
$ nox --session=tests
```

Unit tests are located in the `tests` directory, and are written using the pytest testing framework.

9.4.5 How to submit changes

Open a pull request to submit changes to this project.

Your pull request needs to meet the following guidelines for acceptance:

- The Nox test suite must pass without errors and warnings.
- Include unit tests. This project maintains 100% code coverage.
- If your changes add functionality, update the documentation accordingly.

Feel free to submit early, though—we can always iterate on this.

To run linting and code formatting checks before committing your change, you can install pre-commit as a Git hook by running the following command:

```
$ nox --session=pre-commit -- install
```

It is recommended to open an issue before starting work on anything. This will allow a chance to talk it over with the owners and validate your approach.

9.5 Contributor Covenant Code of Conduct

9.5.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

9.5.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

9.5.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

9.5.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

9.5.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at niknett@gmail.com. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

9.5.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

9.5.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

9.6 MIT License

PYTHON MODULE INDEX

m

merge_stardist_masks.naive_fusion, 24

INDEX

G

get_poly_list_to_label() (*in module merge_stardist_masks.naive_fusion*), 24
get_poly_to_label() (*in module merge_stardist_masks.naive_fusion*), 24

I

inflate_array() (*in module merge_stardist_masks.naive_fusion*), 24

M

merge_stardist_masks.naive_fusion module, 24
mesh_from_shape() (*in module merge_stardist_masks.naive_fusion*), 24
module merge_stardist_masks.naive_fusion, 24
my_polygons_list_to_label() (*in module merge_stardist_masks.naive_fusion*), 24
my_polygons_to_label() (*in module merge_stardist_masks.naive_fusion*), 25
my_polyhedron_list_to_label() (*in module merge_stardist_masks.naive_fusion*), 25
my_polyhedron_to_label() (*in module merge_stardist_masks.naive_fusion*), 25

N

naive_fusion() (*in module merge_stardist_masks.naive_fusion*), 26
naive_fusion_anisotropic_grid() (*in module merge_stardist_masks.naive_fusion*), 27
naive_fusion_isotropic_grid() (*in module merge_stardist_masks.naive_fusion*), 28
no_slicing_slice_point() (*in module merge_stardist_masks.naive_fusion*), 29

P

paint_in_with_overlaps() (*in module merge_stardist_masks.naive_fusion*), 29
paint_in_without_overlaps() (*in module merge_stardist_masks.naive_fusion*), 30
paint_in_without_overlaps_check_probs() (*in module merge_stardist_masks.naive_fusion*), 30
points_from_grid() (*in module merge_stardist_masks.naive_fusion*), 30
poly_list_with_probs() (*in module merge_stardist_masks.naive_fusion*), 31

S

slice_point() (*in module merge_stardist_masks.naive_fusion*), 31