# CristalX

## *Release 1.1.0*

**Zoltan Csati**

**Jul 01, 2021**

# GETTING STARTED

Global intro

# **CRISTALX**

Identification of individual grains in microscopic images

*CristalX* is a Python package that helps in the analysis of polycrystalline microstructures. Its name originates from the French word 'cristal', corresponding to the English word 'crystal'.

## 1.1 Features

- Image segmentation to identify the grains in a microstructure
- Analysis tools for the segmented image
- Explicit geometrical representation of the grains
- Interacting with meshes created on the microstructure
- Mapping fields between a mesh and the grid of DIC measurements
- Simulation tools for the inverse problem arising from a combined numerical-experimental method (in progress . . . )
- Visualization and data exchange

## 1.2 Getting help

1. Read the documentation.
2. Check the existing issues. They may already provide an answer to you question.
3. Open a new issue.

## 1.3 Contributing

Read the `docs/source/contributing.md` file.

## 1.4 Citing *CristalX*

We have an article freely available on SoftwareX, showing the background and the design of *CristalX*.

When using *CristalX* in scientific publications, please cite the following paper:

- Csati, Z.; Witz, J.-F.; Magnier, V.; Bartali, A. E.; Limodin, N. & Najjar, D. *CristalX: Facilitating simulations for experimentally obtained grain-based microstructures*. SoftwareX, **2021**, *14*, 100669

BibTeX entry:

```
@Article{Csati2021,
  author    = {Zoltan Csati and Jean-Fran{\c{c}}ois Witz and Vincent Magnier and
→Ahmed El Bartali and Nathalie Limodin and Denis Najjar},
  journal   = {{SoftwareX}},
  title     = {{CristalX}: {F}acilitating simulations for experimentally obtained
→grain-based microstructures},
  year      = {2021},
  month     = jun,
  pages     = {100669},
  volume    = {14},
  doi       = {10.1016/j.softx.2021.100669},
}
```

# INSTALLATION

*CristalX* is easy to install; you can get started in minutes.

## 2.1 Dependencies

*CristalX* is a written in pure Python, although it relies on packages that use other languages (mostly C and C++). However, it is not a problem from the user's perspective as no manual compilation is needed. Those who are interested can see the dependencies in the `environment.yml` file in the root directory.

Whether you install *CristalX* by *conda* or try it online, all but one dependency is installed. In a common workflow, that single missing dependency will not affect you. For details, see . . .

## 2.2 Try it online

If you just want to get a taste of *CristalX*, you can try it online without installing anything.

Click here to jump to the root directory. Existing Jupyter notebooks are in the `notebooks/` directory. You can modify them and create new ones. If you directly want to open the main tutorial, which describes a real-world application, click here.

## 2.3 Install it locally

If you want more control (debugging, inspecting variables, etc.) over *CristalX* or if you wish to contribute to the project, it is recommended to install it on your machine.

### 2.3.1 Obtain the source

If you are a user of *CristalX*, the best is to get the latest release:

- download it from GitHub

- or clone it with Git:

```
git clone https://github.com/CsatiZoltan/CristalX.git
cd CristalX
git checkout v<version_number>
```

where <version_number> is the version you want to use. E.g. if you want to use version 1.0.1, you need to type git checkout v1.0.1. See the available tags, corresponding to the published releases, for the possibilities.

Note that in this case, you will be in a "detached HEAD" state, meaning that the HEAD does not point to a branch but to the specific tag. Any commit you make in this state will not be associated with a branch. Therefore, if you want to develop or contribute to *CristalX*, check out the *master* branch (see the next paragraph).

If you want to develop *CristalX* or simply want to have access to the latest features, you need to fetch the latest state:

- download it from GitHub

- or clone it with Git:

```
git clone https://github.com/CsatiZoltan/CristalX.git
```

### 2.3.2 Install with *conda*

All you need to have is the *conda* package manager. Open a terminal (or a command prompt if you are under Windows) in the root directory and type

```
conda env create -f environment.yml
```

*CristalX* has been installed to a separate environment, so you can safely work inside it. Activate that environment:

```
conda activate CristalX
```

Once you have finished working with *CristalX*, either close the terminal or type

```
conda deactivate
```

to return to your default *conda* environment.

If you want to uninstall *CristalX*, make sure that the CristalX conda environment is not active and then

```
conda env remove -n CristalX
```

When uninstalled, conda env list will not show it.

### 2.3.3 Install with *pip*

If you do not have *conda* or you prefer *pip*, you can also install *CristalX* by typing

```
pip install -r requirements.txt
```

assuming that you have Python installed. It is highly recommended that you first create a new virtual environment to make sure you do not break your Python installation. However, an important component of *CristalX* (functions that rely on *PythonOCC*) will **not** be installed if you choose *pip*. The reason for this is that *PythonOCC* is not (yet) available on PyPI.

# DOCUMENTATION

The documentation for *CristalX* is available both locally and online. It is generated by Sphinx and consists of the code documentation and the various guides (including the one you are currently reading). If you want to extend the documentation, please read the *corresponding section* in the Developer guide.

## 3.1 Local documentation

It resides in the `docs/` directory. Open a terminal there and type `make html`. The output will be written to `docs/build/`. Open `docs/build/html/index.html` to land on the home page of the documentation.

## 3.2 Online documentation

For every *push event*, the documentation is rebuilt by Read the Docs. The latest HTML version is available on https://cristalx.readthedocs.io/en/latest. Read the Docs is configured to build a PDF and EPUB output as well. All three formats can be downloaded for offline use if you click on the bottom of the left sidebar. For convenience, here are the direct links: HTML, PDF, EPUB.

## 3.3 Using the HTML documentation

For browsing on a computer, the HTML documentation is the most convenient and feature-rich.

The side-bar on the left provides quick navigation.

Code example boxes have two toggle buttons. Clicking on the prompt button >>> hides the prompt and the output. If you then click on the copy icon next to it, you can copy the contents of the box, making it easy to try the code chunks in your Python environment.

```
>>> mesh = TriMesh(np.array([[0, 0], [1, 0], [2, 0], [0, 2], [0, 1], [1, 1]]),
...                np.array([[0, 1, 5], [4, 5, 3], [5, 4, 0], [2, 5, 1]]))
>>> edges = mesh.get_edges()
>>> edges
{(0, 1): [0], (1, 5): [0, 3], (5, 0): [0, 2], (4, 5): [1, 2], (5, 3): [1],
(3, 4): [1], (4, 0): [2], (2, 5): [3], (1, 2): [3]}
>>> mesh.plot(cell_labels=True, vertex_labels=True)
>>> plt.show()
```

```
mesh = TriMesh(np.array([[0, 0], [1, 0], [2, 0], [0, 2], [0, 1], [1, 1]]),
               np.array([[0, 1, 5], [4, 5, 3], [5, 4, 0], [2, 5, 1]]))
edges = mesh.get_edges()
edges

mesh.plot(cell_labels=True, vertex_labels=True)
plt.show()
```
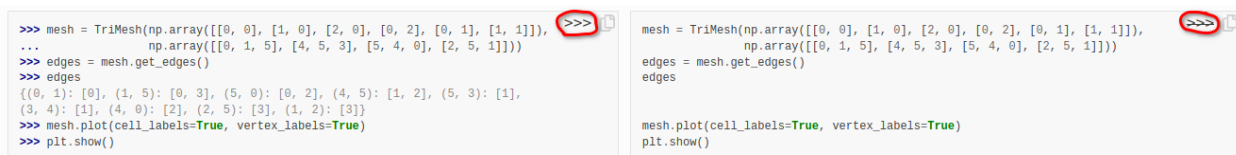
A direct link can be obtained for most sections, including the code documentation, by moving the cursor to the right

of the text and then clicking on the chain icon.

When browsing the API documentation and want to look at the implementation, you can easily access the source code



by clicking on the `[source]` tag.

# HOW TO USE THE CODES

If you want to get to know the classes and functions, the best is to check the *API documentation*. Most components are amply documented and contain one or more examples. If you want to see them in action, check out our *workflow* for a real-world application. If you are curious about the underlying algorithms, consult with the *Algorithms* section.

## 4.1 Tutorials

The tutorials are available in two forms: Python scripts and Jupyter notebooks.

### 4.1.1 Python scripts

The scripts are intended to be run in batch mode. They need to be run in a specific order as the subsequent scripts rely on the output of the previous ones. These scripts are useful if you are already a bit familiar with *CristalX* or if you wish to adapt the scripts to your needs. Indeed, if you intend to extend the package, the safest way is to copy the relevant scripts and modify them. The scripts are named as run_moduleName, where moduleName is the name of the module in the grains/ directory that the script mainly relies on. In some way, run_moduleName acts as a demonstration of what the moduleName module is used for. The scripts are meant to be run as modules, i.e. navigate to the root of the project and type

```
python -m scripts.moduleName
```

### 4.1.2 Jupyter notebooks

Almost the same workflow is available as a single Jupyter notebook, located at the nootebooks/ directory. this allows you to interactively discover *CristalX* through an example. You can easily rerun parts of the code to see the effects of the parameters, and the rich output is embedded into the same document. If you are a novice Python user or just want to have a taste of *CristalX*, notebooks are the recommended way to get started.

## 4.2 Undo changes

When you experiment with *CristalX*, you will probably change parameter values. As the scripts communicate by reading and writing data, the original data that come with *CristalX* will be overwritten. There are multiple ways to undo the changes.

### 4.2.1 In Binder

As written in the *Installation instructions*, you can try *CristalX* online without the need to install anything on your computer. Then a separate virtual environment is created. Whatever changes you make there, they will not influence your local installation (if you have) or the data on the GitHub repository.

### 4.2.2 In a local installation

If you downloaded *CristalX* from GitHub, you can simply replace the new files with the original ones. In case you cloned *CristalX* with Git, you can easily discard the changes you make.

# A DETAILED WORKFLOW

*CristalX* contains several modules that facilitate experimental and numerical works on polycrystalline microstructures. The usage of these modules is demonstrated on a complex example.

## 5.1 Problem formulation

Grain-based microstructures occur in nature but also develop during industrial processes. We are interested in how the size and the distribution of grains, and the material they are made up of influence the resistance of train wheels and axles to fatigue loading. The problem requires that

- the individual grains can be distinguished on a microscopic image
- a good quality mesh is generated on the microstructure
- the numerical solution obtained on this mesh can be compared to full-field measurements

The first two steps are discussed here.

## 5.2 Initial setup

First, we switch to the root directory of the project so that the package imports are consistent. Note that the relative path always works because the current directory in case of a Jupyter notebook is always the directory in which the notebook is.

```python
[1]: import os
     os.chdir('../')
     main_dir = os.path.abspath(os.curdir)
```

Let us set the directory that will hold our data.

```python
[2]: data_dir = os.path.abspath('data')
     try:
         os.mkdir(data_dir)
     except FileExistsError:
         pass
```

We now copy the necessary data files that come with *CristalX* into this freshly created directory.

```python
[3]: from os.path import join
     from shutil import copy
     selected_files = ['1.png', '1_cropped.png', 'splinegons.png', '1_mesh_extended.npz']
```

(continues on next page)

```
for file in selected_files:
    copy(join('scripts/data', file), data_dir)
```

In interactive mode, debug information are displayed. They are not useful for presentation purposes as they pollute the rendered output. Similarly for runtime warnings.

```
[4]: import logging
     logging.basicConfig(level=logging.ERROR)
     import warnings
     warnings.simplefilter('ignore')
```

For the figures to be visible in this notebook, we define a utility function to scale them.

```
[5]: import matplotlib.pyplot as plt
     def scale_figure(dpi):
         plt.gcf().set_dpi(dpi)
```

## 5.3 Identify grains in a microstructure

The picture below was taken with a digital camera and shows the central part of a tensile specimen. In this section, we will identify the individual grains in the image.

```
[6]: from skimage.io import imshow
     imshow(join(data_dir, '1.png'))
     scale_figure(150)
```
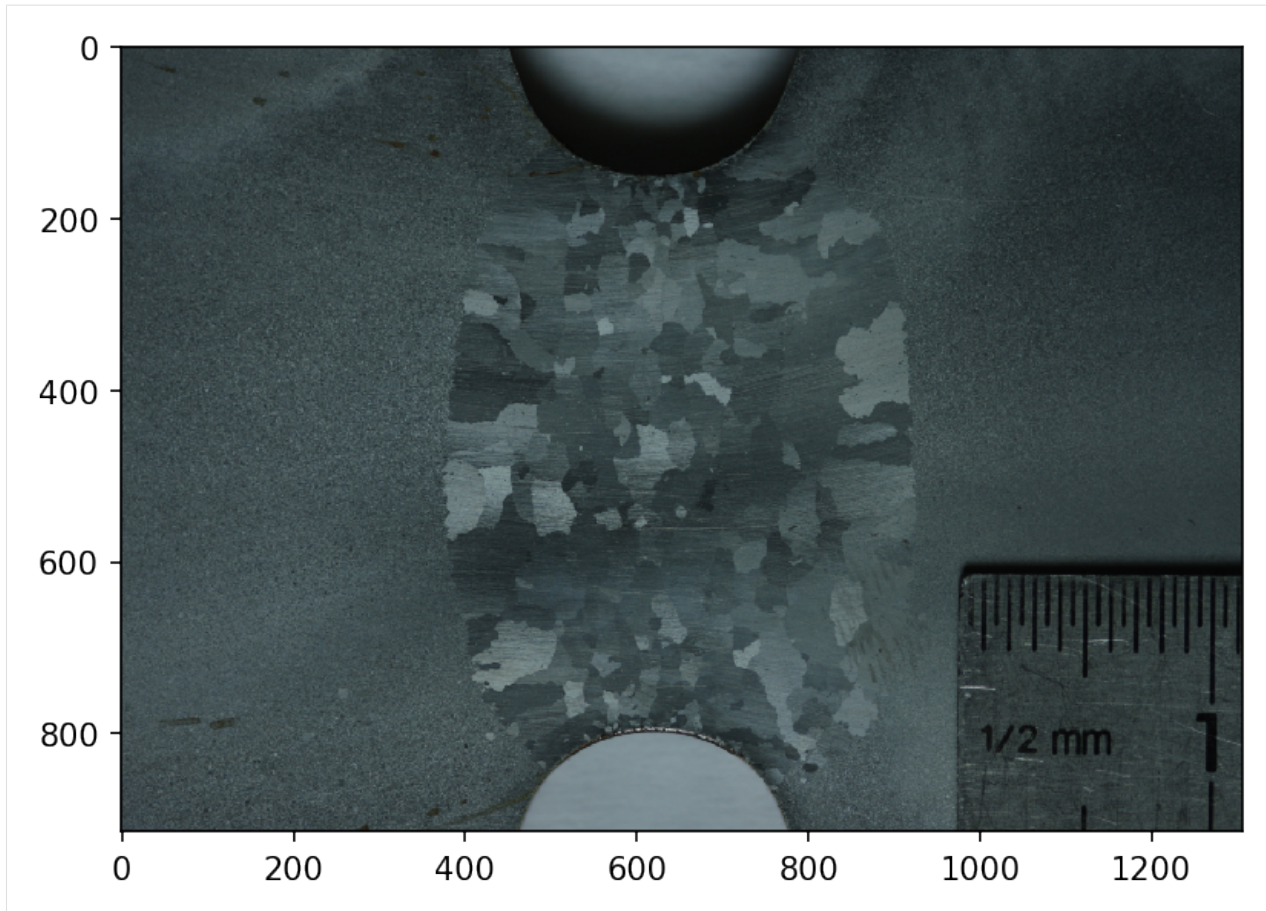
So that the surrounding region does not interfere with the segmentation, we will work on a cropped region.

```
[7]: from skimage.io import imread
     import numpy as np
     image = join(data_dir, '1_cropped.png')
```

As you can see, the RGB image is represented as a 3D numpy array.

```
[8]: image_as_matrix = imread(image)
     type(image_as_matrix), image_as_matrix.shape
```

```
[8]: (numpy.ndarray, (638, 589, 3))
```

However, you do not explicitly have to deal with this matrix as the `Segmentation` class will handle it.

The `Segmentation` class offers a set of methods to perform the consecutive steps of the segmentation workflow. We pass to the constructor the path to the image to be segmented. By default, the resulting images are shown at the end of each step. In batch mode, you probably want to set the `interactive_mode` optional parameter to `False`. But in this interactive document, we want to see the outputs.

```
[9]: from grains.segmentation import Segmentation
     GS = Segmentation(image)
```

```
Image successfully loaded.
```
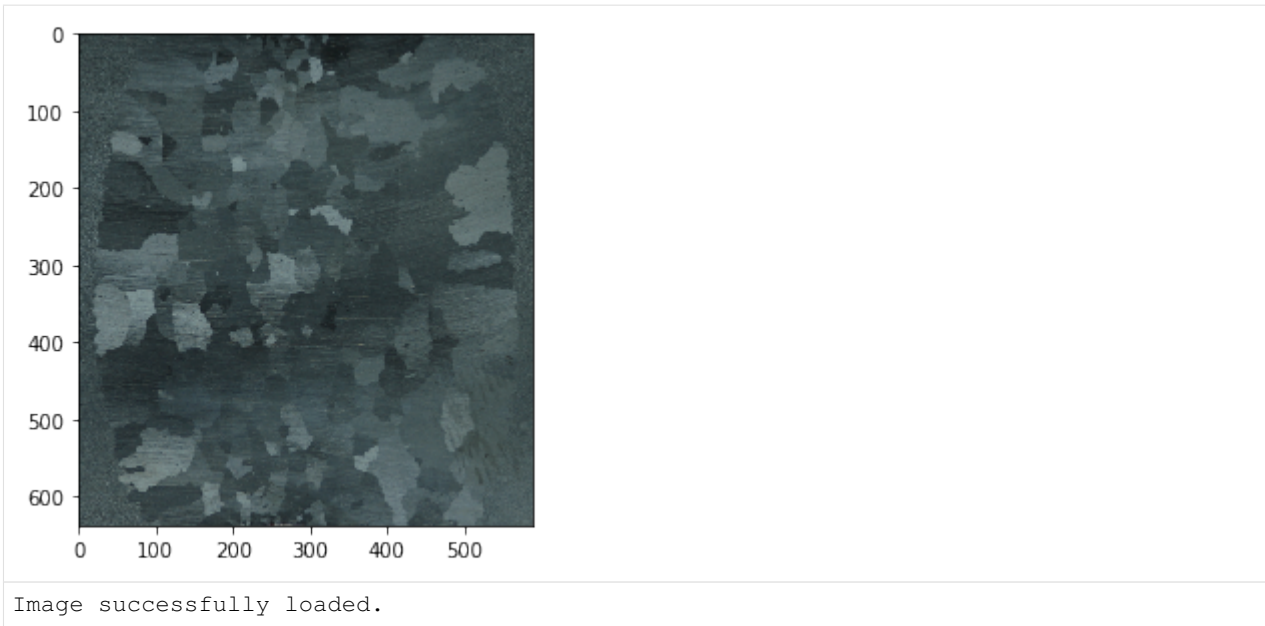
After the image has been loaded and stored in the `GS` object, we want to remove the noise. Since the interfaces among the grains are important for us, we use median filtering for smoothing as it preserves the contours. The smoothing is governed by the *window size*. The larger this number, the smoother the processed image is, and the higher the computational cost becomes.

```
[10]: filter_window_size = 5
      filtered = GS.filter_image(filter_window_size)
```



```
Median filtering finished.
```
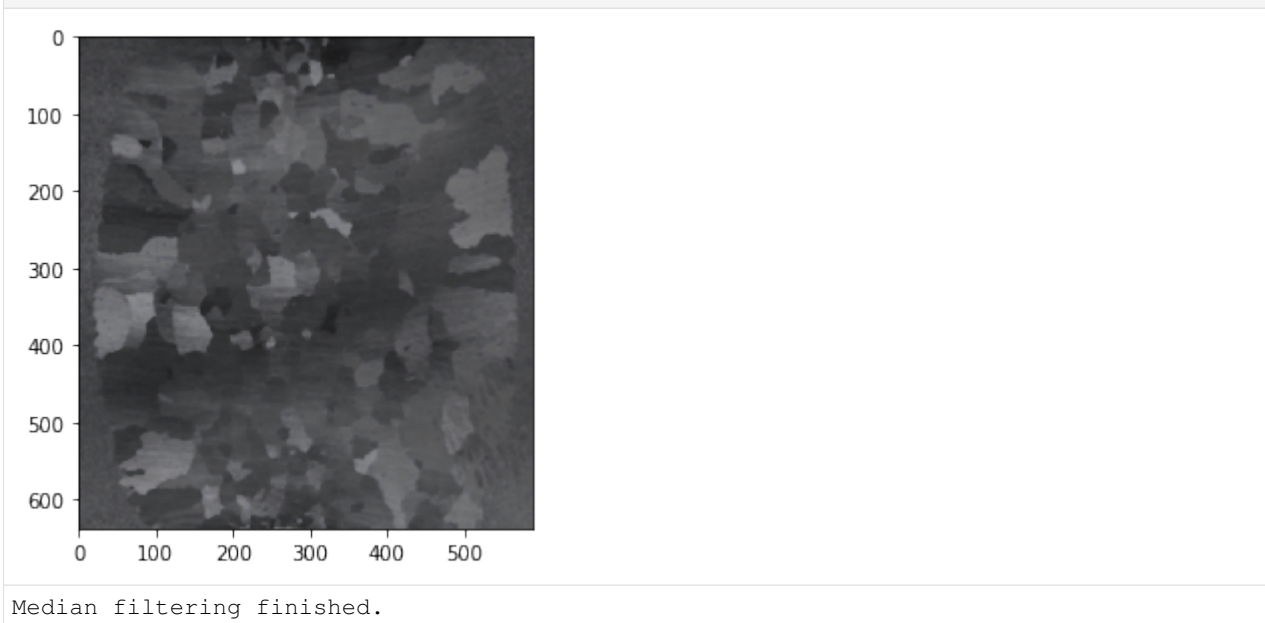
To improve the segmentation result later, we perform *superpixel segmentation* as an initial step. During this process, the image is subdivided into superpixels, groups of pixels that belong together based on colour, spatial distance or other properties. On this image, a superpixel algorithm called *Quick Shift* proved to be the best among the algorithms available in *scikit-image*.

```
[11]: segmented = GS.initial_segmentation(filtered)
```
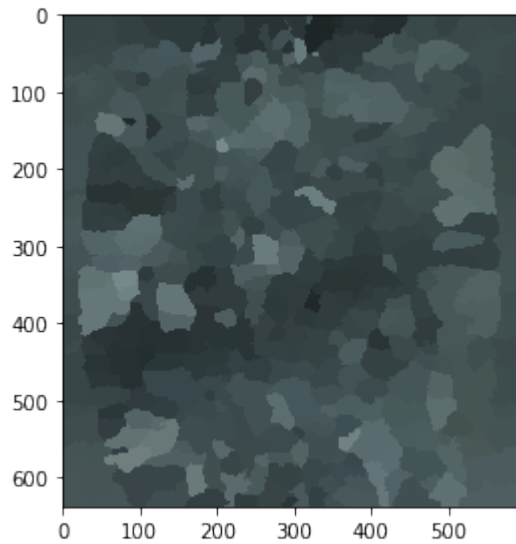


```
Quick shift segmentation finished. Number of segments: 421
```

The result is indeed a segmented (also called labelled) image. Distinct positive integers, called labels, are associated to groups of pixels in the labelled image and each pixel belongs to one and only one such group.

```
[12]: segmented
```

```
[12]: array([[  8,   8,   8, ...,   2,   2,   2],
             [  8,   8,   8, ...,   2,   2,   2],
             [  8,   8,   8, ...,   2,   2,   2],
             ...,
             [400, 400, 400, ..., 414, 414, 414],
             [400, 400, 400, ..., 414, 414, 414],
             [400, 400, 400, ..., 414, 414, 414]])
```

The segmented image has the same size as the original image, but it is not an RGB image any more.

```
[13]: segmented.shape, image_as_matrix.shape
```

```
[13]: ((638, 589), (638, 589, 3))
```

We can visualize a labelled image by associating different colours to the different labels.

```
[14]: from skimage.color import label2rgb
      from numpy.random import random
      nlabel = len(np.unique(segmented))
      imshow(label2rgb(segmented, colors=random((nlabel, 3))))
      scale_figure(150)
```
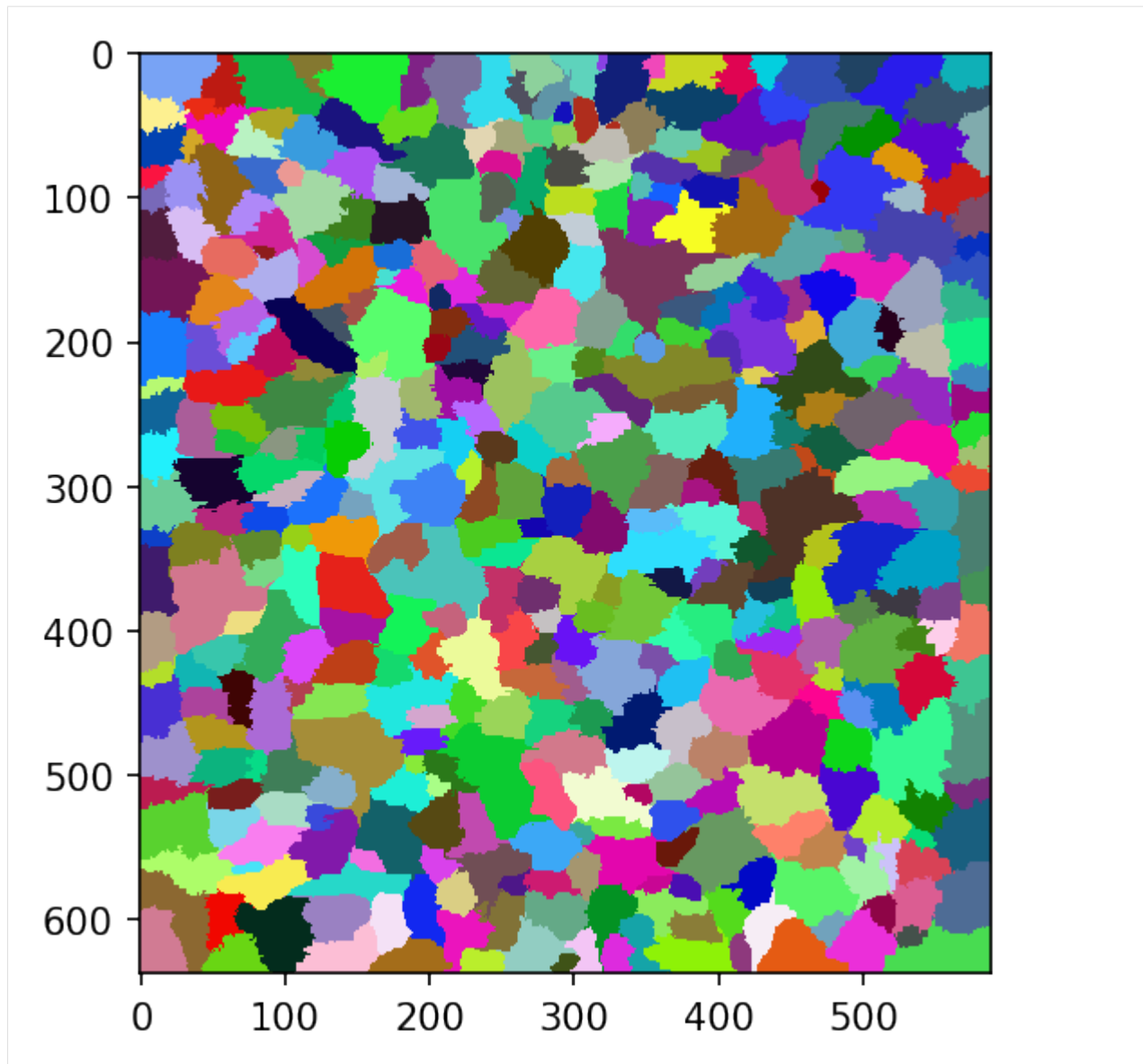
As the superpixel segmentation in the previous step resulted in an oversegmented image, the region adjacency graph is constructed and used to merge some of the neighbouring superpixels based on their similarity with respect to mean colour. Regions connected by edges with smaller weights than a prescribed threshold are combined. As you can see, the number of supersegments decreases.

```
[15]: cluster_merging_threshold = 7
      reduced = GS.merge_clusters(segmented, threshold=cluster_merging_threshold)
```

```
Tiny clusters merged. Number of segments: 177
```

We need a few more steps before succesfully applying another segmentation technique to obtain the final segmented image. First, we detect the grain boundaries. They are shown superposed on the original image.

```
[16]: boundary = GS.find_grain_boundaries(reduced)
```



```
Grain boundaries found.
```
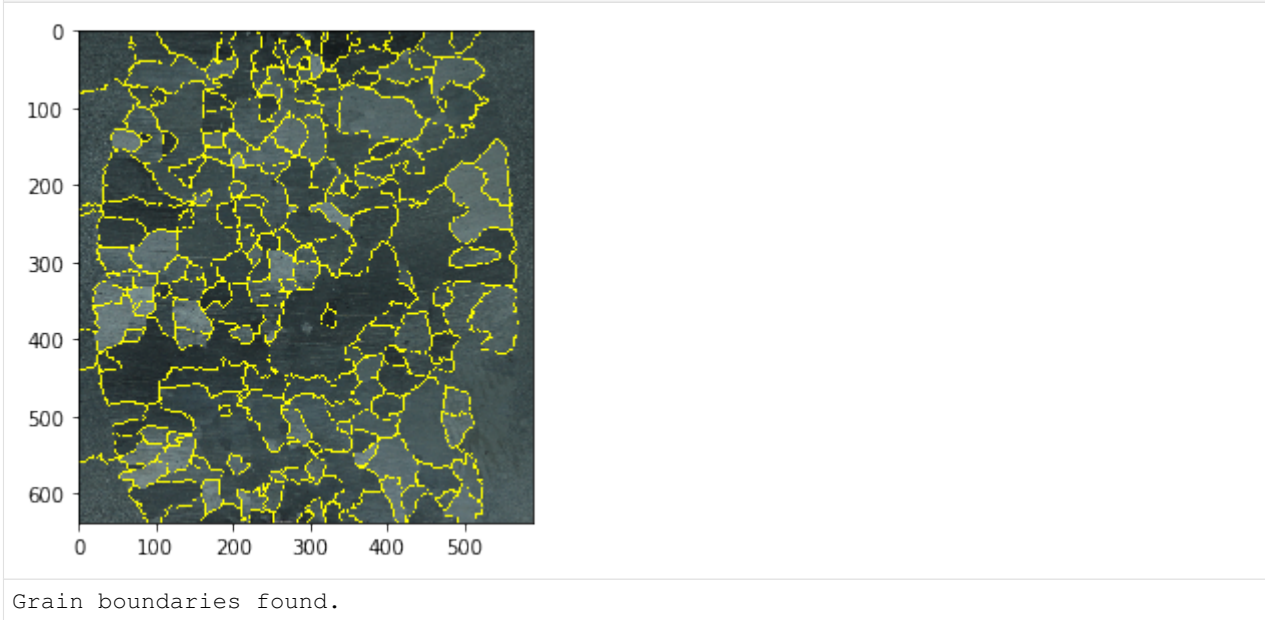
The boundary image is a binary image in which the `True` values indicate the boundaries among the labelled regions.

```
[17]: boundary
```

```
[17]: array([[False, False, False, ..., False, False, False],
             [False, False, False, ..., False, False, False],
             [False, False, False, ..., False, False, False],
             ...,
             [False, False, False, ..., False, False, False],
```

**5.3. Identify grains in a microstructure** 17

```
        [False, False, False, ..., False, False, False],
        [False, False, False, ..., False, False, False]])
```

The we use thinning on the grain boundary to obtain a single-pixel wide skeleton.

```
[18]: skeleton = GS.create_skeleton(boundary)
```
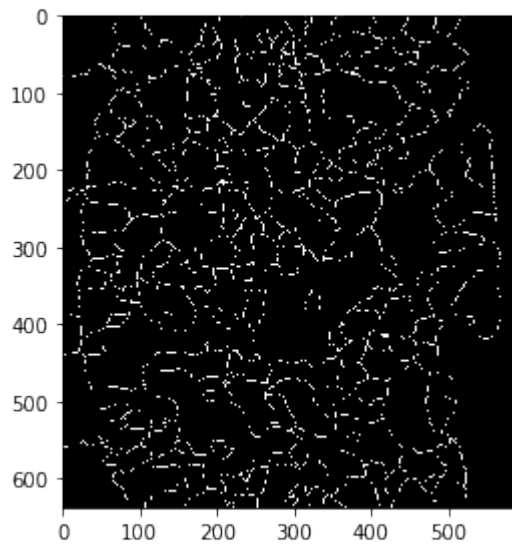


```
Skeleton constructed.
```

If the automatic segmentation carried out so far is not good enough, the user can manually edit the skeleton as a graph in ImagePy. The combination of the automatic segmentation with human supervision is a powerful way to achieve good results in a relatively short amount of time. Here, we stay with the automatic method.

To recover the segments again, we use *watershed segmentation* on the skeleton. Had it been directly applied on the original image, the result of the watershed segmentation would have been an oversegmented image due to the noises that act as local minima. The success of the watershed segmentation depends on how well the catchment basins are identified, which are the locations where the flooding starts. The so-called marker-based watershed segmentation methods rely on markers (computed automatically or given by the user), i.e. the location of the catchment basins, as inputs. In the watershed_segmentation method of the Segmentation class, various transformations are used to obtain a desirable outcome.

```
[19]: watershed = GS.watershed_segmentation(skeleton)
```

```
Watershed segmentation finished. Number of segments: 247
```
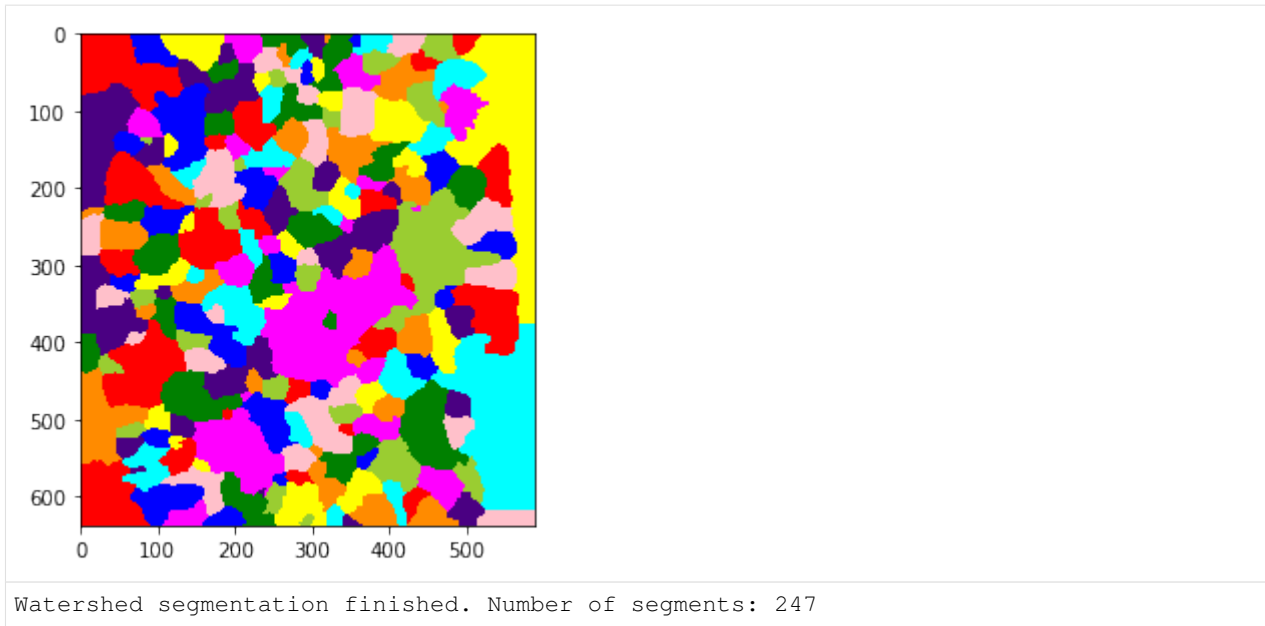
To the left and to the right of the central zone, artificial "grains" appear. This is not a problem, they can either be merged, or left as it is and associate the same material to them.

## 5.4 Extending the image domain

The segmented image corresponds to the recrystallized part of the specimen. When we perform the traction test numerically we need a larger domain, the boundary on which the boundary conditions are applied. The size of this extended domain does not have to match with the physical size of the tensile specimen, but it has to be sufficiently large so that the far-field boundary conditions do no not influence the deformation state in the central region.

As a first step, we extend the domain by padding the corresponding numpy array. Since we want to handle the added domains (grains) on the left and on the right as separate grains, we associate different labels to them. There are about 250 grains in the central region, so choosing labels 500 and 501 ensures that these regions have unique labels.

The extended domain is constructed in such a way that it matches the region of the DIC measurements. This will be useful when comparing the results of the simulation with the experimental data.

```python
[20]: from grains.simulation import change_domain
extended_image = change_domain(watershed, 0.4395, 0, 0, 0, 500)
extended_image = change_domain(extended_image, 0, 0.5568, 0, 0, 501)
imshow(extended_image)
scale_figure(150)
```
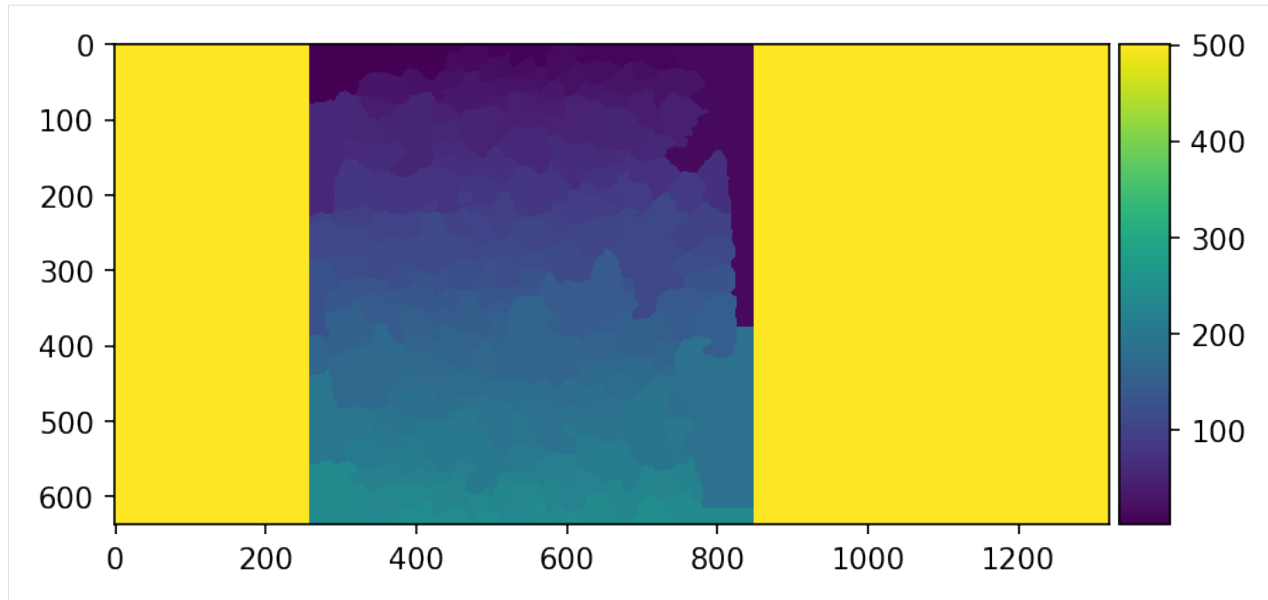
## 5.5 From image to geometry

In this section, we show how to represent the grains as an explicit geometry. See our paper for the motivation.

For grain representation, we can choose between polygon and splinegon approximations. We will base the meshing on the splinegon representation, but the polygons are also shown. The high-level functions `polygonize` and `splinegonize` wrap all the important algorithms. You can see from the code below that they share similar function signature. The only difference is that the spline parameters can be provided to the `splinegonize` function.

```
[21]: from grains.cad import polygonize, search_neighbor, splinegonize
polygons = polygonize(extended_image, search_neighbor(2, np.inf), connectivity=1)
splinegons, _ = splinegonize(extended_image, search_neighbor(2, np.inf),
→connectivity=1, degree_min=3, degree_max=3, continuity='C0', tol=1)
```

Let us plot the grains. For polygons:

```
[22]: from grains.cad import plot_polygons
plot_polygons(list(polygons.values()));
scale_figure(150)
```
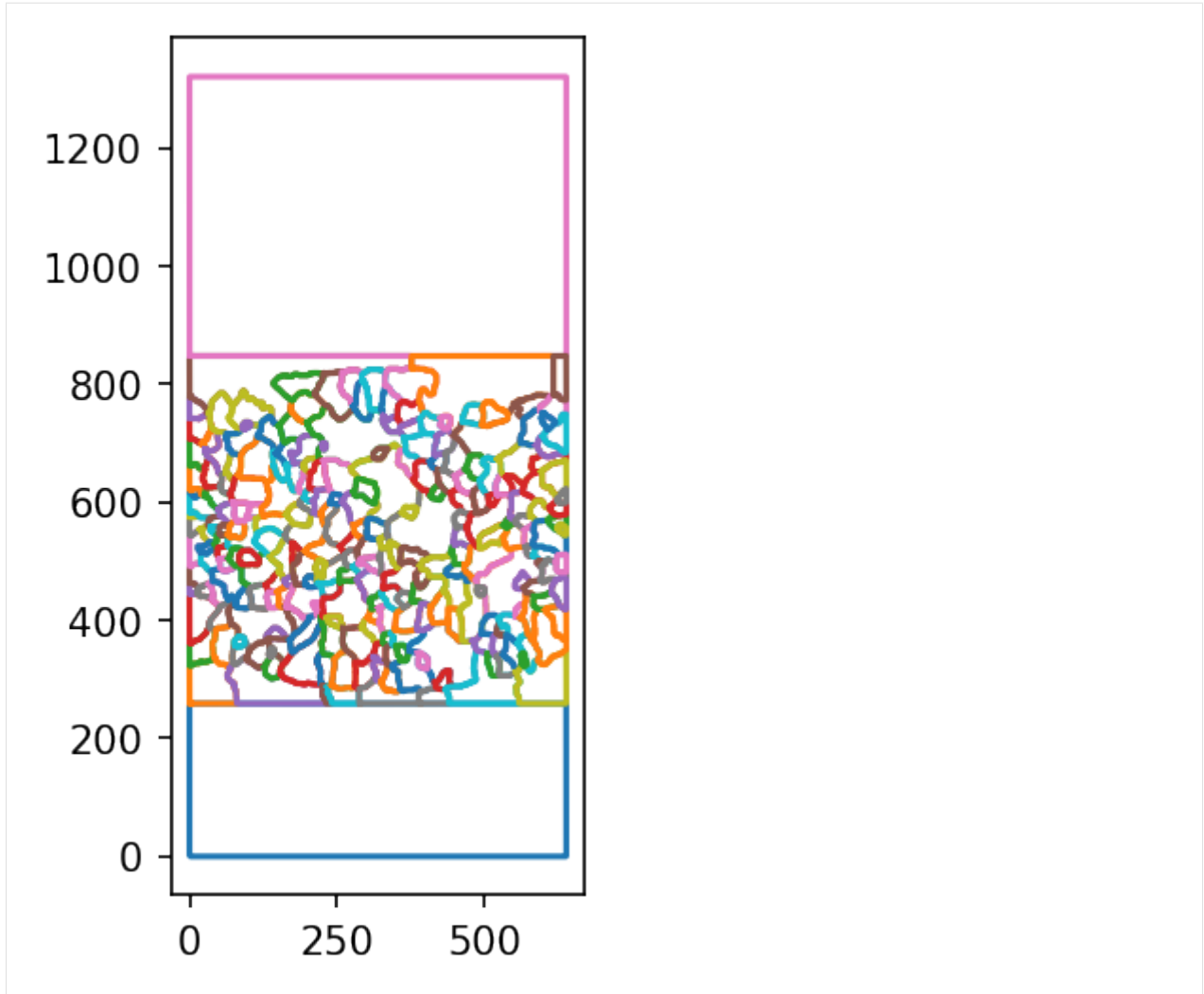
Plotting the spline surfaces will bring up a window.

```
[23]: from grains.cad import plot_splinegons
      from OCC.Display.SimpleGui import init_display
      plot_splinegons(list(splinegons.values()), color=(0, 0, 1))
```

```
Layer manager created
Layer dimensions: 1024, 768
```

To allow reading this document statically, here is a screenshot of the window that popped up.

```
[24]: fig, ax = plt.subplots(dpi=500)
      imshow(join(data_dir, 'splinegons.png'), ax=ax)
      ax.set_axis_off()
```
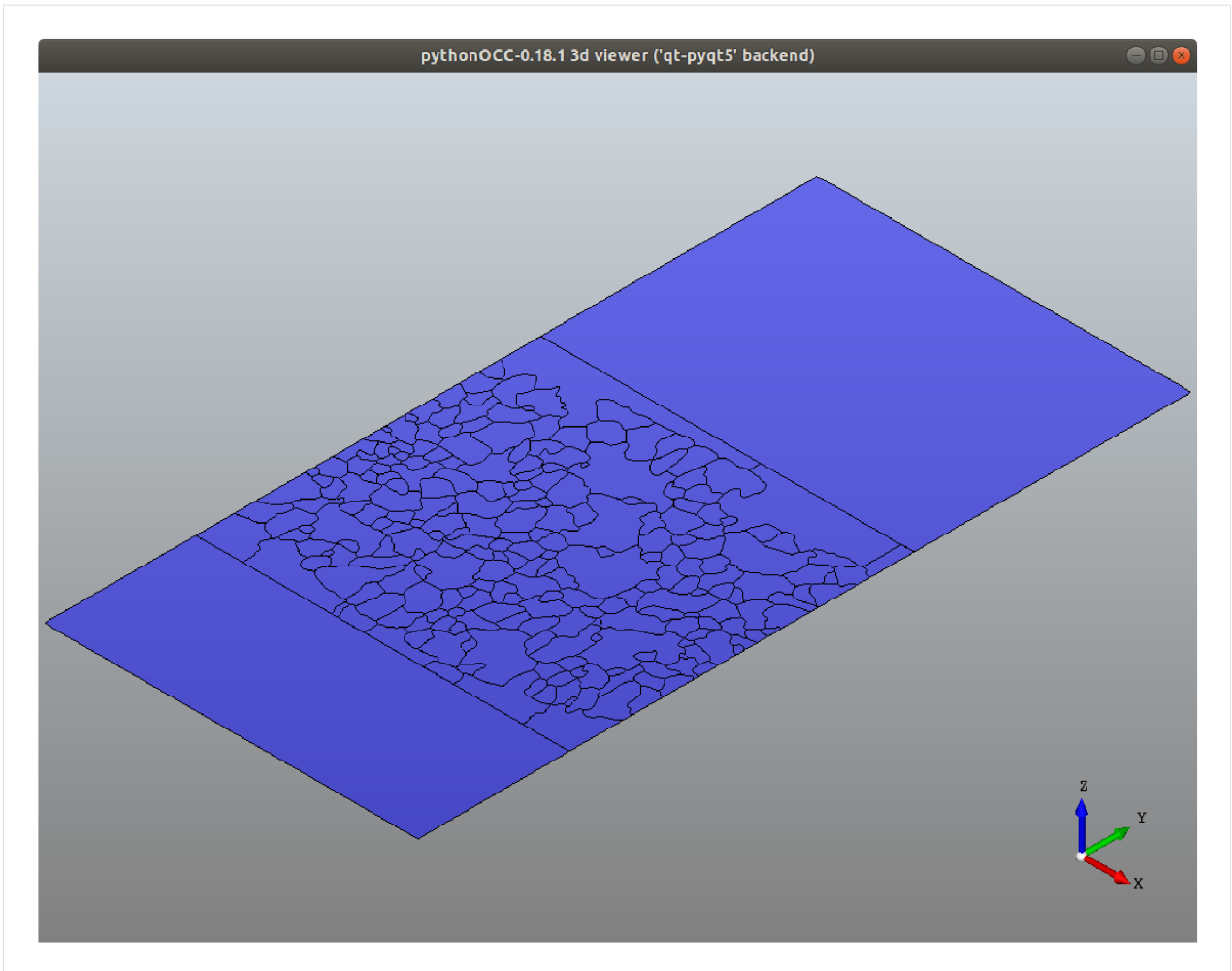
Once we have the spline surfaces, we write them to a STEP file. This allows us to edit the geometry in a CAD program and to generate a mesh for the grains.

```
[25]: from grains.cad import regions2step
      regions2step(list(splinegons.values()), join(data_dir, 'microstructure.stp'))
```

## 5.6 Repairing the geometry

For the sample microstructure, one grain was not identified, i.e. it contains a hole. The hole can be filled in by creating a new grain. We have a detailed guide on how to do this.

## 5.7 Mesh generation

The same guide linked in the previous section continues with the mesh generation. It happens that we constructed the conforming mesh in Salome, and then exported it to a .med file. Using the med module of *CristalX*, the triangular mesh cells for each grain and the boundary nodes have been extracted. We also have a tutorial on how to process a .med file.

## 5.8 Modifying the mesh

First, we obtain the mesh data that was saved using the `med` module.

```
[26]: mesh_file = join(data_dir, '1_mesh_extended.npz')
      with np.load(mesh_file, allow_pickle=True) as mesh:
          nodes = mesh['nodes']
          elements = mesh['elements']
          element_groups = mesh['element_groups']
          node_groups = mesh['node_groups']
      # Retrieve groups, which were stored in dictionaries (https://stackoverflow.com/a/
      ↪40220343/4892892)
      element_groups = element_groups.item()
      node_groups = node_groups.item()
```

The array `elements` contains the label of the nodes in the mesh.

```
[27]: elements
```

```
[27]: array([[  603,    604,   4311],
             [  604,     19,    605],
             [  606,     18,   4312],
             ...,
             [13377, 13369, 13374],
             [13372, 13377, 13373],
             [13373, 13370, 13372]])
```

The array `nodes` holds the coordinates of each node of the mesh.

```
[28]: nodes
```

```
[28]: array([[2.50000000e-01, 2.59000000e+02],
             [8.00000000e+01, 2.59250000e+02],
             [2.31000000e+02, 2.59250000e+02],
             ...,
             [6.28062814e+02, 8.02235833e+02],
             [6.22855688e+02, 7.95968504e+02],
             [6.23756131e+02, 7.89612749e+02]])
```

The elements are available for each grain. For instance, grain 87 is labelled as *Face_87* and it contains the following elements.

```
[29]: element_groups['Face_87']
```

```
[29]: array([23231, 23232, 23233, 23234, 23235, 23236, 23237, 23238, 23239,
             23240, 23241, 23242, 23243, 23244, 23245])
```

We mentioned earlier that some grains are artefacts of the watershed segmentation. In other words, they are outside the recrystallized central region. Therefore, those grains along with the two extensions (that were given the labels 500

and 501) were merged in Salome and a single mesh was constructed on the merged region. That region contains the most elements.

```
[30]: len(element_groups['homogeneous'])
```

```
[30]: 3718
```

To be able to prescribe boundary conditions later, the boundary nodes were also exported from Salome.

```
[31]: node_groups
```

```
[31]: {'bottom': array([   8,   14,  448,  452,  467,  468,  470,  471,  472,  474,  475,
          477,  478,  479,  480,  481,  482,  495,  496,  497,  498,  499,
          557,  558,  559,  560,  561,  562,  563,  564,  565, 3673, 3674,
         3675, 3676, 3706, 3707, 3708, 3839, 3840, 3841, 3842, 3851, 3852,
         3861, 3862, 3863, 3864, 3865, 3866, 3867, 3874, 3880, 3881, 3882,
         3883, 3884, 3885, 3886, 3887, 3888, 3895, 3896, 3897, 3914, 3915,
         3922, 3923, 3924, 3925, 3926, 3930, 3931, 3932, 3933, 3943, 3944,
         3945, 3951, 3952, 3953, 3954, 3955, 3956, 3957, 3958, 3959, 3960,
         3961, 3962, 3963, 3964], dtype=int32),
       'left': array([  9,  10, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493,
         494], dtype=int32),
       'right': array([ 15,  16, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555,
         556], dtype=int32),
       'top': array([   0,   11,   17,   20,   21,   25,   28,   33,   34,   36,   44,
          45,   46,   52,   61,   63,  541,  542,  543,  544,  594,  595,
         596,  597,  598,  599,  600,  601,  602,  609,  653,  654,  655,
         656,  673,  674,  675,  676,  677,  678,  679,  702,  703,  704,
         705,  706,  707,  724,  725,  749,  750,  751,  759,  760,  761,
         762,  763,  764,  812,  813,  814,  815,  816,  817,  827,  828,
         829,  830,  831,  832,  848,  849,  850,  851,  896,  897,  898,
         920,  921, 1637, 1638, 1639, 1640], dtype=int32)}
```

Handling all these data individually is tedious. Hence, we created a mesh class that bundles them and defines operations on them. The `TriMesh` class is an abstraction for a triangular mesh. First, we initialize it with the elements and the nodes that were loaded from the *1_mesh_extended.npz* file.

```
[32]: from grains.geometry import TriMesh
      mesh = TriMesh(nodes, elements)
```

The `TriMesh` class has several options to plot the mesh, for now we consider the default plotting.

```
[33]: mesh.plot()
      scale_figure(150)
```
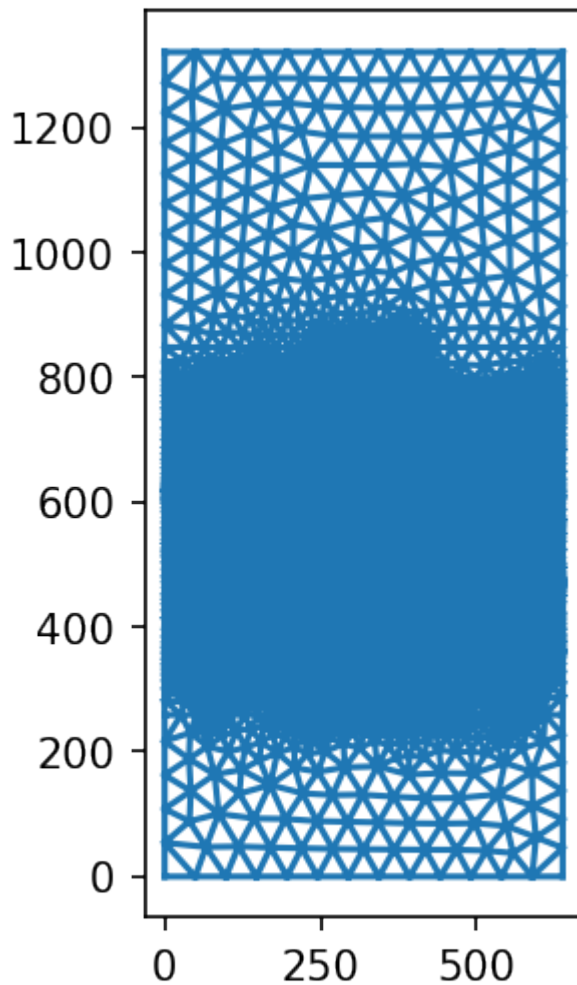
Then the element and node sets are associated to the mesh object.

```
[34]: # Associate the element and node groups to it
      for group_name, elements in element_groups.items():
          mesh.create_cell_set(group_name, elements)
      for group_name, nodes in node_groups.items():
          mesh.create_vertex_set(group_name, nodes)
```

Mesh-based discretization methods often presume that the node ordering within an element is counter-clockwise. To make sure this is the case, we explicitly enforce it.

```
[35]: mesh.change_vertex_numbering('ccw', inplace=True);
```

The mesh was created on the geometry, which was constructed based on the image, i.e. in pixel units. For the simulation to be comparable with the experimental data (full-field measurement), the real size of the tensile speciment must be given. In the first image of this notebook, you can observe a ruler. With the help of it, we set the correspondance between the physical unit and the pixel unit: there are 29.55 pixels in 1 mm.

```
[36]: mesh.scale(1/29.55, inplace=True);
```

For convenience, we rotate the specimen so that its axis is parallel to the horizontal tensile loading.

```
[37]: from math import pi
      mesh.rotate(-pi/2, inplace=True);
```

Let us check if this is what we wanted.

```
[38]: mesh.plot(vertex_legends=True)
      scale_figure(150)
```
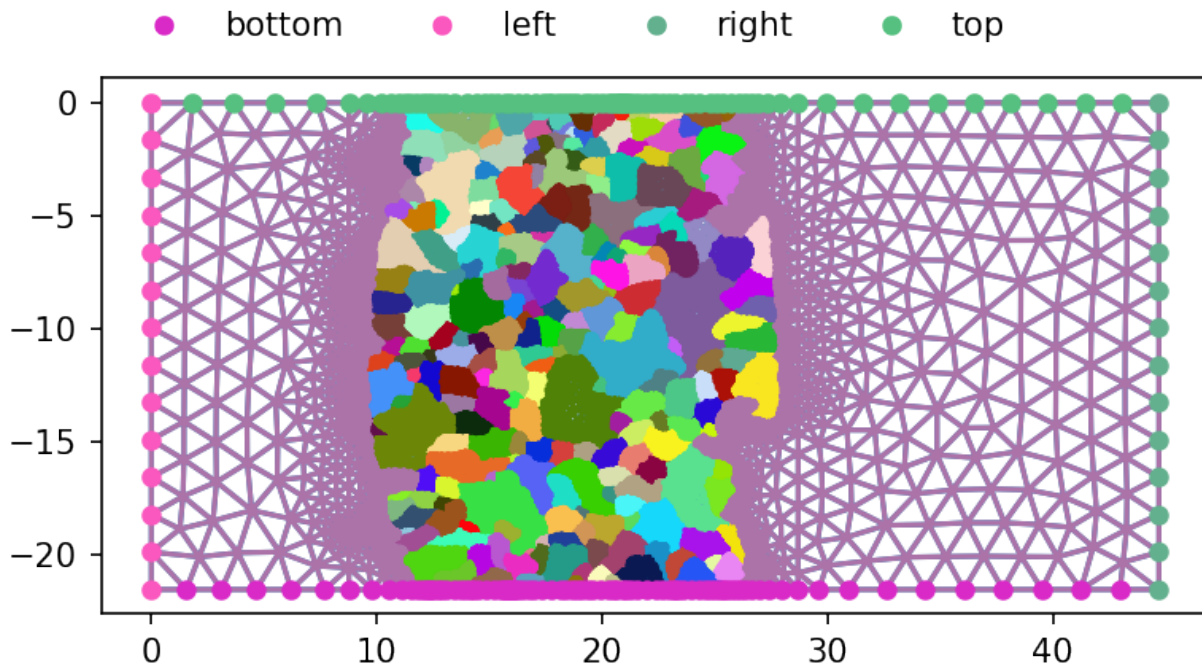


We are satisfied with the resulting transformations, so let us save them.

```
[39]: np.savez_compressed(join(data_dir, '1_mesh_extended_scaled.npz'),
                          nodes=mesh.vertices, elements=mesh.cells,
                          element_groups=mesh.cell_sets, node_groups=mesh.vertex_sets)
```

## 5.9 Conclusions

We successfully obtained a good quality mesh starting from an image.

In the near future, we will solve an inverse problem to identify characteristic parameters in the constitutive model. This optimization task requires the comparison of the numerical solution obtained on the mesh with the measured field values acquired through digital image correlation (DIC). Some of this work is already available in the `dic` and `simulation` modules.

This document is meant to serve as a global overview of what *CristalX* can be used for. Feel free to play with the parameters to investigate their effects. The functions and classes of *CristalX* provide many more functionalities. Browse the documentation to familiarize yourself with the details.

---

The initially created directory (`data/`) is now deleted.

---

```
[40]: from shutil import rmtree
      try:
          rmtree(data_dir)
      except FileNotFoundError:
          pass
```

# GEOMETRY AND MESH PROCESSING IN SALOME

The *cad* module, as shown in the *Algorithms* section, allows to approximate a segmented image with planar spline surfaces (splinegons) and those surfaces can be written to a STEP file. Now, we demonstrate on the sample microstructure how to repair the geometry and generate a conforming mesh on the splinegons. All the manipulations are done in Salome 9.4.0.

## 6.1 Geometry

1. Import the geometry To import a STEP file (obtained by writing the splinegons into a STEP file), select the *Geometry* module and then *File -> Import -> STEP*.

2. Repair the geometry

   For the sample microstructure, one grain has not been identified, i.e. it contains a hole. To fill the hole, we perform the following steps (always select the object that was created the last time):

   1. *Repair -> Close Contour* and select the (in our sample) two boundary curves of the unidentified grain.

   2. *Repair -> Suppress Holes* and select the two boundary curves you selected in the previous step. Salome informs you that a face will be created in place of the hole.

   3. *Repair -> Limit tolerance* and set the tolerance to 1e-3.

   4. *Repair -> Sewing* and set 1e-2 for the tolerance.

3. Extract the grains We will need the mesh on each grain, therefore, we extract the grain faces by "exploding" the microstructure using *New Entity -> Explode*. In the dialog box, select the sewn geometry (end result of the geometry repairing workflow above) as *Main Object* and *Face* as *Sub-shapes Type*. Salome properly identifies the 250 sub-shapes, i.e. the grains.

4. Fix the "artificial" grains

   The sample microstructure describes the central part of a tensile specimen. However, to adhere to the Saint-Venant principle, the loading is exerted further from this central zone. Hence, we created long enough regions in the direction of the prescribed load by attaching two rectangular faces to the central region. These faces are not precise rectangles because the boundary of the microstructure does not consist of straight segments. Nevertheless, from now on, we will use the term rectangle to describe the extended region. To achieve perfect matching between the rectangle and the boundary of the microstructure, one side of the rectangle must contain the same lines as the boundary of the microstructure. The other three sides will be straight line segments.

   1. We want to extract the boundary lines of the microstructure. To do that, we explode the boundary grains into edges with *New Entity -> Explode*. The *Main Object* in the dialog box is a selected boundary grain, and the *Sub-shape Type* is *Edge*. Do this with all the boundary grains.

   2. Create the three sides of the rectangle

1. Create the four vertices of the rectangle with *New Entry -> Basic -> Point*. Two of these points are the extremities of the microstructure, the other two are given such that

   - the rectangle's three sides become parallel to the coordinate axes (so that boundary conditions are easily prescribed)

   - the length of the longer side of the rectangle is long enough compared to the microstructure (we applied a ratio of 5)

2. Connect these vertices with three lines, which forms the three sides of the rectangle: *New Entity -> Build -> Edge*.

3. Create the rectangle by connecting the three line segments constructed in the previous step with the boundary lines of the fourth side, obtained by exploding the boundary grains: *New Entity -> Build -> Wire*.

4. Delete the original "artificial" grain as it will be replaced by the new one.

5. Create the surface enclosed by the rectangle: *New Entity -> Build -> Face*, and select the wire created before. Perform steps 1-5 for the other rectangle (on the other side of the microstructure) as well.

6. In order to create a conforming mesh on the whole domain (i.e. on the microstructure and on the two rectangles), we need to create a compound surface: *New Entity -> Build -> Compound* and select the grains plus the two rectangles.

7. It is not enough to create the mesh, we also need to know which elements belong to which grains. To allow this, we explode the compound surface: select the compound object created in the previous step as the *Main Object* in *New Entity -> Explode*. The *Sub-shape Type* is *Face*, as before.

The geometry is now impeccable, let us start meshing.

## 6.2 Mesh

Select the *Mesh* module.

1. Create the mesh

   1. Choose *Mesh -> Create Mesh* from the menu. The geometry on which the mesh will be created is the compound surface.

   2. The mesh type in this study is *Triangular* and the algorithm is *NETGEN 1D-2D*. Choose *NETGEN 2D Parameters* as a hypothesis and experiment with the settings to suit your needs. Accept the changes.

   3. Choose *Mesh -> Compute* to generate the mesh with the chosen settings.

   If you want to alter the mesh, change the hypothesis and compute the new mesh.

2. Obtain the sub-meshes for the grains

   The segmented image contains regions that are not part of the recrystallized region, but they belong to the homogeneous region. We want to handle them in the same way as the two artificial grains. We do not need to merge the surfaces, just put them in the same element group. Therefore, we create

   - a common element group for the elements lying in the homogeneous region

   - one element group for each grain in the heterogeneous (recrystallized) region

   using *Mesh -> Create Groups from Geometry* and selecting all the 250 faces of the geometry. To form a single group for the heterogeneous region, merge the corresponding element groups in *Mesh -> Union Groups*. Once done, delete the groups that were used in forming the union.

3. We will explicitly need to prescribe boundary conditions on the left and right sides of the rectangular domain. For this, the nodes on those sides are collected with *Mesh -> Create Group* and set the *Elements Type* to *Node*. Select the nodes on the left edge and give the node set a name. Repeat it for the nodes on the right.

4. Export the mesh Click on *File -> Export -> MED file*. The sub-meshes (element groups, node groups) are also exported.

The mesh is ready for further processing. We have another *guide* that discusses how to handle the MED file from within Python.

# PROCESSING A .MED FILE

After exporting the mesh from Salome to a MED file, we may want to perform certain operations on it. The MED-Coupling tool of Salome provides C++ and Python APIs for this purpose. However, that requires the user to

- have Salome installed as those APIs are available from the Salome kernel

- get to know the API

Moreover, it can happen that some mesh processing functionalities they may want to use does not exist. Since meshes consisting of cells of the same type (e.g. triangles) can be represented as homogeneous and contiguous arrays, converting the mesh from MED to numpy arrays seems a reasonable choice. This is what our *med* module does: it provides a thin wrapper around *MEDCoupling* to extract the mesh and the defined groups (cell and vertex groups) from the MED file and convert them to numpy arrays. This way, the user who deals with numerical modelling can implement their mesh processing algorithms based on numpy arrays, which is fast and straightforward. Furthermore, the person who performs the CAD operations and has Salome installed, can use our *med* module to export the mesh to numpy arrays so that the numerical analyst can directly work on it without having to have Salome installed and without any knowledge on the *MEDCoupling* API.

If you want to know more about the implementation details, read the documentation for the *med module*.

## 7.1 Using the module

To use our *med* module, access to the *MEDLoader* module from Salome is required. In the following, we assume that Salome has been installed and added to the path, so the command `salome` is available. If you

- want to use the Python REPL:

```
$ salome shell -- python
Python 3.6.5 (default, Dec 16 2019, 16:42:15)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Now, you have access to the *MEDCoupling* Python API. E.g.

```
>>> from MEDLoader import MEDFileData
```

- want to use the PyCharm IDE

```
$ salome shell "path_to_PyCharm/bin/pycharm.sh"
```

This will start PyCharm. In the IDE (PyCharm in this example), set the interpreter to that of Salome's Python. For me, it is located at `BINARIES-UB18.04/Python/bin/python3`, where `UB18.04` refers to the fact that I downloaded a pre-compiled Salome binaries for Ubuntu 18.04.

To learn more about the `salome` command, read the manual.

# ALGORITHMS

The goal of this document is to provide more low-level algorithmic details than what is given in our paper. In fact, the material presented here is complimentary to the paper. The paper concentrates on the intuition behind the methods and shows the high-level structure, also citing the relevant literature.

## 8.1 From image to geometry

It is recommended to read Section 2.3 of the paper first and then this section.

The algorithm to construct a geometrical representation out of a segmented image is complex. To reduce the complexity, we created composable parts. We start with some notions below. Then the main steps of the algorithm is discussed in the upcoming subsections.

Grains can have different *representations*. When represented as an image (input in our algorithm), it is a set of pixels labelled with the same positive integer. A grain can be given different geometrical representations. It can be the assembly of primitive shapes, usually triangles. One can also describe a grain by its boundary. For example, piecewise linear boundary segments lead to a *polygon*, connecting spline curves lead to a *splinegon*. A grain can also be represented as a *cycle of a graph*.

We make distinction between *topological* and *geometrical* data. Topological data includes connectivity, neighbourhood and membership information. Coordinates of points are examples for geometrical data. Separating these two terms allows us to build abstractions in the code, making it understandable and extensible.

### 8.1.1 Skeleton of the image

Based on the segmented/labelled image, *skan* builds the skeleton network, as demonstrated on a sample image below. The different pixel colors represent different labels.

As a preprocessing step, we create an additional labelled region that surrounds the original labelled image. The reason is that this way, *skan* will create branches along the boundary of the original image, which will close the boundary grains. It also simplifies the algorithms we will use later because the boundary interfaces can be handled in the same way as the internal ones: every interface separates exactly two grains.

## 8.1.2 Which branches form a grain?

The question arises: given the branches, how to reconstruct the grains? First, let us try a graph theoretic approach. In the example below, we want to identify the three grains denoted by roman numerals. The network of branches can be represented as a graph, in which the vertices are the end points and the edges correspond to the branches.

Adequately selected cycles in this graph would give the grains. However, as discussed in the paper, determining the grains based on the graph exclusively would be very challenging because of the following characteristics.

- For general microstructures, the graph contains multiple edges (see the example above), which rules out many graph processing methods.

- Finding all the elementary cycles in the graph is too costly for graphs coming from realistic microstructures. Even if we found all the cycles, we would need a criterion to choose which ones correspond to grains. E.g. the cycle $\bar{6} - \bar{2} - \bar{3} - \bar{5}$ does not encompass a single grain but the union of two grains.

- Another technique to find the correct cycles would be the minimum cycle basis. However, this basis is not unique so there is no guarantee that we find the adequate cycles.

The solution is to use both topological and geometrical information. The skeleton is superimposed on the labelled image (from which it was constructed by *skan*) and the labels around a given skeleton node or end point is detected. Staying with the example in the *first subsection*, the scheme is shown below.

In the following, we describe an algorithm to find out which two grains are incident to a branch (remember that *always two grains neighbor a branch*).

### Algorithm

The neighbor search around every node of a branch is performed and the two most common labels are chosen. Since in the image representation the grain is a set of pixels having the same label, the selected two labels give the neighboring two grains to a branch.

### Neighbor definitions

As described in the paper, certain scenarios necessitate to consider various neighborhood definitions. For those definitions, see the documentation of the `grains.utils.neighborhood` function.

Going back to the first configuration in the beginning of this section, the algorithm gives the following branch-grain connectivities.

$$\bar{1} : [\text{I}]$$
$$\bar{2} : [\text{II}]$$
$$\bar{3} : [\text{I}, \text{II}]$$
$$\bar{4} : [\text{III}, \text{II}]$$
$$\bar{5} : [\text{I}, \text{III}]$$
$$\bar{6} : [\text{II}, \text{I}]$$

Inverting this relationship gives the grain-branch connectivities:

$$\text{I} : [\bar{1}, \bar{3}, \bar{5}, \bar{6}]$$
$$\text{II} : [\bar{2}, \bar{3}, \bar{6}, \bar{4}]$$
$$\text{III} : [\bar{5}, \bar{4}]$$

The grain-branch connectivities are an intermediate representation (topological-geometrical). It is independent of how we geometrically represent a grain later.

### 8.1.3 Grains as oriented planar surfaces

The previous part of the reconstruction algorithm determined which branches bound a grain. In order to obtain a surface representation of a grain, the boundary must be oriented and hence the branches must be connected in the appropriate order. The following figure demonstrates for grain II the working of a brute-force algorithm.

The branches are interlaced based on their common junctions. The arrows show which branches follow in order. Note that the default orientation of branches $\bar{3}$ and $\bar{4}$ needs to be swapped.

Finally, we arrive at a fully geometrical description because each grain is now given by a series of points (nodes and end points) along its boundary.

### 8.1.4 Geometrical representations of grains

Now that we have a list of points, we can build two geometrical representations of a grain. In the *polygon representation*, the list of points are the consecutive vertices of the polygon. Their coordinates are stacked, the first vertex being repeated to "close" the polygon. In the *spline representation*, the list of points on each branch act as a knots of a B-spline. Once the bounding splines have been constructed, the planar spline surface (splinegon) is spanned by those bounding splines.

# PROGRAM DESIGN

*CristalX* is not a black-box library such as BLAS, neither is a GUI-based application intended for end-users. It is rather an easy-to-use and extensible set of Python codes that provide the basic functionalities that scientists can extend based on their needs. The following ideas were kept in mind while writing and maintaining *CristalX*.

- Driven by actual needs

Only implement features that are currently used. Adding extra features requires more testing, possibly more dependencies and therefore code bloat, and increases the cognitive load of the user. Instead, the emphasis is on creating a stable minimum core library that can be easily extended according to users' demands. Consequently, application code is separated from the core modules.

- Build on well-established packages

We rely on the scientific Python stack: *NumPy* for array manipulations, *SciPy* for interpolation and some other computations, *Matplotlib* for visualization and *scikit-image* for image processing. This ensures interoperability with other scientific codes and that our software is hopefully bug-free.

- Minimize the dependencies

Rapid prototyping is essential in scientific code development and Python is an excellent choice to satisfy this requirement. At the same time, relying on fast libraries ensures that the computations are reasonably fast. The libraries mentioned in the previous point are easy to install, often already pre-installed in certain Python distributions.

- High-quality documentation

Future contributors will benefit from the rich documentation. Python doctests are extensively used, serving both as test cases and as examples of usage. The docstrings conform to the *numpydoc* style guide.

- We strive for decoupling the modules

Although part of the *grains* package, if the modules are independent, they can be reused in other projects too just by copy-pasting the required functions.

- Do not overuse classes

In the prototyping phase, prefer using free functions to methods. As an idea evolves, you will naturally find data and algorithms that belong together, and can refactor free functions into member functions of a class.

- Do not use deep hierarchies

Initially, stay away from excessive nesting to avoid fragmenting the code base. If the project grows big, you can still refactor the code by introducing deeper hierarchies. Deep nesting causes unnecessary cognitive load and it also makes the code more verbose at the caller's site. Compare

```python
import package1.package2.module
```

with

```
import package.module
```

- Gradually refactor code

As more and more features are added to the project, we will often find that similar tasks emerge in different contexts. It is a good time to think about how they can be generalized and to reconsider your model. This way, you will come up with utility functions best put into *utils.py*.

- Start writing code only after careful thinking

It is no point in writing code before you completely understand your problem domain you want to model. It is more efficient to build abstractions in your head or on paper, then to split it into modular chunks, and only after that start coding.

- Write the documentation *before* the code

If you document the function parameters and the return values in advance, as well as construct a doctest, you are enforced to think about the problem deeply and to create a good interface. Moreover, it guarantees that the documentation is not missing (what you would anyway have to write at some point, so why not at the beginning?).

- Give doctest-compatible examples

You hit two birds with the same stone: provide an example for the user and get some confidence that your code works as intended (at least for the particular example). As mentioned in the previous point, write them *before* the actual code implementation. Doctests do not replace careful testing.

- Keep the documentation as part of your code

The problem with wiki pages is that they are version controlled in a different Git repository. It makes it longer to change a hosting service (e.g. moving from GitHub to GitLab), you need to maintain two repositories, cannot change the documentation and the code in the same commit, and you have to rely on the rendering capabilities of the hosting service (e.g. GitHub cannot render math). It is therefore better to keep the documentation as part of your project in a dedicated directory (`docs/` in our case), use a documentation generator (*Sphinx* in our case) and host it online (on *Read the Docs* in our case).

# CODING CONVENTIONS

# **DOCUMENTATION**

Writing documentation is necessary when you contribute to this project, either by writing code or by changing/extending the external documents. In this section, we give tips how you can write them. This is an opinionated topic and it lays down how it is currently done. Feel free to suggest new ideas.

Whenever you write the documentation, first *test it locally* before pushing changes: Read the Docs spends about 1000 seconds to build the documentation.

It took me a long time to experiment with the Sphinx settings that provide the output what you can see in the rendered documentation. Some notes concerning these efforts can be found in the Notes section. Another way to learn about Sphinx documentation is by reading the source of the existing documentation. On each page of the HTML documentation, the header contains a *View page source* hyperlink.

## **11.1 Code documentation**

As shown in the *figure*, codes are written either as Python files or as Jupyter notebooks. With the proper extensions (see the `docs/source/conf.py` file), both of them are automatically included in the documentation by Sphinx. In what follows, we concentrate on documenting Python files.

The docstings are written in RST, following the numpydoc style, using the Napoleon Sphinx extension.

Provide doctests to demonstrate the use of the function you write and to provide minimal testing.

The line length is 100 characters, as defined in the `/.editorconfig` file.

## **11.2 External documentation**

The external documentation is written in reStructuredText (RST) and in Markdown. Sphinx, by default, uses RST, extending it with more capabilities. However, recommonmark can parse Markdown files and automatically convert them to RST at documentation build time. This is a great help as more people are used to the simple Markdown than to the more complex (and more capable) RST. Actually, the current document you read was also written in Markdown. Of course, it is perfectly fine if you write the documentation exclusively in RST.

Currently, the structure of the documentation is written in RST, and most of the external documentation in Markdown.

# DEVELOPMENT

Intro ...

## 12.1 General workflow

The development workflow can be followed in the following figure.

1. You write the Python source codes (.py) and the Jupyter notebooks (.ipynb) on your local machine. If you want, you can create the documentation locally using Sphinx.

2. When a logical unit has been finished, you commit the changes with `git push`. This will upload the new version of the modified files to a remote repository (currently GitHub). If you edit files in the remote repository and you want those changes to be present in your local copy, you can use `git pull`. For more details on Git, read the manual.

3. Several commit hooks are attached to the remote repository. When they detect a change, certain actions are activated. One the one hand, the online version of the documentation, hosted on Read the Docs will be updated. On the other hand, static analyzers will reanalyze the new version of the code. Some of them may create a pull request based on their recommendations.

## 12.2 Profiling

I use Pyinstrument for profiling the code. It comes with the *CristalX* installation.

Otherwise, you can install it with `pip install pyinstrument` or by *conda* after the *conda-forge* channel has been activated.

```
conda install -c conda-forge pyinstrument
```

As Pyinstrument has no dependencies, you can safely install it to your current environment. If you do not want to take a risk, create a new environment. With *conda*, you can do e.g.

```
conda create -n pyinstrument python=3.7 scipy matplotlib
conda activate pyinstrument
conda install -c conda-forge pyinstrument
```

The *profiling* module provides a wrapper around Pyinstrument. Put the code you want to profile in the `profile` context manager, e.g.

```
>>> import random
>>> from grains.profiling import profile
>>> with profile('html') as p:
...     for _ in range(1000000):
...         rand_num = random.uniform(1, 2.2)
```

For more details see the *API reference*.

# CONTRIBUTING

*CristalX* is an open-source project that welcomes contributions of any kind.

## 13.1 What can you help in?

The major fields in which you can help are the following (in increasing complexity).

### 13.1.1 Use cases

If you use *CristalX* in your project, let us know. Parts of your code, if they are general enough to be incorporated, could be included in *CristalX*. Make contact with us by *opening an issue*.

### 13.1.2 Documentation

If the documentation of some functions

  • is missing

  • is incomprehensible

  • does not contain examples

  • is not rendered properly

or if the code examples break, *open a pull request*. Similarly, open a pull request if the guides

  • contain typos

  • are incomprehensible

### 13.1.3 Code

  • You can report bugs by *opening an issue*.

  • If you want to help but do not know where to start, consider the currently open issues, especially the ones with the help wanted label.

  • You can implement new features. First of all, *contact us* if you plan to work on a non-trivial feature. This will save work for you. Use the *fork & PR* workflow.

## 13.2 How to contribute

First, create a GitHub account. There are two ways to contribute.

### 13.2.1 Open an issue

To open a new issue, click on the green button on the *Issues* page.

### 13.2.2 Fork & pull request

At the moment, there are no code formatting guidelines, the best is to follow the formatting of the existing code.

1. Fork the GitHub repository by clicking on the *Fork* button at the top right corner.
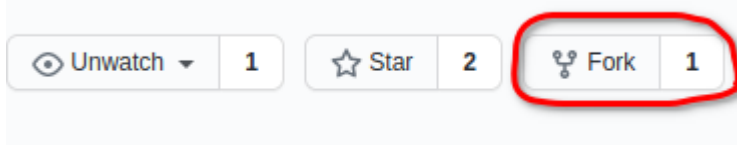
2. Install *CristalX locally* by cloning your fork (`git clone`).

3. Create a new branch.

   Do not work on the default (*master*) branch but create a new feature branch, e.g.

   ```
   git checkout -b new_feature
   ```

4. Make your changes.

   Add or modify files and regularly commit your changes to your local clone with meaningful commit messages (`git commit`). Do not forget to test your code: if you do not provide unit tests, at least write doctests.

5. Push your changes to your remote fork on GitHub (`git push`).

6. Visit your forked repository on GitHub and click on the *Pull request* button. See the GitHub documentation for details.
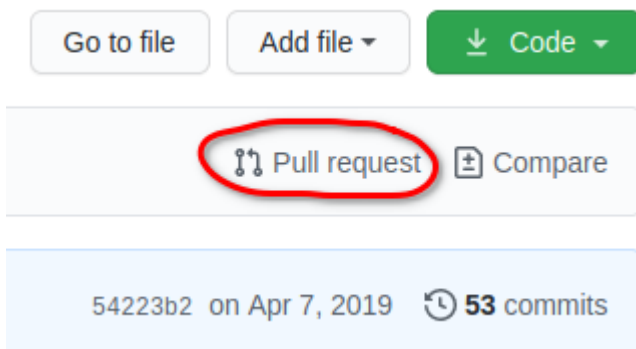
7. Keep your fork up to date

   While you work on your forked repository, changes may be committed to the original (called *upstream*) repository. To make sure you keep your fork up to date with the upstream repository, follow the instructions in the GitHub documentation.

This was a basic overview, for more details check out the following documents:

- https://gist.github.com/Chaser324/ce0505fbed06b947d962

- https://github.com/susam/gitpr

# VERSIONING

The versioning of *CristalX* starts with 1.0.0 and it somewhat follows the rules of Semantic Versioning 2.0.0, with the syntax MAJOR.MINOR.PATCH, where MAJOR introduces significant changes, MINOR comes with smaller changes, and PATCH provides a fixture or a tiny improvement either in the code or in the documentation.

## 14.1 Why not SemVer?

Semantic Versioning (SemVer) is a widely used versioning scheme, applicable for **public APIs**. Its purpose is to be rigorous on how to indicate when a bug fix, new features or incompatible changes in the public API are introduced. There is an excellent discussion about it here and a detailed guide here. Many criticize it for not being indicative about the rate of important changes. E.g. 1.8.5 –> 1.9.0 may include dozens of relevant improvements, while 1.9.5 –> 2.0.0 may merely be a simple clean-up that changes the public API. However, SemVer was never meant to be used for software version numbers.

*CristalX is not a library*, but a collection of tools that operate at a high level. Most of its functions are exposed to the user, except the ones marked with a leading underscore or two leading underscores. However, the public API would be a subset of these functions: e.g. the functions of the *utils* module are exposed but they are mostly intended to be used by other modules, not by the user. As *CristalX* is not a library but rather a tool for rapid prototyping, semantic versioning would not make much sense.

## 14.2 Why not CalVer?

Calendar Versioning (CalVer) is an alternative to SemVer. *CristalX* may not come with regular changes in the future or it will get updates at irregular intervals. We do not want to give the impression that relevant changes are introduced linearly in time. Moreover, the changelog and the time stamp in the git commits clearly show when new releases are published.

## 14.3 Our versioning

As there is no unconditionally best versioning system, we came up with our own, which seems to fit well for research code like *CristalX*. The starting point is SemVer with some differences. Our versioning is intended for humans. The MAJOR version in increased only for substantially new features. This is, of course, subjective but we want to avoid large MAJOR version numbers as e.g. in Firefox. The guideline to follow is that the novelty of a feature makes the MAJOR version increase, not the number of additions. Here are some examples. Assume that we are at 1.2.3. We fixed a set of related bugs in the code: 1.2.3 –> 1.2.4. Then we implemented several functionalities to speed up the code: 1.2.4 –> 1.3.0. A new module was created and several others were modified that allowed us to carry out groundbreaking research: 1.3.0 –> 2.0.0. An existing algorithm was improved to handle the corner cases: 2.0.0 –> 2.1.0.

Similarly to SemVer, when one of the digits increases, the ones right to it are set to zero (e.g. 1.0.1 –> 1.0.2, 1.0.3 –> 1.1.0, 1.8.6 –> 2.0.0).

We try to keep backward compatibility. Insignificant changes are postponed, and are included as part of a MINOR release. If you often feel the need to introduce changes to the function signatures, rather **add** new functions and give **deprecation notices** than remove or modify existing ones. This does not lead to code bloat in the long run because deprecated syntax is removed from time to time. To mark a function, a method or a class as deprecated, import the `deprecated` function from the *deprecation* package and follow its syntax.

The *master* branch of the Git repository contains the latest developments since the last published release. These unreleased modifications are allowed to contain incompatibilities (changes in the function signatures, etc.) compared to the latest release. This flexibility is essential for rapid prototyping in research codes like *CristalX*. However, these possible incompatibilities must be fixed for the next release, see the previous paragraph.

### 14.3.1 Connection with the Git workflow

The version numbers are reflected in the tag names. Your normal Git workflow stays the same: commit modifications and push them to the remote repository. When you want to mark a commit yet to be included in a certain version, type

```
git tag -m "Concise message" v<version_number>
```

where `<version_number>` has the form MAJOR.MINOR.PATCH. Note that it is preceded by the `v` letter, conventionally used for tags. As an example:

```
git tag -m "Initial release." v1.0.0
```

Keep the tag message short: the detailed changes since the previous version are collected in the changelog.

The tag is pushed by

```
git push --tags
```

Whenever you publish a tag, and hence update the changelog, also create a release for that tag on GitHub. Copy the changes the new version brings from the changelog to the description of the release. In this regard, we follow the way of JabRef. The zipped size of *CristalX* is quite small, so size constraint will not be a problem on GitHub.

# **SEGMENTATION**

This module contains the Segmentation class, responsible for the image segmentation of grain-based materials (rocks, metals, etc.)

## 15.1 Classes

| | |
|---|---|
| *Segmentation* | Segmentation of grain-based microstructures |

### 15.1.1 grains.segmentation.Segmentation

**class** grains.segmentation.**Segmentation**(*image_location*, *save_location=None*, *interactive_mode=True*)

Segmentation of grain-based microstructures

**original_image**

Matrix representing the initial, unprocessed image.

> **Type** ndarray

**save_location**

Directory where the processed images are saved

> **Type** str

**__init__**(*image_location*, *save_location=None*, *interactive_mode=True*)

Initialize the class with file paths and with some options

> **Parameters**
>
> - **image_location** (*str*) – Path to the image to be segmented, file extension included.
> - **save_location** (*str, optional*) – Path to directory where images will be outputted. If not given, the same directory is used where the input image is loaded from.
> - **interactive_mode** (*bool, optional*) – When True, images of each image manipulation step are plotted and details are shown in the console. Default is False.
>
> **Returns** *None.*

**Methods**

| | |
|---|---|
| *__init__*(image_location[, save_location, . . . ]) | Initialize the class with file paths and with some options |
| *create_skeleton*(boundary_image) | Use thinning on the grain boundary image to obtain a single-pixel wide skeleton. |
| *filter_image*(window_size[, image_matrix]) | Median filtering on an image. |
| *find_grain_boundaries*(segmented_image) | Find the grain boundaries. |
| *initial_segmentation*(*args) | Perform the quick shift superpixel segmentation on an image. |
| *merge_clusters*(segmented_image[, threshold]) | Merge tiny superpixel clusters. |
| *save_array*(filename, array) | Save an image as a numpy array. |
| *save_image*(filename, array[, is_label_image]) | Save an image as a numpy array. |
| *watershed_segmentation*(skeleton) | Watershed segmentation of a granular microstructure. |

**class** grains.segmentation.**Segmentation**(*image_location*, *save_location=None*, *interactive_mode=True*)

   Bases: [object](#)

   Segmentation of grain-based microstructures

   **original_image**
      Matrix representing the initial, unprocessed image.

         **Type** ndarray

   **save_location**
      Directory where the processed images are saved

         **Type** [str](#)

   **create_skeleton**(*boundary_image*)
      Use thinning on the grain boundary image to obtain a single-pixel wide skeleton.

         **Parameters boundary_image** (*bool ndarray*) – A binary image containing the objects to be skeletonized.

         **Returns skeleton** (*bool ndarray*) – Thinned image.

   **filter_image**(*window_size*, *image_matrix=None*)
      Median filtering on an image. The median filter is useful in our case as it preserves the important borders (i.e. the grain boundaries).

         **Parameters**

            • **window_size** (*int*) – Size of the sampling window.

            • **image_matrix** (*3D ndarray with size 3 in the third dimension, optional*) – Input image to be filtered. If not given, the original image is used.

         **Returns filtered_image** (*3D ndarray with size 3 in the third dimension*) – Filtered image, output of the median filter algorithm.

   **find_grain_boundaries**(*segmented_image*)
      Find the grain boundaries.

         **Parameters segmented_image** (*ndarray*) – Label image, output of a segmentation.

         **Returns boundary** (*bool ndarray*) – A bool ndarray, where True represents a boundary pixel.

**initial_segmentation**(*\*args*)

Perform the quick shift superpixel segmentation on an image. The quick shift algorithm is invoked with its default parameters.

> **Parameters** **\*args** (*3D numpy array with size 3 in the third dimension*) – Input image to be segmented. If not given, the original image is used.

> **Returns** **segment_mask** (*ndarray*) – Label image, output of the quick shift algorithm.

**merge_clusters**(*segmented_image*, *threshold=5*)

Merge tiny superpixel clusters. Superpixel segmentations result in oversegmented images. Based on graph theoretic tools, similar clusters are merged.

> **Parameters**

> - **segmented_image** (*ndarray*) – Label image, output of a segmentation.

> - **threshold** (*float, optional*) – Regions connected by edges with smaller weights are combined.

> **Returns** **merged_superpixels** (*ndarray*) – The new labelled array.

**save_array**(*filename*, *array*)

Save an image as a numpy array. The array is saved in the standard numpy format, into the directory determined by the *save_location* attribute.

> **Parameters**

> - **filename** (*str*) – The array is saved under this name, with extension .npy

> - **array** (*ndarray*) – An image represented as a numpy array.

**save_image**(*filename*, *array*, *is_label_image=False*)

Save an image as a numpy array. The array is saved in the standard numpy format, into the directory determined by the *save_location* attribute.

> **Parameters**

> - **filename** (*str*) – The array is saved under this name, with extension .npy

> - **array** (*ndarray*) – An image represented as a numpy array.

> - **is_label_image** (*bool*) – True if the array represents a labeled image.

**watershed_segmentation**(*skeleton*)

Watershed segmentation of a granular microstructure. Uses the watershed transform to label non-overlapping grains in a cellular microstructure given by the grain boundaries.

> **Parameters** **skeleton** (*bool ndarray*) – A binary image, the skeletonized grain boundaries.

> **Returns** **segmented** (*ndarray*) – Label image, output of the watershed segmentation.

CHAPTER

# SIXTEEN

# GALA

A trimmed version of the Gala project (https://github.com/janelia-flyem/gala) with some additions (new function, added documentation for the existing ones). Gala is licensed by the Janelia Farm License: http://janelia-flyem.github. io/janelia_farm_license.html

Copyright 2012 HHMI. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of HHMI nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 16.1 Functions

| | |
|---|---|
| *imextendedmin*(image, h[, connectivity]) | Extended-minima transform. |
| *hminima*(a, thresh) | Suppress all minima that are shallower than thresh. |
| *imhmin*(a, thresh) | Suppress all minima that are shallower than thresh. |
| *morphological_reconstruction*(marker, mask[, . . . ]) | Perform morphological reconstruction of the marker into the mask. |
| *regional_minima*(a[, connectivity]) | Find the regional minima in an ndarray. |
| *complement*(a) | |

## 16.1.1 grains.gala_light.imextendedmin

grains.gala_light.**imextendedmin**(*image*, *h*, *connectivity=1*)
> Extended-minima transform. The extended minima transform is the regional minima of the h-minima transform. The implementation follows the MATLAB function under the same name.

> **Parameters**

>> - **image** (*ndarray*) – The input array on which to perform imextendedmin.

>> - **h** (*float*) – Any local minima shallower than this will be flattened.

>> - **connectivity** (*int, optional*) – Determines which elements are considered as neighbors of the central element. Elements up to a squared distance of *connectivity* from the center are considered neighbors. If connectivity=1, no diagonal elements are neighbors.

> **Returns** *bool ndarray* – True at places of the extended minima.

## 16.1.2 grains.gala_light.hminima

grains.gala_light.**hminima**(*a*, *thresh*)
> Suppress all minima that are shallower than thresh.

> **Parameters**

>> - **a** (*array*) – The input array on which to perform hminima.

>> - **thresh** (*float*) – Any local minima shallower than this will be flattened.

> **Returns** **out** (*array*) – A copy of the input array with shallow minima suppressed.

## 16.1.3 grains.gala_light.imhmin

grains.gala_light.**imhmin**(*a*, *thresh*)
> Suppress all minima that are shallower than thresh.

> **Parameters**

>> - **a** (*array*) – The input array on which to perform hminima.

>> - **thresh** (*float*) – Any local minima shallower than this will be flattened.

> **Returns** **out** (*array*) – A copy of the input array with shallow minima suppressed.

## 16.1.4 grains.gala_light.morphological_reconstruction

grains.gala_light.**morphological_reconstruction**(*marker*, *mask*, *connectivity=1*)
> Perform morphological reconstruction of the marker into the mask.

> See the Matlab image processing toolbox documentation for details: http://www.mathworks.com/help/toolbox/images/f18-16264.html

### 16.1.5 grains.gala_light.regional_minima

grains.gala_light.**regional_minima**(*a*, *connectivity=1*)

   Find the regional minima in an ndarray. As written in the MATLAB documentation of the imregionalmin function: "Regional minima are connected components of pixels with a constant intensity value, surrounded by pixels with a higher value."

### 16.1.6 grains.gala_light.complement

grains.gala_light.**complement**(*a*)

grains.gala_light.**complement**(*a*)

grains.gala_light.**hminima**(*a*, *thresh*)

   Suppress all minima that are shallower than thresh.

> **Parameters**
>
> - **a** (*array*) – The input array on which to perform hminima.
>
> - **thresh** (*float*) – Any local minima shallower than this will be flattened.
>
> **Returns** **out** (*array*) – A copy of the input array with shallow minima suppressed.

grains.gala_light.**imextendedmin**(*image*, *h*, *connectivity=1*)

   Extended-minima transform. The extended minima transform is the regional minima of the h-minima transform. The implementation follows the MATLAB function under the same name.

> **Parameters**
>
> - **image** (*ndarray*) – The input array on which to perform imextendedmin.
>
> - **h** (*float*) – Any local minima shallower than this will be flattened.
>
> - **connectivity** (*int, optional*) – Determines which elements are considered as neighbors of the central element. Elements up to a squared distance of *connectivity* from the center are considered neighbors. If connectivity=1, no diagonal elements are neighbors.
>
> **Returns** *bool ndarray* – True at places of the extended minima.

grains.gala_light.**imhmin**(*a*, *thresh*)

   Suppress all minima that are shallower than thresh.

> **Parameters**
>
> - **a** (*array*) – The input array on which to perform hminima.
>
> - **thresh** (*float*) – Any local minima shallower than this will be flattened.
>
> **Returns** **out** (*array*) – A copy of the input array with shallow minima suppressed.

grains.gala_light.**morphological_reconstruction**(*marker*, *mask*, *connectivity=1*)

   Perform morphological reconstruction of the marker into the mask.

   See the Matlab image processing toolbox documentation for details: http://www.mathworks.com/help/toolbox/images/f18-16264.html

grains.gala_light.**regional_minima**(*a*, *connectivity=1*)

   Find the regional minima in an ndarray. As written in the MATLAB documentation of the imregionalmin function: "Regional minima are connected components of pixels with a constant intensity value, surrounded by pixels with a higher value."

# SEVENTEEN

# ANALYSIS

This module contains the Analysis class, responsible for the analysis of segmented grain-based microstructures.

All the examples assume that the modules *numpy* and *matplotlib.pyplot* were imported as *np* and *plt*, respectively.

## 17.1 Classes

| | |
|---|---|
| *Analysis* | Analysis of grain assemblies. |

### 17.1.1 grains.analysis.Analysis

**class** grains.analysis.**Analysis**(*label_image*, *interactive_mode=False*)

    Analysis of grain assemblies.

    **original_image**

        Matrix representing the initial, unprocessed image.

        **Type** ndarray

    **save_location**

        Directory where the processed images are saved

        **Type** str

    **__init__**(*label_image*, *interactive_mode=False*)

        Initialize the class with file paths and with some options

        **Parameters**

            • **label_image** (*ndarray*) – Segmented image.

            • **interactive_mode** (*bool, optional*) – When True, images of each image manipulation step are plotted and details are shown in the console. Default is False.

        **Returns** *None.*

**Methods**

| | |
|---|---|
| *__init__*(label_image[, interactive_mode]) | Initialize the class with file paths and with some options |
| *compute_properties*() | Determines relevant properties of the grains. |
| *set_scale*([pixel_per_unit]) | Defines a scale for performing computations in that unit. |
| *show_grains*([grain_property]) | Display the grains, optionally with a property superposed. |
| *show_properties*([gui]) | Displays previously computed properties of the grains |

# 17.2 Functions

| | |
|---|---|
| *feret_diameter*(prop) | Determines the maximum Feret diameter. |
| *plot_prop*(prop[, pixel_per_unit, show_axis]) | Plots relevant region properties into a single figure. |
| *plot_grain_characteristic*(characteristic, …) | Plots the distribution of a given grain characteristic. |
| *show_label_image*(label_image[, alpha]) | Displays a labeled image. |
| *label_image_skeleton*(label_image) | Skeleton of a labeled image. |
| *thicken_skeleton*(skeleton, thickness) | Thickens a skeleton by morphological dilation. |
| *label_image_apply_mask*(label_image, mask, value) | Changes parts of a labeled image to a given value. |

## 17.2.1 grains.analysis.feret_diameter

grains.analysis.**feret_diameter**(*prop*)
    Determines the maximum Feret diameter.

> **Parameters prop** (*RegionProperties*) – Describes a labeled region.

> **Returns max_feret_diameter** (*float*) – Maximum Feret diameter of the region.

**See also:**

**skimage.measure.regionprops()** Measure properties of labeled image regions

**Examples**

```
>>> import numpy as np
>>> from skimage.measure import regionprops
>>> image = np.ones((2,2), dtype=np.int8)
>>> prop = regionprops(image)[0]
>>> feret_diameter(prop)
2.23606797749979
```

## 17.2.2 grains.analysis.plot_prop

grains.analysis.**plot_prop**(*prop*, *pixel_per_unit=1*, *show_axis=True*)
    Plots relevant region properties into a single figure. Four subfigures are created, giving the region's

- image, its area and its center

- filled image, its area

- bounding box, its area

- convex image, its area

        **Parameters**

- **prop** (*RegionProperties*) – Describes a labeled region.

- **pixel_per_unit** (*float or int, optional*) – Number of pixels contained in a certain unit. The default is 1, in which case all measurements are performed in pixel units.

      **Returns** **fig** (*matplotlib.figure.Figure*) – The figure object is returned in case further manipulations are necessary.

## 17.2.3 grains.analysis.plot_grain_characteristic

grains.analysis.**plot_grain_characteristic**(*characteristic*, *centers*, *interpolation='linear'*, *grid_size=(100, 100)*, *\*\*kwargs*)
    Plots the distribution of a given grain characteristic.

    One way to gain insight into a grain assembly is to plot the distribution of a certain grain property in the domain the grains occupy. In this function, for each grain, and arbitrary (scalar) quantity is associated to the center of the grain. In case of *n* grains, *n* data points span the interpolant and the given characteristic is interpolated on a grid of the AABB of the grain centers.

        **Parameters**

- **characteristic** (*ndarray*) – Characteristic property, the distribution of which is sought. A 1D numpy array.

- **centers** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a grain, and the two columns giving the Cartesian coordinates of the grain center.

- **interpolation** (*{'nearest', 'linear', 'cubic'}, optional*) – Type of the interpolation for creating the distribution. The default is *'linear'*.

- **grid_size** (*tuple of int, optional*) – 2-tuple, the size of the grid on which the data is interpolated. The default is (100, 100).

        **Other Parameters**

- **center_marker** (*str, optional*) – Marker indicating the center of the grains. The default is *'P'*. For a list of supported markers, see the documentation. If you do not want the centers to be shown, choose *'none'*.

- **show_axis** (*bool, optional*) – If True, the axes are displayed. The default is False.

      **Returns** *None*

    See also:

    scipy.interpolate.griddata()

### Notes

This function knows nothing about how the center of a grain is determined and what characteristic features a grain has. It only performs interpolation and visualization, hence decoupling the plotting from the actual representation of grains and their characteristics. For instance, a grain can be represented as a spline surface, as a polygon, as an assembly of primitives (often triangles), as pixels, just to mention some typical scenarios. Calculating the center of a grain depends on the grain representation at hand. Similarly, one can imagine various grain characteristics, such as area, diameter, Young modulus.

### Examples

Assume that the grain centers are sampled from a uniformly random distribution on the unit square.

```
>>> n_data = 100
>>> points = np.random.random((n_data, 2))
```

The quantity we want to plot has a parabolic distribution with respect to the position of the grain centers.

```
>>> func = lambda x, y: 1 - (x-0.5)**2 - (y-0.5)**2
>>> plot_grain_characteristic(func(points[:, 0], points[:, 1]), points, center_
↪marker='*')
>>> plt.show()
```

## 17.2.4 grains.analysis.show_label_image

grains.analysis.**show_label_image**(*label_image*, *alpha=1*)
  Displays a labeled image.

  A random color is associated with each labeled region. If boundary pixels are present in the image, they are plotted in black.

> **Parameters**
>> • **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of non-negative integers. The label 0 is assumed to denote a boundary pixel.
>>
>> • **alpha** (*float, optional*) – Opacity of colorized labels. Must be within [0, 1].
>
> **Returns** *None*

## 17.2.5 grains.analysis.label_image_skeleton

grains.analysis.**label_image_skeleton**(*label_image*)
  Skeleton of a labeled image.

  The skeleton of a labeled image is a single-pixel wide network that separates the labeled regions.

> **Parameters label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> **Returns** *ndarray* – A 2D bool numpy array having the same size as label_image, where True represents the skeleton pixels.

  **See also:**

  *thicken_skeleton()*

## 17.2.6 grains.analysis.thicken_skeleton

grains.analysis.**thicken_skeleton**(*skeleton*, *thickness*)
Thickens a skeleton by morphological dilation.

>   **Parameters**
>
>   - **skeleton** (*ndarray*) – Skeleton of a binary image, represented as a bool 2D numpy array.
>
>   - **thickness** (*int*) – Thickness of the resulting boundaries.
>
>   **Returns** *ndarray* – A 2D bool numpy array, where True represents the thickened skeleton.
>
>   **See also:**
>
>   *label_image_skeleton()*

## 17.2.7 grains.analysis.label_image_apply_mask

grains.analysis.**label_image_apply_mask**(*label_image*, *mask*, *value*)
Changes parts of a labeled image to a given value.

>   Convenience function, equivalent to `label_image[mask] = value` but the original array `label_image` is *not* overwritten.
>
>   **Parameters**
>
>   - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
>   - **mask** (*ndarray*) – Boolean array of the same size as `label_image`, marking the pixels that will be replaced by `value`.
>
>   - **value** (*int*) – The masked pixels are replaced by this value.
>
>   **Returns** *ndarray* – Copy of the input image, its selected pixels being replaced by the given value.

**class** grains.analysis.**Analysis**(*label_image*, *interactive_mode=False*)
Bases: `object`

>   Analysis of grain assemblies.
>
>   **original_image**
>   Matrix representing the initial, unprocessed image.
>
>   >   **Type** ndarray
>
>   **save_location**
>   Directory where the processed images are saved
>
>   >   **Type** str
>
>   **compute_properties**()
>   Determines relevant properties of the grains. The area of each grain is determined in the units previously given in the *set_scale* method.
>
>   >   **Parameters**
>   >
>   >   - **window_size** (*int*) – Size of the sampling window.
>   >
>   >   - **image_matrix** (*3D ndarray with size 3 in the third dimension, optional*) – Input image to be filtered. If not given, the original image is used.

> **Returns  filtered_image** (*3D ndarray with size 3 in the third dimension*) – Filtered image, output
> of the median filter algorithm.

**set_scale**(*pixel_per_unit=1*)

> Defines a scale for performing computations in that unit. Image measures (area, diameter, etc.) are per-
> formed on a matrix corresponding to a label image. Therefore, the result of all the computations are
> obtained in pixel units. It is often of interest to access the results in physical units (mm, cm, inch, etc.).
> Manually converting pixels, pixel squares, etc. to pyhsical units, physical unit sqaures, etc. are tedious
> and error prone. Once the conversion between a pixel and a physical unit is given, all the subsequent
> calculations are performed in the desired physical unit.

> > **Parameters  pixel_per_unit** (*float or int or scalar ndarray, optional*) – Number of pixels con-
> > tained in a certain unit. The default is 1, in which case all measurements are performed in
> > pixel units.

> > **Returns**  *None.*

**show_grains**(*grain_property=None*)

> Display the grains, optionally with a property superposed.

> > **Parameters**

> > > **grain_property** (*{None, 'area', 'centroid', 'coordinate',*) – '**equivalent_diameter', 'feret_diameter', 'label'}**
> > > optional

> > > If not None, the selected property is shown on the grain as text.

> > **Returns**  *None.*

**show_properties**(*gui=False*)

> Displays previously computed properties of the grains

> > **Parameters  gui** (*bool, optional*) – If true, the grain properties are shown in a GUI. If false, they
> > are printed to stdout. The default is False. The GUI requires the *dfgui* modul, which can be
> > obtained from https://github.com/bluenote10/PandasDataFrameGUI

> > **Returns**  *None.*

grains.analysis.**feret_diameter**(*prop*)

> Determines the maximum Feret diameter.

> > **Parameters  prop** (*RegionProperties*) – Describes a labeled region.

> > **Returns  max_feret_diameter** (*float*) – Maximum Feret diameter of the region.

> **See also:**

> **skimage.measure.regionprops()**  Measure properties of labeled image regions

> **Examples**

> ```
> >>> import numpy as np
> >>> from skimage.measure import regionprops
> >>> image = np.ones((2,2), dtype=np.int8)
> >>> prop = regionprops(image)[0]
> >>> feret_diameter(prop)
> 2.23606797749979
> ```

grains.analysis.**label_image_apply_mask**(*label_image*, *mask*, *value*)

> Changes parts of a labeled image to a given value.

Convenience function, equivalent to `label_image[mask] = value` but the original array `label_image` is *not* overwritten.

> **Parameters**
>
> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> - **mask** (*ndarray*) – Boolean array of the same size as `label_image`, marking the pixels that will be replaced by `value`.
>
> - **value** (*int*) – The masked pixels are replaced by this value.
>
> **Returns** *ndarray* – Copy of the input image, its selected pixels being replaced by the given value.

`grains.analysis.`**`label_image_skeleton`**(*label_image*)

> Skeleton of a labeled image.
>
> The skeleton of a labeled image is a single-pixel wide network that separates the labeled regions.
>
> > **Parameters label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
> >
> > **Returns** *ndarray* – A 2D bool numpy array having the same size as `label_image`, where True represents the skeleton pixels.
>
> **See also:**
>
> *thicken_skeleton()*

`grains.analysis.`**`plot_grain_characteristic`**(*characteristic*, *centers*, *interpolation='linear'*, *grid_size=(100, 100), \*\*kwargs*)

> Plots the distribution of a given grain characteristic.
>
> One way to gain insight into a grain assembly is to plot the distribution of a certain grain property in the domain the grains occupy. In this function, for each grain, and arbitrary (scalar) quantity is associated to the center of the grain. In case of *n* grains, *n* data points span the interpolant and the given characteristic is interpolated on a grid of the AABB of the grain centers.
>
> > **Parameters**
> >
> > - **characteristic** (*ndarray*) – Characteristic property, the distribution of which is sought. A 1D numpy array.
> >
> > - **centers** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a grain, and the two columns giving the Cartesian coordinates of the grain center.
> >
> > - **interpolation** (*{'nearest', 'linear', 'cubic'}, optional*) – Type of the interpolation for creating the distribution. The default is *'linear'*.
> >
> > - **grid_size** (*tuple of int, optional*) – 2-tuple, the size of the grid on which the data is interpolated. The default is (100, 100).
> >
> > **Other Parameters**
> >
> > - **center_marker** (*str, optional*) – Marker indicating the center of the grains. The default is *'P'*. For a list of supported markers, see the [documentation](#). If you do not want the centers to be shown, choose *'none'*.
> >
> > - **show_axis** (*bool, optional*) – If True, the axes are displayed. The default is False.
> >
> > **Returns** *None*
>
> **See also:**
>
> `scipy.interpolate.griddata()`

### Notes

This function knows nothing about how the center of a grain is determined and what characteristic features a grain has. It only performs interpolation and visualization, hence decoupling the plotting from the actual representation of grains and their characteristics. For instance, a grain can be represented as a spline surface, as a polygon, as an assembly of primitives (often triangles), as pixels, just to mention some typical scenarios. Calculating the center of a grain depends on the grain representation at hand. Similarly, one can imagine various grain characteristics, such as area, diameter, Young modulus.

### Examples

Assume that the grain centers are sampled from a uniformly random distribution on the unit square.

```
>>> n_data = 100
>>> points = np.random.random((n_data, 2))
```

The quantity we want to plot has a parabolic distribution with respect to the position of the grain centers.

```
>>> func = lambda x, y: 1 - (x-0.5)**2 - (y-0.5)**2
>>> plot_grain_characteristic(func(points[:, 0], points[:, 1]), points, center_
→marker='*')
>>> plt.show()
```

grains.analysis.**plot_prop**(*prop*, *pixel_per_unit=1*, *show_axis=True*)
    Plots relevant region properties into a single figure. Four subfigures are created, giving the region's

- image, its area and its center

- filled image, its area

- bounding box, its area

- convex image, its area

    #### Parameters

- **prop** (*RegionProperties*) – Describes a labeled region.

- **pixel_per_unit** (*float or int, optional*) – Number of pixels contained in a certain unit. The default is 1, in which case all measurements are performed in pixel units.

    **Returns** **fig** (*matplotlib.figure.Figure*) – The figure object is returned in case further manipulations are necessary.

grains.analysis.**show_label_image**(*label_image*, *alpha=1*)
    Displays a labeled image.

    A random color is associated with each labeled region. If boundary pixels are present in the image, they are plotted in black.

    #### Parameters

- **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of non-negative integers. The label 0 is assumed to denote a boundary pixel.

- **alpha** (*float, optional*) – Opacity of colorized labels. Must be within [0, 1].

    **Returns** *None*

grains.analysis.**thicken_skeleton**(*skeleton*, *thickness*)
    Thickens a skeleton by morphological dilation.

Parameters

- **skeleton** (*ndarray*) – Skeleton of a binary image, represented as a bool 2D numpy array.

- **thickness** (*int*) – Thickness of the resulting boundaries.

**Returns** *ndarray* – A 2D bool numpy array, where True represents the thickened skeleton.

**See also:**

*label_image_skeleton()*

# MESHING

## 18.1 Classes

### 18.1.1 grains.meshing.SkeletonGeometry

**class** grains.meshing.**SkeletonGeometry**(*leftright_periodicity*, *topbottom_periodicity*)

> **abstract __init__**(*leftright_periodicity*, *topbottom_periodicity*)
>     Initialize self. See help(type(self)) for accurate signature.

> **Methods**

| | |
|---|---|
| __init__(leftright_periodicity, ...) | Initialize self. |

### 18.1.2 grains.meshing.QuadSkeletonGeometry

**class** grains.meshing.**QuadSkeletonGeometry**(*leftright_periodicity=False*, *topbottom_periodicity=False*)

> **__init__**(*leftright_periodicity=False*, *topbottom_periodicity=False*)
>     Initialize self. See help(type(self)) for accurate signature.

**Methods**

| | |
|---|---|
| _\_\_init\_\__([leftright_periodicity, ... ]) | Initialize self. |

### 18.1.3 grains.meshing.TriSkeletonGeometry

**class** grains.meshing.**TriSkeletonGeometry**(*leftright_periodicity=False,* *topbottom_periodicity=False,* *arrangement='conservative'*)

> **\_\_init\_\_** (*leftright_periodicity=False*, *topbottom_periodicity=False*, *arrangement='conservative'*)
> Initialize self. See help(type(self)) for accurate signature.

**Methods**

| | |
|---|---|
| _\_\_init\_\__([leftright_periodicity, ... ]) | Initialize self. |

### 18.1.4 grains.meshing.FixedDict

**class** grains.meshing.**FixedDict**(*dictionary*)
> https://stackoverflow.com/a/14816446/4892892

> **\_\_init\_\_** (*dictionary*)
> Initialize self. See help(type(self)) for accurate signature.

**Methods**

| | |
|---|---|
| _\_\_init\_\__(dictionary) | Initialize self. |

### 18.1.5 grains.meshing.OOF2

**class** grains.meshing.**OOF2**

> **\_\_init\_\_** ()
> Initialize self. See help(type(self)) for accurate signature.

**Methods**

| | | |
|---|---|---|
| _\_\_init\_\__ | Initialize self. | |
| _create_material_(name) | | |
| | **Parameters name** (*TYPE*) – DESCRIPTION. | |
| _create_microstructure_([name]) | Creates a microstructure from an image. | |

Continued on next page

| Table 6 – continued from previous page | |
|---|---|
| *create_skeleton*(nelem_x, nelem_y, geometry) | |
| *load_pixelgroups*(microstructure_file) | **Parameters microstructure_file** (*str*) – DESCRIPTION. |
| *materials2groups*(materials[, groups]) | **Parameters**<br>• **materials** (*list of str*) – DESCRIPTION. |
| *pixel2group*() | |
| *read_image*(label_image) | |
| *save_microstructure*([name]) | |
| *save_pixelgroups*([name]) | **Parameters name** (*str*) – DESCRIPTION. |
| *show*() | **returns** *None.* |
| *write_script*([name]) | |

## Attributes

| |
|---|
| *script* |

**class** grains.meshing.**FixedDict**(*dictionary*)
> Bases: object

> https://stackoverflow.com/a/14816446/4892892

**class** grains.meshing.**OOF2**
> Bases: object

> **create_material**(*name*)

>> **Parameters name** (*TYPE*) – DESCRIPTION.

>> **Raises Exception** – DESCRIPTION.

>> **Returns** *None.*

> **create_microstructure**(*name=None*)
>> Creates a microstructure from an image.

>> **Parameters name** (*str, optional*) – Path to the image on which the micrstucture is created, file extension included. If not given, the microstructure is given the same name as the input image.

>> **Raises Exception** – DESCRIPTION.

>> **Returns** *None.*

> **create_skeleton**(*nelem_x*, *nelem_y*, *geometry*, *name=None*)

> **load_pixelgroups**(*microstructure_file*)

Parameters **microstructure_file** (*str*) – DESCRIPTION.

Returns *None.*

**materials2groups**(*materials*, *groups=None*)

Parameters

- **materials** (*list of str*) – DESCRIPTION.

- **groups** (*list of int, optional*) – DESCRIPTION. The default is None.

Returns *None.*

**pixel2group**()

**read_image**(*label_image*)

**save_microstructure**(*name=None*)

**save_pixelgroups**(*name=None*)

Parameters **name** (*str*) – DESCRIPTION.

Returns *None.*

**script = []**

**show**()

Returns *None.*

**write_script**(*name=None*)

**class** grains.meshing.**QuadSkeletonGeometry**(*leftright_periodicity=False*, *topbottom_periodicity=False*)

Bases: *grains.meshing.SkeletonGeometry*

**class** grains.meshing.**SkeletonGeometry**(*leftright_periodicity*, *topbottom_periodicity*)

Bases: abc.ABC

**class** grains.meshing.**TriSkeletonGeometry**(*leftright_periodicity=False*, *topbottom_periodicity=False*, *arrangement='conservative'*)

Bases: *grains.meshing.SkeletonGeometry*

grains.meshing.**nt**

alias of grains.meshing.modules

# CAD

This module contains functions to provide geometrical descriptions of a label image. Each labelled region is turned to a surface. This allows mesh generators to work on the geometry instead of an image, thereby creating good quality meshes.

The following functions require PythonOCC version 0.18 to be installed to allow spline manipulations:

- *fit_spline*
- *branches2splines*
- *region_as_splinegon*
- *write_step_file*
- *regions2step*
- *plot_splinegons*
- *splinegonize*
- *face_area*
- *subtract*
- *make_holes*

## 19.1 Functions

| | |
|---|---|
| *build_skeleton*(label_image[, connectivity, . . . ]) | Builds skeleton connectivity of a label image. |
| *skeleton2regions*(skeleton_network, . . . ) | Determines the regions bounded by a skeleton network. |
| *polygon_orientation*(polygon) | Determines whether a polygon is oriented clockwise or counterclockwise. |
| *branches2boundary*(branches[, orientation]) | Joins connecting branches so that they form the boundary of a surface. |
| *region_as_polygon*(branches[, orientation, close]) | Represents a region as a polygon. |
| *polygonize*(label_image, . . . [, connectivity, . . . ]) | Polygon representation of a label image. |
| *branches2splines*(branches[, degree_min, . . . ]) | Lays splines on branches. |
| *fit_spline*(points[, degree_min, degree_max, . . . ]) | Approximates a set of points on the plane with a B-spline. |
| *region_as_splinegon*(boundary_splines) | Represents a region as a splinegon. |
| *make_holes*(splinegons) | Creates holes for splinegons that contain other splinegons. |

| Table 1 – continued from previous page | |
|---|---|
| *splinegonize*(label_image, … [, … ]) | Polygon representation of a label image. |
| *regions2step*(splinegons, filename[, … ]) | Writes splinegon regions to a STEP file. |
| *plot_polygons*(polygons, **kwargs) | Plots polygons. |
| *plot_splinegons*(splinegons[, transparency, … ]) | Plots splinegons. |
| *write_step_file*(shape, filename[, … ]) | Exports a shape to a STEP file. |
| *plot_polygon*(vertices[, ax]) | Plots a polygon. |
| *overlay_regions*(label_image, polygons[, axes]) | Plots a label image, and overlays polygonal regions over it. |
| *search_neighbor*(radius, norm[, method]) | Neighbor search algorithm. |
| *face_area*(face) | Area of a face. |
| *subtract*(shape1, shape2) | Boolean difference of two shapes. |

## 19.1.1 grains.cad.build_skeleton

grains.cad.**build_skeleton**(*label_image*, *connectivity=1*, *detect_boundaries=True*)

Builds skeleton connectivity of a label image.

A single-pixel wide network is created, separating the labelled image regions. The resulting network contains information about how the regions are connected.

> **Parameters**
>
> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> - **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.
>
> - **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.
>
> **Returns  skeleton_network** (*Skeleton*) – Geometrical and topological information about the skeleton network of the input image.

> **See also:**
>
> skan.csr.Skeleton

## 19.1.2 grains.cad.skeleton2regions

grains.cad.**skeleton2regions**(*skeleton_network*, *neighbor_search_algorithm*)

Determines the regions bounded by a skeleton network.

This function can be perceived as an intermediate step between a skeleton network and completely geometrical representation of the regions. That is, it keeps the key topological information required to create a fully geometrical description, but it also contains the coordinates of the region boundaries. The outputs of this function can be used to build different region representations.

> **Parameters**
>
> - **skeleton_network** (*Skeleton*) – Geometrical and topological information about the skeleton network of a label image.

- **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is *search_neighbor()*. For further details, see the Notes below.

**Returns**

- **region_branches** (*dict*) – For each region it contains the branch indices that bound that region.

- **branch_coordinates** (*list*) – Coordinates of the points on each branch.

- **branch_regions** (*dict*) – For each branch it contains the neighboring regions. This auxiliary data is not essential as it can be restored from `region_branches`. However, it is computed as temporary data needed for `region_branches`.

**See also:**

*build_skeleton()*, *search_neighbor()*, *overlay_regions()*

### Notes

Although the algorithms were created to require minimum user intervention, some parameters must be fine-tuned so as to achieve an optimal result in identifying the regions. Visualization plays an important role in it. Full automation is either not possible or would require a huge computational cost. The shortcoming of the algorithms in this function is the following. The recognition of which branches form a region is based on the premise that a node of a branch belongs to a region if its *n*-pixel neighbourhood contains a pixel from that region. Ideally, *n=1* would be used, meaning that the single-pixel width skeleton is located at most 1 pixel afar from the regions it lies among. This is true but the nodes of the skeleton can be farther than 1 pixel from a region. Hence, *n* has to be a parameter of our model. Increasing *n* helps in identifying the connecting regions to a node of a branch. On the other hand, if *n* is too large, regions that "in reality" are not neighbors of a branch will be included. Currently, we recommend trying different parameters *n*, plot the reconstructed regions over the label image using the *overlay_regions()* function, and see how good the result is. As a heuristic, start with *n=2*.

## 19.1.3 grains.cad.polygon_orientation

grains.cad.**polygon_orientation**(*polygon*)
    Determines whether a polygon is oriented clockwise or counterclockwise.

    **Parameters polygon** (*list*) – Each element of the list denotes a vertex of the polygon and in turn is another list of two elements: the x and y coordinates of a vertex.

    **Returns orientation** (*{'cw', 'ccw'}*) – 'cw': clockwise, 'ccw': counterclockwise orientation

### Notes

The formula to determine the orientation is from https://stackoverflow.com/a/1165943/4892892. For simple polygons (polygons that admit a well-defined interior), a faster algorithm exits, see https://en.wikipedia.org/wiki/Curve_orientation#Orientation_of_a_simple_polygon.

```
>>> polygon = [[5, 0], [6, 4], [4, 5], [1, 5], [1, 0]]
>>> polygon_orientation(polygon)
'ccw'
```

## 19.1.4 grains.cad.branches2boundary

grains.cad.**branches2boundary**(*branches*, *orientation='ccw'*)

Joins connecting branches so that they form the boundary of a surface.

This function assumes that you already know that a set of branches form the boundary of a surface, but you want to know the ordering. Clockwise or counterclockwise orientation is supported. A branch is given by its two end points but it can also contain intermediate points in between, as in the general case. The points on the branches are assumed to be ordered. If certain branches are not used in forming the boundary of a surface, they are excluded from the output lists.

**Parameters**

- **branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points.
- **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the surface boundary. The default is 'ccw'.

**Returns**

- **order** (*list*) – Order of the branches so that they form the boundary of a surface.
- **redirected** (*list*) – A list of bool with True value if the orientation of the corresponding branch had to be swapped to form the surface boundary.
- **polygon** (*ndarray*) – The polygon formed by connecting the points of the branches along the boundary. It is given given as an Mx2 ndarray, where M is the number of unique points on the boundary (i.e. only one end point is kept for two connecting branches). This auxiliary data is not essential as it can be restored from the set of branches, and their ordering. However, it is computed as temporary data needed for determining the orientation of the boundary.

**Examples**

```
>>> import numpy as np
>>> branches = [np.array([[1, 1], [1.5, 2], [2, 3]]), np.array([[1, 1], [-1, 2]]),
... np.array([[1.5, -3], [2, 3]]), np.array([[1.5, -3], [-1, 2]])]
>>> order, redirected, polygon = branches2boundary(branches, orientation='cw')
>>> order
[0, 2, 3, 1]
>>> redirected
[False, True, True, False]
>>> polygon
array([[ 1. ,  1. ],
       [ 1.5,  2. ],
       [ 2. ,  3. ],
```

```
       [ 1.5, -3. ],
       [-1. ,  2. ]])
```

## 19.1.5 grains.cad.region_as_polygon

grains.cad.**region_as_polygon**(*branches*, *orientation='ccw'*, *close='False'*)

> Represents a region as a polygon.
>
> Based on the combined topological-geometrical (intermediate) representation of the regions, (done by *skeleton2regions()*), this function provides a fully geometrical description of a region.
>
> > **Parameters**
> >
> > - **branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points. The branches are not necessarily ordered.
> >
> > - **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the polygons. The default is 'ccw'.
> >
> > - **close** (*bool, optional*) – When True, one vertex in the polygons is repeated to indicate that the polygons are indeed closed. The default is False.
> >
> > **Returns polygon** (*ndarray*) – The resulting polygon, given as an Mx2 ndarray, where M is the number of unique points on the polygon (i.e. only one end point is kept for two connecting branches).
>
> **See also:**
>
> *skeleton2regions()*

## 19.1.6 grains.cad.polygonize

grains.cad.**polygonize**(*label_image*, *neighbor_search_algorithm*, *connectivity=1*, *detect_boundaries=True*, *orientation='ccw'*, *close=False*)

> Polygon representation of a label image.
>
> > **Parameters**
> >
> > - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
> >
> > - **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is *skeleton2regions()*.
> >
> > - **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.
> >
> > - **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.
> >
> > - **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the polygons. The default is 'ccw'.

- **close** (*bool, optional*) – When True, one vertex in the polygons is repeated to indicate that the polygons are indeed closed. The default is False.

> **Returns polygons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are ndarray objects with two columns, the x and y coordinates of the polygons.

**See also:**

*build_skeleton()*, *skeleton2regions()*, *region_as_polygon()*

### Examples

```
>>> test_image = np.array([
...    [1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]],
...    dtype=np.int8)
>>> polygons = polygonize(test_image, search_neighbor(2, np.inf), connectivity=1)
```

## 19.1.7 grains.cad.branches2splines

grains.cad.**branches2splines**(*branches*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.001*)

> Lays splines on branches.

> **Parameters branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points.

> **Other Parameters degree_min, degree_max, continuity, tol** – See the *fit_spline()* function.

> **Returns** *list* – Each member of the list is a *Geom_BSplineCurve* object, the B-spline approximation of the corresponding input branch. For details on the resulting spline, see the Open-CASCADE documentation: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_geom___b_spline_curve.html

**See also:**

*fit_spline()*

## 19.1.8 grains.cad.fit_spline

grains.cad.**fit_spline**(*points*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.001*)

Approximates a set of points on the plane with a B-spline.

> **Parameters**
>
> - **points** (*ndarray*) – 2D ndarray with N>=2 rows and 2 columns, representing the points on the plane, ordered from one end point to the other. The first column gives the x, the second column gives the the y coordinates of the points.
>
> - **degree_min** (*int, optional*) – Minimum degree of the spline. The default is 3.
>
> - **degree_max** (*int, optional*) – Maximum degree of the spline. The default is 8.
>
> - **continuity** (*{'C0', 'G1', 'C1', 'G2', 'C2', 'C3', 'CN'}, optional*) – The continuity of the spline will be at least *continuity*. The default is 'C2'. For their meanings, consult with https://www.opencascade.com/doc/occt-7.4.0/refman/html/_geom_abs___shape_8hxx.html
>
> - **tol** (*float, optional*) – The distance from the points to the spline will be lower than *tol*. The default is 1e-3.
>
> **Returns spline** (*Geom_BSplineCurve*) – For details on the resulting spline, see the Open-CASCADE documentation: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_geom___b_spline_curve.html
>
> **Raises** `ValueError` – If the minimum degree of the B-spline to be constructed is greater than its maximum degree.

## 19.1.9 grains.cad.region_as_splinegon

grains.cad.**region_as_splinegon**(*boundary_splines*)

Represents a region as a splinegon.

Based on the combined topological-geometrical (intermediate) representation of the regions, (done by `skeleton2regions()`), this function provides a fully geometrical description of a region.

> **Parameters boundary_splines** (*list*) – Each element of the list is a *Handle_Geom_BSplineCurve* object, giving a reference to the B-splines bounding the region. The splines must either be ordered (see `branches2boundary()`) or they must appear in an order such that the n-th spline in the list can be connected to one of the first n-1 splines in the list.
>
> **Returns**
>
> - **splinegon** (*TopoDS_Face*) – The resulting splinegon (surface). For details on the object, see the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_topo_d_s___face.html
>
> - **boundary** (*TopoDS_Wire*) – The boundary of the splinegon. For details on the resulting object, see the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_topo_d_s___wire.html

**See also:**

`skeleton2regions()`, `fit_spline()`

**Notes**

The syntax *Handle_classname* in PythonOCC corresponds to wrapping the object of class *classname* with a smart pointer. In the C++ interface, it is done by a template: *Handle<classname>*.

### 19.1.10 grains.cad.make_holes

grains.cad.**make_holes**(*splinegons*)
    Creates holes for splinegons that contain other splinegons.

When a smaller splinegon is embedded to a larger one, the boundary of the larger splinegon does not describe it properly. The smaller splinegons have to be subtracted to reproduce the non-simply connected domain. As an analogy to material science, the intended application of this module, we will use the term *matrix* for the larger grain and *inclusion* for the smaller one. This function handles multiple inclusion in a matrix and should be called after the splinegon regions have been constructed. No need to directly call `make_holes()` if you use the `splinegonize()` utility function.

---

**Todo:** Implement it more generally, see the related issue.

---

> **Parameters splinegons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Face* objects, the surfaces of the regions.

> **Returns splinegons** (*dict*) – The input dictionary modified such that the inclusions are removed from the matrix.

### 19.1.11 grains.cad.splinegonize

grains.cad.**splinegonize**(*label_image*,    *neighbor_search_algorithm*,    *connectivity=1*,    *detect_boundaries=True*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.0001*)
    Polygon representation of a label image.

> **Parameters**
>
> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> - **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is `skeleton2regions()`.
>
> - **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.
>
> - **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.
>
> **Other Parameters degree_min, degree_max, continuity, tol** – See the `fit_spline()` function.
>
> **Returns**
>
> - **splinegons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Face* objects, the surfaces of the regions.

- **boundaries** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Wire* objects, the boundaries of the regions.

**See also:**

*build_skeleton()*, *skeleton2regions()*, *region_as_splinegon()*

**Examples**

```
>>> test_image = np.array([
...    [1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...    [2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]],
...    dtype=np.int8)
>>> splinegons, _ = splinegonize(test_image, search_neighbor(2, np.inf),
→connectivity=1, tol=0.1)
>>> plot_splinegons(list(splinegons.values()))
```

## 19.1.12 grains.cad.regions2step

grains.cad.**regions2step**(*splinegons*, *filename*, *application_protocol='AP203'*)
    Writes splinegon regions to a STEP file.

   **Parameters**

- **splinegons** (*list*) – Each member of the list is a *TopoDS_Face* object, the surface of a region.

- **filename** (*str*) – Name of the file the regions are saved into.

- **application_protocol** (*{'AP203', 'AP214IS', 'AP242DIS'}, optional*) – Version of schema used for the output STEP file. The default is 'AP203'.

   **Returns** *None*

**See also:**

*splinegonize()*, *write_step_file()*

## 19.1.13 grains.cad.plot_polygons

grains.cad.**plot_polygons**(*polygons*, *\*\*kwargs*)

Plots polygons.

> **Parameters  polygons** (*list*) – Each member of the list is a 2D numpy array with 2 columns, each row corresponding to a vertex of the polygon, and the two columns giving the Cartesian coordinates of the vertices.
>
> **Other Parameters  kwargs** (*optional*) – Keyword arguments supported by the constructor of the `matplotlib.figure.Figure` class.
>
> **Returns  fig** (*matplotlib.figure.Figure*) – Figure the polygons are plotted into.

See also:

`polygonize()`

## 19.1.14 grains.cad.plot_splinegons

grains.cad.**plot_splinegons**(*splinegons*, *transparency=0.5*, *color='random'*)

Plots splinegons.

> **Parameters**
>
> - **splinegons** (*list*) – Each member of the list is a *TopoDS_Face* object, the surface of a splinegon.
>
> - **transparency** (*float, optional*) – The transparency value should be between 0 and 1. At 0, an object will be totally opaque, and at 1, fully transparent.
>
> - **color** (*tuple or str, optional*) – Color of the splinegons. Either 'random' to associate a random color for each splinegon in the list *splinegons* or a 3-tuple with entries between 0 and 1 to give the same RGB values to all the splinegons.
>
> **Returns**  *None*

See also:

`splinegonize()`

## 19.1.15 grains.cad.write_step_file

grains.cad.**write_step_file**(*shape*, *filename*, *application_protocol='AP203'*)

Exports a shape to a STEP file.

> **Parameters**
>
> - **shape** (*TopoDS_Shape*) – Shape to be exported, an object of the *TopoDS_Shape* class or one of its subclasses. See https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_topo_d_s___shape.html
>
> - **filename** (*str*) – Name of the file the shape is saved into.
>
> - **application_protocol** (*{'AP203', 'AP214IS', 'AP242DIS'}, optional*) – Version of schema used for the output STEP file. The default is 'AP203'.

**Notes**

For details on how to read and write STEP files, see the documentation on https://dev.opencascade.org/doc/overview/html/occt_user_guides__step.html.

## 19.1.16 grains.cad.plot_polygon

grains.cad.**plot_polygon**(*vertices*, *ax=None*, ***kwargs*)

Plots a polygon.

> **Parameters**
>
> - **vertices** (*ndarray*) – 2D ndarray of size Nx2, with each row designating a vertex and the two columns giving the x and y coordinates of the vertices, respectively.
>
> - **ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the polygon is plotted into. The default is None, in which case a new *Axes* within a new figure is created.
>
> - ****kwargs** (*Line2D properties, optional*) – Keyword arguments accepted by matplotlib.pyplot.plot
>
> **Returns** *None*

See also:

matplotlib.pyplot.plot()

**Examples**

```
>>> plot_polygon(np.array([[1, 1], [2, 3], [1.5, -3], [-1, 2]]), marker='o');
→plt.show()
```

## 19.1.17 grains.cad.overlay_regions

grains.cad.**overlay_regions**(*label_image*, *polygons*, *axes=None*)

Plots a label image, and overlays polygonal regions over it.

> **Parameters**
>
> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> - **polygons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are ndarray objects with two columns, the x and y coordinates of the polygons. This format is respected by the output of the *polygonize* function.
>
> - **axes** (*matplotlib.axes.Axes, optional*) – An Axes object on which to draw. If None, a new one is created.
>
> **Returns** **axes** (*matplotlib.axes.Axes*) – The Axes object on which the plot is drawn.

See also:

*polygonize()*, matplotlib.collections.LineCollection

### 19.1.18 grains.cad.search_neighbor

grains.cad.**search_neighbor**(*radius*, *norm*, *method='sphere'*)

> Neighbor search algorithm.
>
> > **Parameters radius, norm, method** – See the `grains.utils.neighborhood()` function.
> >
> > **Returns** *functools.partial* – A new function with partial application of the given input arguments on the `grains.utils.neighborhood()` function. The returned partial function is used internally by `skeleton2regions()` when performing neighbor searching.
>
> **See also:**
>
> `grains.utils.neighborhood()`, `skeleton2regions()`

### 19.1.19 grains.cad.face_area

grains.cad.**face_area**(*face*)

> Area of a face.
>
> > **Parameters face** (*TopoDS_Face*) – The face for which the area is searched.
> >
> > **Returns** *float* – Area of the face.
>
> #### Notes
>
> For references, see
>
> - https://dev.opencascade.org/doc/refman/html/class_b_rep_g_prop.html#abd91b892df8d0f6b8571deed5562ca1f
> - https://techoverflow.net/2019/06/13/how-to-compute-surface-area-of-topods_face-in-opencascade/
> - https://github.com/tpaviot/pythonocc-demos/blob/master/examples/core_shape_properties.py#L48

### 19.1.20 grains.cad.subtract

grains.cad.**subtract**(*shape1*, *shape2*)

> Boolean difference of two shapes.
>
> > **Parameters shape1, shape2** (*TopoDS_Shape*) – Surfaces.
> >
> > **Returns difference** (*TopoDS_Shape*) – The set difference `shape1 \ shape2`.
>
> #### Notes
>
> The implementation follows the following sources:
>
> - https://techoverflow.net/2019/06/14/how-to-fuse-topods_shapes-in-opencascade-boolean-and/
> - https://github.com/tpaviot/pythonocc-demos/blob/master/examples/core_boolean_fuzzy_cut_emmenthaler.py#L41-L54
>
> For Boolean operations in Open CASCADE, see its documentation.

grains.cad.**_face_error**(*error_code*)

> Indicates the outcome of the face construction.
>
> > **Parameters error_code** (*int*) – Error code returned by the *Error* method of *BRepBuilder-API_MakeFace*.

**Returns** *None*

**Raises** `ValueError` – If the error code is 1, 2, 3 or 4. See the OpenCAS-CADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/_b_rep_builder_a_p_i__face_error_8hxx.html The exception is raised within this function if the face could not be created.

grains.cad.**_spline_continuity_enum**(*continuity*)
    Enumeration value corresponding to the continuity of a B-spline.

**Parameters continuity** (*{'C0', 'G1', 'C1', 'G2', 'C2', 'C3', 'CN'}*) – Continuity of the B-spline.

**Returns enum** (*int*) – Integer value corresponding to continuity key in the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/_geom_abs___shape_8hxx.html

grains.cad.**_wire_error**(*error_code*)
    Indicates the outcome of the wire construction.

**Parameters error_code** (*int*) – Error code returned by the *Error* method of *BRepBuilder-API_MakeWire*.

**Returns** *None*

**Raises** `ValueError` – If the error code is 1, 2 or 3. See the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/_b_rep_builder_a_p_i___wire_error_8hxx.html The exception is raised within this function, as an incompatible wire makes it impossible to construct a valid surface.

grains.cad.**branches2boundary**(*branches*, *orientation='ccw'*)
    Joins connecting branches so that they form the boundary of a surface.

    This function assumes that you already know that a set of branches form the boundary of a surface, but you want to know the ordering. Clockwise or counterclockwise orientation is supported. A branch is given by its two end points but it can also contain intermediate points in between, as in the general case. The points on the branches are assumed to be ordered. If certain branches are not used in forming the boundary of a surface, they are excluded from the output lists.

    **Parameters**

    • **branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points.

    • **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the surface boundary. The default is 'ccw'.

    **Returns**

    • **order** (*list*) – Order of the branches so that they form the boundary of a surface.

    • **redirected** (*list*) – A list of bool with True value if the orientation of the corresponding branch had to be swapped to form the surface boundary.

    • **polygon** (*ndarray*) – The polygon formed by connecting the points of the branches along the boundary. It is given given as an Mx2 ndarray, where M is the number of unique points on the boundary (i.e. only one end point is kept for two connecting branches). This auxiliary data is not essential as it can be restored from the set of branches, and their ordering. However, it is computed as temporary data needed for determining the orientation of the boundary.

**Examples**

```
>>> import numpy as np
>>> branches = [np.array([[1, 1], [1.5, 2], [2, 3]]), np.array([[1, 1], [-1, 2]]),
... np.array([[1.5, -3], [2, 3]]), np.array([[1.5, -3], [-1, 2]])]
>>> order, redirected, polygon = branches2boundary(branches, orientation='cw')
>>> order
[0, 2, 3, 1]
>>> redirected
[False, True, True, False]
>>> polygon
array([[ 1. ,   1. ],
       [ 1.5,   2. ],
       [ 2. ,   3. ],
       [ 1.5, -3. ],
       [-1. ,   2. ]])
```

grains.cad.**branches2splines**(*branches*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.001*)

Lays splines on branches.

> **Parameters branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points.
>
> **Other Parameters degree_min, degree_max, continuity, tol** – See the *fit_spline()* function.
>
> **Returns** *list* – Each member of the list is a *Geom_BSplineCurve* object, the B-spline approximation of the corresponding input branch. For details on the resulting spline, see the Open-CASCADE documentation: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_geom___b_spline_curve.html

See also:

*fit_spline()*

grains.cad.**build_skeleton**(*label_image*, *connectivity=1*, *detect_boundaries=True*)

Builds skeleton connectivity of a label image.

A single-pixel wide network is created, separating the labelled image regions. The resulting network contains information about how the regions are connected.

> **Parameters**
>
> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>
> - **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.
>
> - **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.
>
> **Returns skeleton_network** (*Skeleton*) – Geometrical and topological information about the skeleton network of the input image.

See also:

skan.csr.Skeleton

---

grains.cad.**face_area**(*face*)

 Area of a face.

> **Parameters face** (*TopoDS_Face*) – The face for which the area is searched.
>
> **Returns** *float* – Area of the face.

#### Notes

For references, see

- https://dev.opencascade.org/doc/refman/html/class_b_rep_g_prop.html#abd91b892df8d0f6b8571deed5562ca1f

- https://techoverflow.net/2019/06/13/how-to-compute-surface-area-of-topods_face-in-opencascade/

- https://github.com/tpaviot/pythonocc-demos/blob/master/examples/core_shape_properties.py#L48

grains.cad.**fit_spline**(*points*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.001*)

 Approximates a set of points on the plane with a B-spline.

> **Parameters**
>
> - **points** (*ndarray*) – 2D ndarray with N>=2 rows and 2 columns, representing the points on the plane, ordered from one end point to the other. The first column gives the x, the second column gives the the y coordinates of the points.
>
> - **degree_min** (*int, optional*) – Minimum degree of the spline. The default is 3.
>
> - **degree_max** (*int, optional*) – Maximum degree of the spline. The default is 8.
>
> - **continuity** (*{'C0', 'G1', 'C1', 'G2', 'C2', 'C3', 'CN'}, optional*) – The continuity of the spline will be at least *continuity*. The default is 'C2'. For their meanings, consult with https://www.opencascade.com/doc/occt-7.4.0/refman/html/_geom_abs___shape_8hxx.html
>
> - **tol** (*float, optional*) – The distance from the points to the spline will be lower than *tol*. The default is 1e-3.
>
> **Returns spline** (*Geom_BSplineCurve*) – For details on the resulting spline, see the OpenCASCADE documentation: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_geom___b_spline_curve.html
>
> **Raises** `ValueError` – If the minimum degree of the B-spline to be constructed is greater than its maximum degree.

grains.cad.**make_holes**(*splinegons*)

 Creates holes for splinegons that contain other splinegons.

When a smaller splinegon is embedded to a larger one, the boundary of the larger splinegon does not describe it properly. The smaller splinegons have to be subtracted to reproduce the non-simply connected domain. As an analogy to material science, the intended application of this module, we will use the term *matrix* for the larger grain and *inclusion* for the smaller one. This function handles multiple inclusion in a matrix and should be called after the splinegon regions have been constructed. No need to directly call `make_holes()` if you use the `splinegonize()` utility function.

---

**Todo:** Implement it more generally, see the related issue.

---

> **Parameters splinegons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Face* objects, the surfaces of the regions.

> **Returns splinegons** (*dict*) – The input dictionary modified such that the inclusions are removed from the matrix.

grains.cad.**overlay_regions** (*label_image*, *polygons*, *axes=None*)

> Plots a label image, and overlays polygonal regions over it.

> **Parameters**

>> • **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.

>> • **polygons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are ndarray objects with two columns, the x and y coordinates of the polygons. This format is respected by the output of the *polygonize* function.

>> • **axes** (*matplotlib.axes.Axes, optional*) – An Axes object on which to draw. If None, a new one is created.

> **Returns axes** (*matplotlib.axes.Axes*) – The Axes object on which the plot is drawn.

> **See also:**

> *polygonize()*, matplotlib.collections.LineCollection

grains.cad.**plot_polygon** (*vertices*, *ax=None*, *\*\*kwargs*)

> Plots a polygon.

> **Parameters**

>> • **vertices** (*ndarray*) – 2D ndarray of size Nx2, with each row designating a vertex and the two columns giving the x and y coordinates of the vertices, respectively.

>> • **ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the polygon is plotted into. The default is None, in which case a new *Axes* within a new figure is created.

>> • **\*\*kwargs** (*Line2D properties, optional*) – Keyword arguments accepted by matplotlib.pyplot.plot

> **Returns** *None*

> **See also:**

> matplotlib.pyplot.plot()

> **Examples**

> ```
> >>> plot_polygon(np.array([[1, 1], [2, 3], [1.5, -3], [-1, 2]]), marker='o'); ␣
> ↪plt.show()
> ```

grains.cad.**plot_polygons** (*polygons*, *\*\*kwargs*)

> Plots polygons.

> **Parameters polygons** (*list*) – Each member of the list is a 2D numpy array with 2 columns, each row corresponding to a vertex of the polygon, and the two columns giving the Cartesian coordinates of the vertices.

> **Other Parameters kwargs** (*optional*) – Keyword arguments supported by the constructor of the matplotlib.figure.Figure class.

> **Returns fig** (*matplotlib.figure.Figure*) – Figure the polygons are plotted into.

**See also:**

*polygonize()*

grains.cad.**plot_splinegons**(*splinegons*, *transparency=0.5*, *color='random'*)

Plots splinegons.

> **Parameters**
>
>> - **splinegons** (*list*) – Each member of the list is a *TopoDS_Face* object, the surface of a splinegon.
>>
>> - **transparency** (*float, optional*) – The transparency value should be between 0 and 1. At 0, an object will be totally opaque, and at 1, fully transparent.
>>
>> - **color** (*tuple or str, optional*) – Color of the splinegons. Either 'random' to associate a random color for each splinegon in the list *splinegons* or a 3-tuple with entries between 0 and 1 to give the same RGB values to all the splinegons.
>
> **Returns** *None*

**See also:**

*splinegonize()*

grains.cad.**polygon_orientation**(*polygon*)

Determines whether a polygon is oriented clockwise or counterclockwise.

> **Parameters  polygon** (*list*) – Each element of the list denotes a vertex of the polygon and in turn is another list of two elements: the x and y coordinates of a vertex.
>
> **Returns  orientation** (*{'cw', 'ccw'}*) – 'cw': clockwise, 'ccw': counterclockwise orientation

#### Notes

The formula to determine the orientation is from https://stackoverflow.com/a/1165943/4892892. For simple polygons (polygons that admit a well-defined interior), a faster algorithm exits, see https://en.wikipedia.org/wiki/Curve_orientation#Orientation_of_a_simple_polygon.

#### Examples

```
>>> polygon = [[5, 0], [6, 4], [4, 5], [1, 5], [1, 0]]
>>> polygon_orientation(polygon)
'ccw'
```

grains.cad.**polygonize**(*label_image*, *neighbor_search_algorithm*, *connectivity=1*, *detect_boundaries=True*, *orientation='ccw'*, *close=False*)

Polygon representation of a label image.

> **Parameters**
>
>> - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.
>>
>> - **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is *skeleton2regions()*.
>>
>> - **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.

- **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.

- **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the polygons. The default is 'ccw'.

- **close** (*bool, optional*) – When True, one vertex in the polygons is repeated to indicate that the polygons are indeed closed. The default is False.

    **Returns polygons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are ndarray objects with two columns, the x and y coordinates of the polygons.

**See also:**

*build_skeleton()*, *skeleton2regions()*, *region_as_polygon()*

### Examples

```
>>> test_image = np.array([
...     [1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]],
...     dtype=np.int8)
>>> polygons = polygonize(test_image, search_neighbor(2, np.inf), connectivity=1)
```

grains.cad.**region_as_polygon**(*branches*, *orientation='ccw'*, *close='False'*)
    Represents a region as a polygon.

Based on the combined topological-geometrical (intermediate) representation of the regions, (done by *skeleton2regions()*), this function provides a fully geometrical description of a region.

    **Parameters**

- **branches** (*list*) – Each element of the list gives N>=2 points on the branch, ordered from one end point to the other. If N=2, the two end points are meant. The points are provided as an Nx2 ndarray, the first column giving the x, the second column giving the y coordinates of the points. The branches are not necessarily ordered.

- **orientation** (*{'cw', 'ccw'}, optional*) – Clockwise ('cw') or counterclockwise ('ccw') orientation of the polygons. The default is 'ccw'.

- **close** (*bool, optional*) – When True, one vertex in the polygons is repeated to indicate that the polygons are indeed closed. The default is False.

> > **Returns polygon** (*ndarray*) – The resulting polygon, given as an Mx2 ndarray, where M is the number of unique points on the polygon (i.e. only one end point is kept for two connecting branches).

> **See also:**

> *skeleton2regions()*

grains.cad.**region_as_splinegon**(*boundary_splines*)

> Represents a region as a splinegon.

> Based on the combined topological-geometrical (intermediate) representation of the regions, (done by *skeleton2regions()*), this function provides a fully geometrical description of a region.

> > **Parameters boundary_splines** (*list*) – Each element of the list is a *Handle_Geom_BSplineCurve* object, giving a reference to the B-splines bounding the region. The splines must either be ordered (see *branches2boundary()*) or they must appear in an order such that the n-th spline in the list can be connected to one of the first n-1 splines in the list.

> > **Returns**

> > > - **splinegon** (*TopoDS_Face*) – The resulting splinegon (surface). For details on the object, see the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_topo_d_s___face.html

> > > - **boundary** (*TopoDS_Wire*) – The boundary of the splinegon. For details on the resulting object, see the OpenCASCADE API: https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_topo_d_s___wire.html

> **See also:**

> *skeleton2regions()*, *fit_spline()*

> ### Notes

> The syntax *Handle_classname* in PythonOCC corresponds to wrapping the object of class *classname* with a smart pointer. In the C++ interface, it is done by a template: *Handle<classname>*.

grains.cad.**regions2step**(*splinegons*, *filename*, *application_protocol='AP203'*)

> Writes splinegon regions to a STEP file.

> > **Parameters**

> > > - **splinegons** (*list*) – Each member of the list is a *TopoDS_Face* object, the surface of a region.

> > > - **filename** (*str*) – Name of the file the regions are saved into.

> > > - **application_protocol** (*{'AP203', 'AP214IS', 'AP242DIS'}, optional*) – Version of schema used for the output STEP file. The default is 'AP203'.

> > **Returns** *None*

> **See also:**

> *splinegonize()*, *write_step_file()*

grains.cad.**search_neighbor**(*radius*, *norm*, *method='sphere'*)

> Neighbor search algorithm.

> > **Parameters radius, norm, method** – See the *grains.utils.neighborhood()* function.

> > **Returns** *functools.partial* – A new function with partial application of the given input arguments on the *grains.utils.neighborhood()* function. The returned partial function is used internally by *skeleton2regions()* when performing neighbor searching.

---

**See also:**

*grains.utils.neighborhood()*, *skeleton2regions()*

grains.cad.**skeleton2regions**(*skeleton_network*, *neighbor_search_algorithm*)
    Determines the regions bounded by a skeleton network.

    This function can be perceived as an intermediate step between a skeleton network and completely geometrical representation of the regions. That is, it keeps the key topological information required to create a fully geometrical description, but it also contains the coordinates of the region boundaries. The outputs of this function can be used to build different region representations.

    **Parameters**

    - **skeleton_network** (*Skeleton*) – Geometrical and topological information about the skeleton network of a label image.

    - **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is *search_neighbor()*. For further details, see the Notes below.

    **Returns**

    - **region_branches** (*dict*) – For each region it contains the branch indices that bound that region.

    - **branch_coordinates** (*list*) – Coordinates of the points on each branch.

    - **branch_regions** (*dict*) – For each branch it contains the neighboring regions. This auxiliary data is not essential as it can be restored from region_branches. However, it is computed as temporary data needed for region_branches.

    **See also:**

    *build_skeleton()*, *search_neighbor()*, *overlay_regions()*

    ### Notes

    Although the algorithms were created to require minimum user intervention, some parameters must be fine-tuned so as to achieve an optimal result in identifying the regions. Visualization plays an important role in it. Full automation is either not possible or would require a huge computational cost. The shortcoming of the algorithms in this function is the following. The recognition of which branches form a region is based on the premise that a node of a branch belongs to a region if its $n$-pixel neighbourhood contains a pixel from that region. Ideally, $n=1$ would be used, meaning that the single-pixel width skeleton is located at most 1 pixel afar from the regions it lies among. This is true but the nodes of the skeleton can be farther than 1 pixel from a region. Hence, $n$ has to be a parameter of our model. Increasing $n$ helps in identifying the connecting regions to a node of a branch. On the other hand, if $n$ is too large, regions that "in reality" are not neighbors of a branch will be included. Currently, we recommend trying different parameters $n$, plot the reconstructed regions over the label image using the *overlay_regions()* function, and see how good the result is. As a heuristic, start with $n=2$.

grains.cad.**splinegonize**(*label_image*, *neighbor_search_algorithm*, *connectivity=1*, *detect_boundaries=True*, *degree_min=3*, *degree_max=8*, *continuity='C2'*, *tol=0.0001*)
    Polygon representation of a label image.

    **Parameters**

    - **label_image** (*ndarray*) – Labeled input image, represented as a 2D numpy array of positive integers.

- **neighbor_search_algorithm** (*functools.partial*) – Specifies which algorithm to use for constructing the branch-region connectivity. The function to be passed (along with its arguments) is `skeleton2regions()`.

- **connectivity** (*{1,2}, optional*) – A connectivity of 1 (default) means pixels sharing an edge will be considered neighbors. A connectivity of 2 means pixels sharing a corner will be considered neighbors.

- **detect_boundaries** (*bool, optional*) – When True, the image boundaries will be treated as part of the skeleton. This allows identifying boundary regions in the *skeleton2regions* function. The default is True.

**Other Parameters  degree_min, degree_max, continuity, tol** – See the `fit_spline()` function.

**Returns**

- **splinegons** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Face* objects, the surfaces of the regions.

- **boundaries** (*dict*) – The keys in the dictionary correspond to the labels of the input image, while the values are *TopoDS_Wire* objects, the boundaries of the regions.

**See also:**

`build_skeleton()`, `skeleton2regions()`, `region_as_splinegon()`

**Examples**

```
>>> test_image = np.array([
...     [1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3],
...     [2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]],
...    dtype=np.int8)
>>> splinegons, _ = splinegonize(test_image, search_neighbor(2, np.inf),
↪connectivity=1, tol=0.1)
>>> plot_splinegons(list(splinegons.values()))
```

`grains.cad.`**`subtract`**(*shape1*, *shape2*)

> Boolean difference of two shapes.

> > **Parameters  shape1, shape2** (*TopoDS_Shape*) – Surfaces.

> > **Returns  difference** (*TopoDS_Shape*) – The set difference `shape1 \ shape2`.

### Notes

The implementation follows the following sources:

- https://techoverflow.net/2019/06/14/how-to-fuse-topods_shapes-in-opencascade-boolean-and/

- https://github.com/tpaviot/pythonocc-demos/blob/master/examples/core_boolean_fuzzy_cut_
  emmenthaler.py#L41-L54

For Boolean operations in Open CASCADE, see its documentation.

`grains.cad.`**`write_step_file`**(*shape*, *filename*, *application_protocol='AP203'*)
Exports a shape to a STEP file.

> **Parameters**
>
> - **shape** (*TopoDS_Shape*) – Shape to be exported, an object of the *TopoDS_Shape* class or
>   one of its subclasses. See https://www.opencascade.com/doc/occt-7.4.0/refman/html/class_
>   topo_d_s___shape.html
>
> - **filename** (*str*) – Name of the file the shape is saved into.
>
> - **application_protocol** (*{'AP203', 'AP214IS', 'AP242DIS'}, optional*) – Version of schema
>   used for the output STEP file. The default is 'AP203'.

### Notes

For details on how to read and write STEP files, see the documentation on https://dev.opencascade.org/doc/
overview/html/occt_user_guides__step.html.

# MED

Extracting and processing meshes from .med files. The functions were tested on the MEDCoupling API, version 9.4.0.

---

**Todo:** Support renumbering (https://docs.salome-platform.org/latest/dev/MEDCoupling/user/html/data_optimization.html).

---

Getting help:

- This module relies on the Python interface of MEDCoupling. Click here for the latest documentation.

- User's manual for the Python interface

- To know more about the MED file format, which is a specialization of HDF5, see the documentation. For a discussion on the relation between the MED format and the APIS, see this page and that one.

- The definitions, such as *group*, used in this module are from the development guide.

- A (mostly English) tutorial for the Python interface to MEDCoupling is also useful. Particularly interesting are the mesh manipulation examples

- Main page of the documentation

## 20.1 Functions

| | |
|---|---|
| *read_mesh* | Reads a mesh file in .med format. |
| *get_nodes* | Obtains the nodes and the node groups of a mesh. |
| *get_elements* | Obtains the elements for each group of a mesh. |

### 20.1.1 grains.med.read_mesh

grains.med.**read_mesh**(*filename*)

Reads a mesh file in .med format. Only one mesh, the first one, is supported. However, that mesh can contain groups.

> **Parameters filename** (*str*) – Path to the mesh file.

> **Returns** *MEDFileUMesh* – Represents an unstructured mesh. For details, see the manual on https://docs.salome-platform.org/latest/dev/MEDCoupling/developer/classMEDCoupling_1_1MEDFileUMesh.html

## 20.1.2 grains.med.get_nodes

grains.med.**get_nodes**(*mesh*)
    Obtains the nodes and the node groups of a mesh.

> **Parameters**  mesh (*MEDFileUMesh*) – Unstructured mesh.

> **Returns**

> - **nodes** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a node, and the two columns giving the Cartesian coordinates of the nodes.

> - **node_groups** (*dict*) – The keys in the dictionary are the node group names, while the values are list of integers, giving the nodes that belong to the particular group.

> **See also:**

> *get_elements()*, getGroupArr

## 20.1.3 grains.med.get_elements

grains.med.**get_elements**(*mesh*, *numbering='global'*)
    Obtains the elements for each group of a mesh.

    Elements of the same dimension as the mesh are collected (e.g. faces for a 2D mesh).

---

**Todo:**  put those elements that do not belong to any group into an automatically created group

---

**Todo:**  support ordering *elements* in alphabetical order

---

**Todo:**  implement the *'global'* strategy

---

> **Parameters**

> - **mesh** (*MEDFileUMesh*) – Unstructured mesh.

> - **numbering** (*{'global'}, optional*) –

>   **Determines how to allocate element numbers in the mesh.** 'global': numbers the elements without taking into account which group they belong to. Use this strategy if you are not sure whether an element belongs to more than one group. 'group': numbers the elements group-wise. This is much faster than the *'global'* strategy, but use this option if you are sure that the groups of the mesh do not contain common elements.

>   The default is 'global'.

> **Returns**

> - **elements** (*ndarray*) – Element-node connectivities in a 2D numpy array, in which each row corresponds to an element and the columns are the nodes of the elements. It is assumed that all the elements have the same number of nodes.

> - **element_groups** (*dict*) – The keys in the dictionary are the element group names, while the values are list of integers, giving the elements that belong to the particular group.

---

> **Warning:** Currently, elements that do not fit into any groups are discarded.

**See also:**

*get_nodes()*, change_node_numbering()

## Notes

The element-node connectivities are read from the mesh. If you want to change the ordering of the nodes, use the change_node_numbering() function.

Both this and the *get_nodes()* function relies on getGroupsOnSpecifiedLev to obtain the groups based on a parameter, called *meshDimRelToMaxExt*. This parameter designates the relative dimension of the mesh entities whose IDs are required. If it is 1, it denotes the nodes. If 0, entities of the same dimension as the mesh are meant (e.g. group of volumes for a 3D mesh, or group of faces for a 2D mesh). When -1, entities of spatial dimension immediately below that of the mesh are collected (e.g. group of faces for a 3D mesh, or group of edges for a 2D mesh). For -2, entities of two dimensions below that of the mesh are fetched (e.g. group of edges for a 3D mesh).

grains.med.**get_elements**(*mesh*, *numbering='global'*)
　Obtains the elements for each group of a mesh.

Elements of the same dimension as the mesh are collected (e.g. faces for a 2D mesh).

---

**Todo:** put those elements that do not belong to any group into an automatically created group

---

**Todo:** support ordering *elements* in alphabetical order

---

**Todo:** implement the *'global'* strategy

---

　**Parameters**

- **mesh** (*MEDFileUMesh*) – Unstructured mesh.

- **numbering** (*{'global'}, optional*) –

　**Determines how to allocate element numbers in the mesh.** 'global': numbers the elements without taking into account which group they belong to. Use this strategy if you are not sure whether an element belongs to more than one group. 'group': numbers the elements group-wise. This is much faster than the *'global'* strategy, but use this option if you are sure that the groups of the mesh do not contain common elements.

　The default is 'global'.

　**Returns**

- **elements** (*ndarray*) – Element-node connectivities in a 2D numpy array, in which each row corresponds to an element and the columns are the nodes of the elements. It is assumed that all the elements have the same number of nodes.

- **element_groups** (*dict*) – The keys in the dictionary are the element group names, while the values are list of integers, giving the elements that belong to the particular group.

> **Warning:** Currently, elements that do not fit into any groups are discarded.

**See also:**

`get_nodes()`, `change_node_numbering()`

## Notes

The element-node connectivities are read from the mesh. If you want to change the ordering of the nodes, use the `change_node_numbering()` function.

Both this and the `get_nodes()` function relies on getGroupsOnSpecifiedLev to obtain the groups based on a parameter, called *meshDimRelToMaxExt*. This parameter designates the relative dimension of the mesh entities whose IDs are required. If it is 1, it denotes the nodes. If 0, entities of the same dimension as the mesh are meant (e.g. group of volumes for a 3D mesh, or group of faces for a 2D mesh). When -1, entities of spatial dimension immediately below that of the mesh are collected (e.g. group of faces for a 3D mesh, or group of edges for a 2D mesh). For -2, entities of two dimensions below that of the mesh are fetched (e.g. group of edges for a 3D mesh).

`grains.med.`**`get_nodes`**(*mesh*)

Obtains the nodes and the node groups of a mesh.

> **Parameters mesh** (*MEDFileUMesh*) – Unstructured mesh.
>
> **Returns**
>
> > • **nodes** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a node, and the two columns giving the Cartesian coordinates of the nodes.
> >
> > • **node_groups** (*dict*) – The keys in the dictionary are the node group names, while the values are list of integers, giving the nodes that belong to the particular group.
>
> **See also:**
>
> `get_elements()`, getGroupArr

`grains.med.`**`read_mesh`**(*filename*)

Reads a mesh file in .med format. Only one mesh, the first one, is supported. However, that mesh can contain groups.

> **Parameters filename** (*str*) – Path to the mesh file.
>
> **Returns** *MEDFileUMesh* – Represents an unstructured mesh. For details, see the manual on https://docs.salome-platform.org/latest/dev/MEDCoupling/developer/classMEDCoupling_1_1MEDFileUMesh.html

# GEOMETRY

This module implements computational geometry algorithms, needed for other modules.

All the examples assume that the modules *numpy* and *matplotlib.pyplot* were imported as *np* and *plt*, respectively.

## 21.1 Classes

| | |
|---|---|
| *Mesh* | Data structure for a general mesh. |
| *TriMesh* | Unstructured triangular mesh. |
| *Polygon* | Represents a polygon. |

### 21.1.1 grains.geometry.Mesh

**class** grains.geometry.**Mesh**(*vertices*, *cells*)

Data structure for a general mesh.

This class by no means wants to provide intricate functionalities and does not strive to be efficient at all. Its purpose is to give some useful features the project relies on. The two main entities in the mesh are the *vertices* and the *cells*. They are expected to be passed by the user, so reading from various mesh files is not implemented. This keeps the class simple, requires few package dependencies and keeps the class focused as there are powerful tools to convert among mesh formats (see e.g. meshio).

> **Parameters**
>
> - **vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.
>
> - **cells** (*ndarray*) – Cell-vertex connectivities in a 2D numpy array, in which each row corresponds to a cell and the columns are the vertices of the cells. It is assumed that all the cells have the same number of vertices.

> **See also:**
>
> *change_vertex_numbering*

### Notes

Although not necessary, it is highly recommended that the local vertex numbering in the cells are the same, either clockwise or counter-clockwise. Some methods, such as *get_boundary()* even requires it. If you are not sure whether the cells you provide have a consistent numbering, it is better to renumber them by calling the *change_vertex_numbering()* method.

**__init__**(*vertices*, *cells*)

---

**Todo:** Error checking

---

### Methods

| | |
|---|---|
| *__init__*(vertices, cells) | |
| *associate_field*(vertex_values[, name]) | Associates a scalar, vector or tensor field to the nodes. |
| *create_cell_set*(name, cells) | Forms a group from a set of cells. |
| *create_vertex_set*(name, vertices) | Forms a group from a set of vertices. |
| *get_boundary*() | Extracts the boundary of the mesh. |
| *get_edges*() | Constructs edge-cell connectivities of the mesh. |

## 21.1.2 grains.geometry.TriMesh

**class** grains.geometry.**TriMesh**(*vertices*, *cells*)
Unstructured triangular mesh.

Vertices and cells are both stored as numpy arrays. This makes the simple mesh manipulations easy and provides interoperability with the whole scientific Python stack.

**Parameters**

- **vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.

- **cells** (*ndarray*) – Cell-vertex connectivities in a 2D numpy array, in which each row corresponds to a cell and the columns are the vertices of the cells. It is assumed that all the cells have the same number of vertices.

**__init__**(*vertices*, *cells*)
Error checking

**Methods**

| | |
|---|---|
| *__init__*(vertices, cells) | |
| associate_field(vertex_values[, name]) | Associates a scalar, vector or tensor field to the nodes. |
| *cell_area*(cell) | Computes the area of a cell. |
| *cell_set_area*(cell_set) | Computes the area of a cell set. |
| *cell_set_to_mesh*(cell_set) | Creates a mesh from a cell set. |
| *change_vertex_numbering*(orientation[, in-place]) | Changes cell vertex numbering. |
| create_cell_set(name, cells) | Forms a group from a set of cells. |
| create_vertex_set(name, vertices) | Forms a group from a set of vertices. |
| get_boundary() | Extracts the boundary of the mesh. |
| get_edges() | Constructs edge-cell connectivities of the mesh. |
| *plot*(*args, **kwargs) | Plots the mesh. |
| *plot_field*(component, *args[, show_mesh]) | Plots a field on the mesh. |
| *rotate*(angle[, point, inplace]) | Rotates a 2D mesh about a given point by a given angle. |
| *sample_mesh*(sample[, param]) | Provides sample meshes. |
| *scale*(factor[, inplace]) | Scales the geometry by modifying the coordinates of the vertices. |
| *write_inp*(filename) | Writes the mesh to an Abaqus .inp file. |

**Attributes**

| | |
|---|---|
| *plot_options* | |

### 21.1.3 grains.geometry.Polygon

**class** grains.geometry.**Polygon**(*vertices*)

Represents a polygon.

This class works as expected as long as the given polygon is *simple*, i.e. it is not self-intersecting and does not contain holes.

A simple class that has *numpy* and *matplotlib* (for the visualization) as the only dependencies. It does not want to provide extensive functionalities (for those, check out Shapely). The polygon is represented by its vertices, given in a consecutive order.

> **Parameters vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.

> **Raises**

> - **Exception** – If all the vertices of the polygon lie along the same line. If the polygon is not given in R^2.

> - **ValueError** – If the polygon does not have at least 3 vertices.

### Examples

Try to give a "polygon", in which all vertices are collinear

```
>>> poly = Polygon(np.array([[0, 0], [1, 1], [2, 2]]))
Traceback (most recent call last):
...
Exception: All vertices are collinear. Not a valid polygon.
```

Now we give a valid polygon:

```
>>> pentagon = Polygon(np.array([[2, 1], [0, 0], [0.5, 3], [-1, 4], [3, 5]]))
```

Use Python's *print* function to display basic information about a polygon:

```
>>> print(pentagon)
A non-convex polygon with 5 vertices, oriented clockwise.
```

**__init__**(*vertices*)
   Initialize self. See help(type(self)) for accurate signature.

### Methods

| | |
|---|---|
| *__init__*(vertices) | Initialize self. |
| *area*() | Signed area of the polygon. |
| *centroid*() | Centroid of the polygon. |
| *diameter*([definition]) | Diameter of the polygon. |
| *is_convex*() | Decides whether the polygon is convex. |
| *orientation*() | Orientation of the polygon. |
| *plot*(*args, **kwargs) | Plots the polygon. |

### Attributes

| | |
|---|---|
| *plot_options* | |

## 21.2 Functions

| | |
|---|---|
| *is_collinear*(points[, tol]) | Decides whether a set of points is collinear. |
| *squared_distance*(x, y) | Squared Euclidean distance between two points. |
| *distance_matrix*(points) | A symmetric square matrix, containing the pairwise squared Euclidean distances among points. |
| *_polygon_area*(x, y) | Computes the signed area of a non-self-intersecting, possibly concave, polygon. |

## 21.2.1 grains.geometry.is_collinear

grains.geometry.**is_collinear**(*points*, *tol=None*)

    Decides whether a set of points is collinear.

    Works in any dimensions.

> **Parameters**
>
> > - **points** (*ndarray*) – 2D numpy array with N columns, each row corresponding to a point, and the N columns giving the Cartesian coordinates of the point.
> >
> > - **tol** (*float, optional*) – Tolerance value passed to numpy's *matrix_rank* function. This tolerance gives the threshold below which SVD values are considered zero.
>
> **Returns** *bool* – True for collinear points.

See also:

numpy.linalg.matrix_rank()

### Notes

The algorithm for three points is from Tim Davis.

### Examples

Two points are always collinear

```
>>> is_collinear(np.array([[1, 0], [1, 5]]))
True
```

Three points in 3D which are supposed to be collinear (returns false due to numerical error)

```
>>> is_collinear(np.array([[0, 0, 0], [1, 1, 1], [5, 5, 5]]), tol=0)
False
```

The previous example with looser tolerance

```
>>> is_collinear(np.array([[0, 0, 0], [1, 1, 1], [5, 5, 5]]), tol=1e-14)
True
```

## 21.2.2 grains.geometry.squared_distance

grains.geometry.**squared_distance**(*x*, *y*)

    Squared Euclidean distance between two points.

    For points $x(x_1, ..., x_n)$ and $y(y_1, ...y_n)$ the following metric is computed

$$\sum_{i=1}^{n}(x_i - y_i)^2$$

> **Parameters** **x, y** (*ndarray*) – 1D numpy array, containing the coordinates of the two points.
>
> **Returns** *float* – Squared Euclidean distance.

See also:

*distance_matrix()*

### Examples

```
>>> squared_distance(np.array([0, 0, 0]), np.array([1, 1, 1]))
3.0
```

## 21.2.3 grains.geometry.distance_matrix

grains.geometry.**distance_matrix**(*points*)

A symmetric square matrix, containing the pairwise squared Euclidean distances among points.

> **Parameters** **points** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a point,
> and the two columns giving the Cartesian coordinates of the points.

> **Returns** **dm** (*ndarray*) – Distance matrix.

**See also:**

*squared_distance()*

### Examples

```
>>> points = np.array([[1, 1], [3, 0], [-1, -1]])
>>> distance_matrix(points)
array([[ 0.,   5.,   8.],
       [ 5.,   0.,  17.],
       [ 8.,  17.,   0.]])
```

## 21.2.4 grains.geometry._polygon_area

grains.geometry.**_polygon_area**(*x*, *y*)

Computes the signed area of a non-self-intersecting, possibly concave, polygon.

Directly taken from http://rosettacode.org/wiki/Shoelace_formula_for_polygonal_area#Python

> **Parameters** **x, y** (*list*) – Coordinates of the consecutive vertices of the polygon.

> **Returns** *float* – Area of the polygon.

> **Warning:** If numpy vectors are passed as inputs, the resulting area is incorrect! WHY?

### Notes

The code is not optimized for speed and for numerical stability. Intended to be used to compute the area of finite element cells, in which case the numerical stability is not an issue (unless the cell is degenerate). As this function is called possibly as many times as the number of cells in the mesh, no input checking is performed.

**Examples**

```
>>> _polygon_area([0, 1, 1], [0, 0, 1])
0.5
```

**class** grains.geometry.**Mesh**(*vertices*, *cells*)

Bases: abc.ABC

Data structure for a general mesh.

This class by no means wants to provide intricate functionalities and does not strive to be efficient at all. Its purpose is to give some useful features the project relies on. The two main entities in the mesh are the *vertices* and the *cells*. They are expected to be passed by the user, so reading from various mesh files is not implemented. This keeps the class simple, requires few package dependencies and keeps the class focused as there are powerful tools to convert among mesh formats (see e.g. meshio).

**Parameters**

- **vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.

- **cells** (*ndarray*) – Cell-vertex connectivities in a 2D numpy array, in which each row corresponds to a cell and the columns are the vertices of the cells. It is assumed that all the cells have the same number of vertices.

**See also:**

*change_vertex_numbering*

**Notes**

Although not necessary, it is highly recommended that the local vertex numbering in the cells are the same, either clockwise or counter-clockwise. Some methods, such as *get_boundary()* even requires it. If you are not sure whether the cells you provide have a consistent numbering, it is better to renumber them by calling the *change_vertex_numbering()* method.

**static _ismatrix**(*array*)

Decides whether the input is a matrix.

**Parameters array** (*ndarray*) – Numpy array to be checked.

**Returns** *bool* – True if the input is a 2D array. Otherwise, False.

**See also:**

*_isvector()*

**static _isvector**(*array*)

Decides whether the input is a vector.

**Parameters array** (*ndarray*) – Numpy array to be checked.

**Returns** *bool* – True if the input is a 1D array or if it is a column or row vector. Otherwise, False.

**See also:**

*_ismatrix()*

**associate_field**(*vertex_values*, *name='field'*)

Associates a scalar, vector or tensor field to the nodes.

Only one field can be present at a time. If you want to use a new field, call this method again with the new field values, which will replace the previous ones.

> **Parameters**
>
> > - **vertex_values** (*ndarray*) – Field values at the nodes.
> >
> > - **name** (*str, optional*) – Name of the field. If not given, it will be 'field'.
>
> **Returns** *None*

**create_cell_set** (*name*, *cells*)
> Forms a group from a set of cells.
>
> > **Parameters**
> >
> > > - **name** (*str*) – Name of the cell set.
> > >
> > > - **cells** (*list*) – List of cells to be added to the set.
> >
> > **Returns** *None*

**create_vertex_set** (*name*, *vertices*)
> Forms a group from a set of vertices.
>
> > **Parameters**
> >
> > > - **name** (*str*) – Name of the vertex set.
> > >
> > > - **vertices** (*list*) – List of vertices to be added to the set.

**get_boundary** ()
> Extracts the boundary of the mesh.
>
> It is expected that all the cells have the same orientation, i.e. the cell vertices are consistently numbered (either clockwise or counter-clockwise). See the constructor for details.
>
> > **Returns**
> >
> > > - **boundary_vertices** (*ndarray*) – Ordered 1D ndarray of vertices, the boundary vertices of the mesh.
> > >
> > > - **boundary_edges** (*dict*) – The keys of the returned dictionary are 2-tuples, representing the two vertices of the boundary edges, while the values are the list of cells containing a particular boundary edge. The dictionary is ordered: the consecutive keys represent the consecutive boundary edges. Although a boundary edge is part of a single cell, that cell is given in a list so as to maintain the same format as the one used in the `get_edges()` method.

### Notes

The reason why consistent cell vertex numbering is demanded is because in that case the boundary edges are oriented in such a way that the second vertex of a boundary edge is the first vertex of the boundary edge it connects to.

### Examples

Let us consider the same example mesh as the one described in the `get_edges()` method.

```
>>> mesh = TriMesh(np.array([[0, 0], [1, 0], [2, 0], [0, 2], [0, 1], [1, 1]]),
...                np.array([[0, 1, 5], [4, 5, 3], [5, 4, 0], [2, 5, 1]]))
```

We extract the boundary of that mesh using

```
>>> bnd_vertices, bnd_edges = mesh.get_boundary()
>>> bnd_vertices
array([0, 1, 2, 5, 3, 4])
>>> bnd_edges
{(0, 1): [0], (1, 2): [3], (2, 5): [3], (5, 3): [1], (3, 4): [1], (4, 0): [2]}
```

**get_edges**()

Constructs edge-cell connectivities of the mesh.

The cells of the mesh do not necessarily have to have a consistent vertex numbering.

> **Returns edges** (*dict*) – The keys of the returned dictionary are 2-tuples, representing the two vertices of the edges, while the values are the list of cells containing a particular edge.

### Notes

We traverse through the cells of the mesh, and within each cell the edges. The edges are stored as new entries in a dictionary if they are not already stored. Checking if a key exists in a dictionary is performed in O(1). The number of edges in a cell is independent of the mesh density. Therefore, the time complexity of the algorithm is O(N), where N is the number of cells in the mesh.

**See also:**

*get_boundary()*

### Examples

We show an example for a triangular mesh (as the *Mesh* class is abstract).

```
>>> mesh = TriMesh(np.array([[0, 0], [1, 0], [2, 0], [0, 2], [0, 1], [1, 1]]),
...                np.array([[0, 1, 5], [4, 5, 3], [5, 4, 0], [2, 5, 1]]))
>>> edges = mesh.get_edges()
>>> edges
{(0, 1): [0], (1, 5): [0, 3], (5, 0): [0, 2], (4, 5): [1, 2], (5, 3): [1],
(3, 4): [1], (4, 0): [2], (2, 5): [3], (1, 2): [3]}
>>> mesh.plot(cell_labels=True, vertex_labels=True)
>>> plt.show()
```

**class** grains.geometry.**Polygon**(*vertices*)

Bases: object

Represents a polygon.

This class works as expected as long as the given polygon is *simple*, i.e. it is not self-intersecting and does not contain holes.

A simple class that has *numpy* and *matplotlib* (for the visualization) as the only dependencies. It does not want to provide extensive functionalities (for those, check out Shapely). The polygon is represented by its vertices, given in a consecutive order.

> **Parameters vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.

> **Raises**

> - **Exception** – If all the vertices of the polygon lie along the same line. If the polygon is not given in R^2.
> - **ValueError** – If the polygon does not have at least 3 vertices.

### Examples

Try to give a "polygon", in which all vertices are collinear

```
>>> poly = Polygon(np.array([[0, 0], [1, 1], [2, 2]]))
Traceback (most recent call last):
...
Exception: All vertices are collinear. Not a valid polygon.
```

Now we give a valid polygon:

```
>>> pentagon = Polygon(np.array([[2, 1], [0, 0], [0.5, 3], [-1, 4], [3, 5]]))
```

Use Python's *print* function to display basic information about a polygon:

```
>>> print(pentagon)
A non-convex polygon with 5 vertices, oriented clockwise.
```

**area()**
> Signed area of the polygon.

> The signed area is computed by the shoelace formula[1]

$$A = \frac{1}{2} \sum_{i=1}^{N} (x_i y_{i+1} - x_{i+1} y_i)$$

> **Returns** *float* – Signed area.

### References

### Examples

```
>>> poly = Polygon(np.array([[0, 0], [1, 0], [1, 1], [-1, 1]]))
>>> poly.area()
1.5
>>> poly = Polygon(np.array([[-1, 1], [1, 1], [1, 0], [0, 0]]))
>>> poly.area()
-1.5
```

**centroid()**
> Centroid of the polygon.

---

[1] http://paulbourke.net/geometry/polygonmesh/centroid.pdf

The centroid is computed according to the following formula[2]

$$C_x = \frac{1}{6A} \sum_{i=1}^{N} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A} \sum_{i=1}^{N} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

where $A$ is the signed area determined by the `area()` method and $x_i, y_i$ are the vertex coordinates with $x_{N+1} = x_1$ and $y_{N+1} = y_1$.

> **Returns** *tuple* – 2-tuple, the coordinates of the centroid.

### References

### Examples

```
>>> poly = Polygon(np.array([[0, 0], [0, 1], [1, 1], [1, 0]]))
>>> poly.centroid()
(0.5, 0.5)
>>> poly = Polygon(np.array([[2, 1], [0, 0], [0.5, 3], [-1, 4], [3, 5]]))
>>> poly.centroid()
(1.254..., 2.807...)
```

**diameter**(*definition='set'*)

> Diameter of the polygon.
>
> Multiple definitions are supported for the diameter:
>
> - *Diameter of a set*. The polygon is considered as a set $A$ of points comprised
>
> of the polygon vertices. Let $(X, d)$ be a metric space. The diameter of the set is defined as
>
> $$\mathrm{diam}(A) = \sup\{d(x, y) \mid x, y \in A\}. \tag{21.1}$$
>
> Here, the Euclidean metric is used.
>
> - *Equivalent diameter*. Diameter of the circle of the same area as that of the polygon.
>
> > **Parameters definition** (*{'set', 'equivalent'}, optional*) – The default is 'set'.
> >
> > **Returns** *float* – Diameter of the polygon, based on the chosen definition.

### Notes

When `definition` is 'set', computing the diameter by (21.1) is equivalent to determining the distance of the furthest points in the convex hull of $A$. Therefore, the diameter will always be the distance between two points on the convex hull of $A$. Then for each vertex of the hull finding which other hull vertex is farthest away from it, the *rotating caliper* algorithm can be used. Our brute-force method is simpler as it needs neither the convex hull nor the rotating caliper algorithm: all the pairwise distances among the polygon vertices are computed and the largest one is chosen. Pair of points a maximum distance apart. Since the polygons in our applications do not have that many vertices, this simplistic approach is a viable alternative.

---

[2] http://paulbourke.net/geometry/polygonmesh/centroid.pdf

### Examples

```
>>> poly = Polygon(np.array([[2, 5], [0, 1], [4, 3], [4, 5]]))
>>> poly.diameter('set')
5.6568542...
>>> poly.diameter('equivalent')
3.1915382...
>>> poly = Polygon(np.array([[2, 1], [3, -4], [-1, -1], [-4, -2], [-3, 0]]))
>>> poly.diameter('set')
7.2801098...
>>> poly.diameter('equivalent')
4.2967398...
```

**is_convex()**
Decides whether the polygon is convex.

> **Returns** *bool* – True if the polygon is convex. False otherwise.

### Notes

The algorithm works by checking if all pairs of consecutive edges in the polygon are either all clockwise or all counter-clockwise oriented. This method is valid only for simple polygons. The implementation follows this code, extended for the case when two consecutive edges are collinear. If the polygon was not simple, a more complicated algorithm would be needed, see e.g. here.

### Examples

A triangle is always convex:

```
>>> poly = Polygon(np.array([[1, 1], [0, 1], [0, 0]]))
>>> poly.is_convex()
True
```

Let us define a concave deltoid:

```
>>> poly = Polygon(np.array([[-1, -1], [0, 1], [1, -1], [0, 5]]))
>>> poly.is_convex()
False
```

Give a polygon that has two collinear edges:

```
>>> poly = Polygon(np.array([[0.5, 0], [1, 0], [1, 1], [0, 1], [0, 0]]))
>>> poly.is_convex()
True
```

**orientation()**
Orientation of the polygon.

> **Returns** *str* – 'cw' if the polygon has clockwise orientation, 'ccw' if counter-clockwise.

**plot**(*\*args*, *\*\*kwargs*)
Plots the polygon.

> **Parameters ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the polygon resides in. The default is None, in which case a new *Axes* within a new figure is created.
>
> **Other Parameters**

- **show_axes** (*bool, optional*) – If True, the coordinate system is shown. The default is True.

- **vertex_labels** (*bool, optional*) – If True, vertex labels are shown. The default is False.

- **args, kwargs** (*optional*) – Additional arguments and keyword arguments to be specified. Those arguments are the ones supported by `matplotlib.axes.Axes.plot()`.

**Returns** *None*

### Examples

Consider the pentagon used in the example of the constructor. Plot it in black with red diamond symbols representing its vertices. Moreover, display the vertex numbers and do not show the coordinate system.

```
>>> pentagon = Polygon(np.array([[2, 1], [0, 0], [0.5, 3], [-1, 4], [3, 5]]))
>>> pentagon.plot('k-d', vertex_labels=True, markerfacecolor='r', show_
↪axes=False)
>>> plt.show()
```

**plot_options = {'ax': None, 'show_axes': True, 'vertex_labels': False}**

**class** `grains.geometry.`**TriMesh**(*vertices*, *cells*)

Bases: `grains.geometry.Mesh`

Unstructured triangular mesh.

Vertices and cells are both stored as numpy arrays. This makes the simple mesh manipulations easy and provides interoperability with the whole scientific Python stack.

**Parameters**

- **vertices** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertex.

- **cells** (*ndarray*) – Cell-vertex connectivities in a 2D numpy array, in which each row corresponds to a cell and the columns are the vertices of the cells. It is assumed that all the cells have the same number of vertices.

**cell_area**(*cell*)

Computes the area of a cell.

**Parameters** **cell** (*int*) – Cell label.

**Returns** **area** (*float*) – Area of the cell.

See also:

*cell_set_area()*

**cell_set_area**(*cell_set*)

Computes the area of a cell set.

**Parameters** **cell_set** (*str*) – Name of the cell set.

**Returns** **area** (*float*) – Area of the cell set.

See also:

*cell_area()*

**cell_set_to_mesh**(*cell_set*)

Creates a mesh from a cell set.

The cell orientation is preserved. I.e. if the cells had a consistent orientation (clockwise or counter-clockwise), the cells of the new mesh inherit this property.

> **Parameters cell_set** (*str*) – Name of the cell set being used to construct the new mesh. The cell set must be present in the `cell_sets` member variable of the current mesh object.

> **Returns** *TriMesh* – A new *TriMesh* object, based on the selected cell set of the original mesh.

### Notes

The implementation is based on https://stackoverflow.com/a/13572640/4892892.

### Examples

```
>>> mesh = TriMesh(np.array([[0, 0], [1, 0], [0, 1], [1, 1]]),
...                np.array([[0, 1, 2], [1, 3, 2]]))
>>> mesh.create_cell_set('set', [1])
>>> new_mesh = mesh.cell_set_to_mesh('set')
>>> new_mesh.cells  # note that the vertices have been relabelled
array([[0, 2, 1]])
>>> new_mesh.vertices
array([[1, 0],
       [0, 1],
       [1, 1]])
>>> new_mesh.plot(cell_labels=True, vertex_labels=True)
>>> plt.show()
```

**change_vertex_numbering**(*orientation*, *inplace=False*)

Changes cell vertex numbering.

> **Parameters**
>
> - **orientation** (*{'ccw', 'cw'}*) – Vertex numbering within a cell, either *'ccw'* (counter-clockwise, default) or *'cw'* (clock-wise).
>
> - **inplace** (*bool, optional*) – If True, the vertex ordering is updated in the mesh object. The default is False.
>
> **Returns reordered_cells** (*ndarray*) – Same format as the `cells` member variable, with the requested vertex ordering.

### Notes

Supposed to be used with planar P1 or Q1 finite s.

### Examples

```
>>> mesh = TriMesh(np.array([[1, 1], [3, 5], [7,3]]), np.array([0, 1, 2]))
>>> mesh.change_vertex_numbering('ccw')
array([[2, 1, 0]])
```

**plot**(*\*args*, *\*\*kwargs*)

Plots the mesh.

> **Parameters ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the mesh resides in. The default is None, in which case a new *Axes* within a new figure is created.

> **Other Parameters**
> - **cell_sets, vertex_sets** (*bool, optional*) – If True, the cell/vertex sets (if exist) are highlighted in random colors. The default is True.
>
> - **cell_legends, vertex_legends** (*bool, optional*) – If True, cell/vertex set legends are shown. The default is False. For many sets, it is recommended to leave these options as False, otherwise the plotting becomes very slow.
>
> - **cell_labels, vertex_labels** (*bool, optional*) – If True, cell/vertex labels are shown. The default is False. Recommended to be left False in case of many cells/vertices. Cell labels are positioned in the centroids of the cells.
>
> - **args, kwargs** (*optional*) – Additional arguments and keyword arguments to be specified. Those arguments are the ones supported by `matplotlib.axes.Axes.plot()`.

> **Returns** *None*

### Notes

If you do not want to plot the cells, only the vertices, pass the `'.'` option, e.g.:

```
mesh.plot('k.')
```

to plot the vertices in black. Here, `mesh` is a *TriMesh* object.

### Examples

A sample mesh is constructed by creating uniformly randomly distributed points on the rectangular domain [-1, 1] x [1, 2]. These points will constitute the vertices of the mesh, while its cells are the Delaunay triangles on the vertices.

```
>>> from grains.geometry import TriMesh
>>> msh = TriMesh(*TriMesh.sample_mesh(1))
```

The cells are drawn in greeen, in 3 points of line width, and the vertices of the mesh are shown in blue.

```
>>> msh.plot('go-', linewidth=3, markerfacecolor='b', vertex_labels=True)
>>> plt.show()
```

### Notes

The plotting is done by calling `triplot()`, which internally makes a deep copy of the triangles. This increases the memory usage in case of many elements.

**plot_field**(*component*, *\*args*, *show_mesh=True*, *\*\*kwargs*)
> Plots a field on the mesh.

> The aim of this method is to support basic post-processing for finite element visualization. Only the basic contour plot type is available. For vector or tensor fields, the components to be plotted must be chosen. For faster and more comprehensive plotting capabilities, turn to well-established scientific visualization software, such as ParaView or Mayavi. Another limitation of the `plot_field()` method is that field values are assumed to be associated to the vertices of the mesh, which restricts us to $P1$ Lagrange elements.
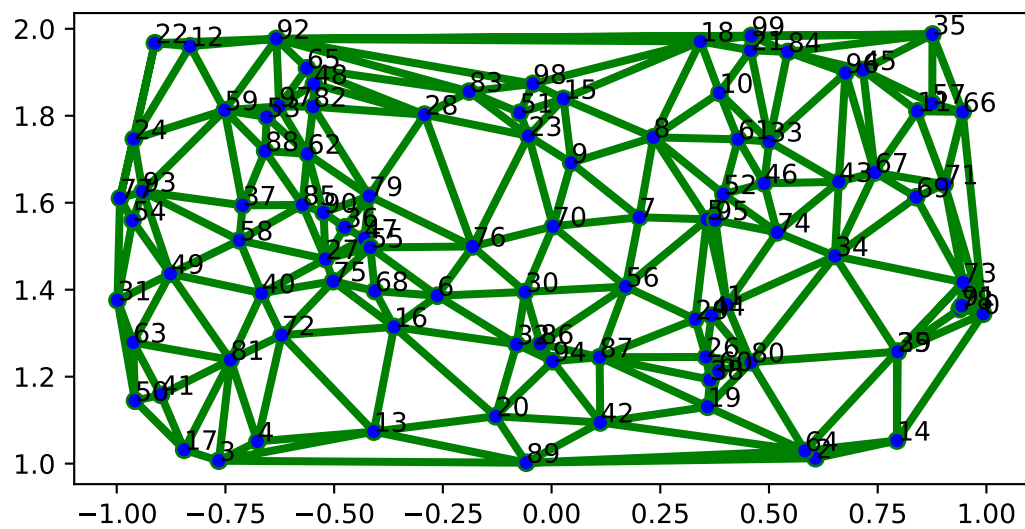
**Parameters**

- **component** (*int*) – Positive integer, the selected component of the field to be plotted. Components are indexed from 0.

- **show_mesh** (*bool, optional*) – If True, the underlying mesh is shown. The default is True.

- **ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the plot resides in. The default is None, in which case a new *Axes* within a new figure is created.

**Other Parameters** See them described in the :meth:`plot` method.

**Returns** *None*

See also:

*plot()*

## Examples

The following example considers the same type of mesh as in the example shown for *plot()*.

```
>>> msh = TriMesh(*TriMesh.sample_mesh(1))
```

We pretend that the field is an analytical function, evaluated at the vertices.

```
>>> field = lambda x, y: 1 - (x + y**2) * np.sign(x)
>>> field = field(msh.vertices[:, 0], msh.vertices[:, 1])
```

We associate this field to the mesh and plot it with and without the mesh

```
>>> msh.associate_field(field, 'analytical field')
>>> _, (ax1, ax2) = plt.subplots(1, 2)
>>> msh.plot_field(0, 'bo-', ax=ax1, linewidth=1, markerfacecolor='k')
>>> msh.plot_field(0, ax=ax2, show_mesh=False)
>>> plt.show()
```

**plot_options = {'ax': None, 'cell_labels': False, 'cell_legends': False, 'cell_sets': '**

**rotate** (*angle*, *point=(0, 0)*, *inplace=False*)
    Rotates a 2D mesh about a given point by a given angle.

**Parameters**

- **angle** (*float*) – Angle of rotation, in radians.

- **point** (*list or tuple, optional*) – Coordinates of the point about which the mesh is rotated. If not given, it is the origin.

- **inplace** (*bool, optional*) – If True, the vertex positions are updated in the mesh object. The default is False.

**Returns** **rotated_vertices** (*ndarray*) – Same format as the `vertices` member variable, with the requested rotation.

**Notes**

Rotating a point $P$ given by its coordinates in the global coordinate system as $P(x, y)$ around a point $A(x, y)$ by an angle $\alpha$ is done as follows.

1. The coordinates of $P$ in the local coordinate system, the origin of which is $A$, is expressed as

$$P(x', y') = P(x, y) - A(x, y).$$

2. The rotation is performed in the local coordinate system as $P'(x', y') = RP(x', y')$, where $R$ is the rotation matrix:

$$R = \begin{pmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{pmatrix}.$$

3. The rotated point $P'$ is expressed in the original (global) coordinate system:

$$P'(x, y) = P'(x', y') + A(x, y).$$

**static sample_mesh**(*sample*, *param=100*)
  Provides sample meshes.

  **Parameters**

  - **sample** (*int*) – Integer, giving the sample mesh to be considered. Possibilities:

  - **param** – Parameters to the sample meshes. Possibilities:

  **Returns**

  - **nodes** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a vertex, and the two columns giving the Cartesian coordinates of the vertices.

  - **cells** (*ndarray*) – Cell-vertex connectivity in a 2D numpy array, in which each row corresponds to a cell and the columns are the vertices of the cells. It is assumed that all the cells have the same number of vertices.

**scale**(*factor*, *inplace=False*)
  Scales the geometry by modifying the coordinates of the vertices.

  **Parameters**

  - **factor** (*float*) – Each vertex coordinate is multiplied by this non-negative number.

  - **inplace** (*bool, optional*) – If True, the vertex positions are updated in the mesh object. The default is False.

  **Returns** *None.*

**write_inp**(*filename*)
  Writes the mesh to an Abaqus .inp file.

  **Parameters** **filename** (*str*) – Path to the mesh file.

  **Returns** *None*

grains.geometry.**_polygon_area**(*x*, *y*)
  Computes the signed area of a non-self-intersecting, possibly concave, polygon.

  Directly taken from http://rosettacode.org/wiki/Shoelace_formula_for_polygonal_area#Python

  **Parameters** **x, y** (*list*) – Coordinates of the consecutive vertices of the polygon.

  **Returns** *float* – Area of the polygon.

> **Warning:** If numpy vectors are passed as inputs, the resulting area is incorrect! WHY?

### Notes

The code is not optimized for speed and for numerical stability. Intended to be used to compute the area of finite element cells, in which case the numerical stability is not an issue (unless the cell is degenerate). As this function is called possibly as many times as the number of cells in the mesh, no input checking is performed.

### Examples

```
>>> _polygon_area([0, 1, 1], [0, 0, 1])
0.5
```

grains.geometry.**distance_matrix**(*points*)

A symmetric square matrix, containing the pairwise squared Euclidean distances among points.

> **Parameters points** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a point, and the two columns giving the Cartesian coordinates of the points.

> **Returns dm** (*ndarray*) – Distance matrix.

See also:

*squared_distance()*

### Examples

```
>>> points = np.array([[1, 1], [3, 0], [-1, -1]])
>>> distance_matrix(points)
array([[ 0.,  5.,  8.],
       [ 5.,  0., 17.],
       [ 8., 17.,  0.]])
```

grains.geometry.**is_collinear**(*points*, *tol=None*)

Decides whether a set of points is collinear.

Works in any dimensions.

> **Parameters**
>
> - **points** (*ndarray*) – 2D numpy array with N columns, each row corresponding to a point, and the N columns giving the Cartesian coordinates of the point.
>
> - **tol** (*float, optional*) – Tolerance value passed to numpy's *matrix_rank* function. This tolerance gives the threshold below which SVD values are considered zero.

> **Returns** *bool* – True for collinear points.

See also:

numpy.linalg.matrix_rank()

## Notes

The algorithm for three points is from Tim Davis.

## Examples

Two points are always collinear

```
>>> is_collinear(np.array([[1, 0], [1, 5]]))
True
```

Three points in 3D which are supposed to be collinear (returns false due to numerical error)

```
>>> is_collinear(np.array([[0, 0, 0], [1, 1, 1], [5, 5, 5]]), tol=0)
False
```

The previous example with looser tolerance

```
>>> is_collinear(np.array([[0, 0, 0], [1, 1, 1], [5, 5, 5]]), tol=1e-14)
True
```

grains.geometry.**squared_distance**(*x*, *y*)

Squared Euclidean distance between two points.

For points $x(x_1, ..., x_n)$ and $y(y_1, ...y_n)$ the following metric is computed

$$\sum_{i=1}^{n}(x_i - y_i)^2$$

**Parameters**  **x, y** (*ndarray*) – 1D numpy array, containing the coordinates of the two points.

**Returns**  *float* – Squared Euclidean distance.

**See also:**

*distance_matrix()*

## Examples

```
>>> squared_distance(np.array([0, 0, 0]), np.array([1, 1, 1]))
3.0
```

# ABAQUS

> **Warning:** This module will substantially be rewritten. See this issue.

This module allows to create and manipulate Abaqus input files through the Abaqus keywords, thereby providing automation. Note that it is not intended to be a complete API to Abaqus. If you want fine control over the whole Abaqus ecosystem, consult with the Abaqus Scripting Reference Guide (*ASRG*). However, *ASRG* needs Abaqus to be installed, moreover, you must use the Python interpreter embedded into Abaqus. That version of Python is very old even in the latest versions of Abaqus. Furthermore, if you need to use custom Python packages for your work, chances are high that they will not work with the embedded interpreter, and may even crash the installation. To use this module, no Abaqus installation is needed. In fact, only functions from the Python standard library are used.

The documentation of Abaqus version 2017 is hosted on the following website: https://abaqus-docs.mit.edu/2017/ English/SIMACAEEXCRefMap/simaexc-c-docproc.htm. Throughout the documentation of this module (`grains. abaqus`), we will make references to that website. If the links cease to exist, please let me know by opening an issue. Alternatively, once you have registered, you can browse the documentation on the official website.

## 22.1 Classes

| | |
|---|---|
| *Geometry* | Geometrical operations on the mesh. |
| *Material* | Adds, removes, modifies materials. |
| *Procedure* | Handling analysis steps during the simulation. |

### 22.1.1 grains.abaqus.Geometry

**class** grains.abaqus.**Geometry**
    Geometrical operations on the mesh.

    **__init__**()
        Initialize self. See help(type(self)) for accurate signature.

**Methods**

| | |
|---|---|
| *__init__*() | Initialize self. |
| *read*(inp_file) | Reads material data from an Abaqus .inp file. |
| *scale*(factor) | Scales the geometry by modifying the coordinates of the nodes. |
| *write*([output_file]) | Writes material data to an Abaqus .inp file. |

## 22.1.2 grains.abaqus.Material

**class** grains.abaqus.**Material**(*from_Abaqus=False*)

Adds, removes, modifies materials. Requirements: be able to - create an empty .inp file, containing only the materials - add materials to an existing .inp file

TODO: only document public attributes! .. attribute:: materials

materials, their behaviors and their parameters are stored here. Intended for internal representation. To view them, use the *show* method, to write them to an input file, use the *write* method.

**type** dict

**read**(*inp_file*)

Reads material data from an Abaqus .inp file.

**write**(*output_file=None*)

Writes material data to an Abaqus .inp file.

**remove**(*inp_file*, *output_file=None*)

Removes material definitions from an Abaqus .inp file.

**create**(*inp_file*)

Creates empty Abaqus .inp file.

**show**()

Shows material data as it appears in the Abaqus .inp file.

**Notes**

The aim of this class is to programmatically add, remove, modify materials in a format understandable by Abaqus. This class does not target editing materials through a GUI (that can be done in Abaqus CAE).

**__init__**(*from_Abaqus=False*)

**Parameters from_Abaqus** (*bool, optional*) – True if the input file was generated by Abaqus. The default is False. Abaqus generates input files with a consistent format. This allows certain optimizations: the input file may not need to be completely traversed to extract the materials. Third-party programs sometimes generate .inp files, which are valid but do not follow the Abaqus pattern. In this case, it cannot be predicted where the material definition ends in the file – the whole file needs to be traversed.

**Returns** *None.*

**Methods**

| | |
|---|---|
| *__init__*([from_Abaqus]) | |
| | **Parameters from_Abaqus** (*bool, optional*) – True if the input file was generated by Abaqus. The default is False. |

| | |
|---|---|
| *add_linearelastic*(material, E, nu) | Adds linear elastic behavior to a given material. |
| *add_material*(material) | Defines a new material by its name. |
| *add_plastic*(material, sigma_y, epsilon_p) | Adds metal plasticity behavior to a given material. |
| *add_sections*(inp_file[, output_file]) | Adds section definitions to an Abaqus .inp file. |
| *create*(inp_file) | Creates empty Abaqus .inp file. |
| *read*(inp_file) | Reads material data from an Abaqus .inp file. |
| *remove*(inp_file[, output_file]) | Removes material definitions from an Abaqus .inp file. |
| *show*() | Shows material data as it appears in the Abaqus .inp file. |
| *write*([output_file]) | Writes material data to an Abaqus .inp file. |

## 22.1.3 grains.abaqus.Procedure

**class** grains.abaqus.**Procedure**(*from_Abaqus=False*)

Handling analysis steps during the simulation.

TODO: only document public attributes!

**steps**

analysis steps, each step collecting the necessary information. Intended for internal representation. To view them, use the *show()* method, to write them to an input file, use the *write()* method.

> **Type** dict

**__init__** (*from_Abaqus=False*)

> **Parameters from_Abaqus** (*bool, optional*) – True if the input file was generated by Abaqus. The default is False. Abaqus generates input files with a consistent format. This allows certain optimizations: the input file may not need to be completely traversed to extract the materials. Third-party programs sometimes generate .inp files, which are valid but do not follow the Abaqus pattern. In this case, it cannot be predicted where the material definition ends in the file – the whole file needs to be traversed.

> **Returns** *None.*

**Methods**

| | |
|---|---|
| *__init__*([from_Abaqus]) | |
| | **Parameters from_Abaqus** (*bool, optional*) – True if the input file was generated by Abaqus. The default is False. |

Table 4 – continued from previous page

| | |
|---|---|
| *add_analysis*(step[, time_period, …]) | Adds an analysis type to a given step. |
| *add_boundary_condition*(name, step, nodes, …) | Adds boundary condition to a given step. |
| *create_step*(name[, nlgeom, max_increments]) | Defines a new step. |
| *read*(inp_file) | Reads procedure data from an Abaqus .inp file. |
| *show*() | Shows the text for the *step* module, as it appears in the Abaqus .inp file. |
| *write*([output_file]) | Writes step definition data to an Abaqus .inp file. |

## 22.2 Functions

| | |
|---|---|
| *extract*(keyword) | Obtains Abaqus keyword and its parameters. |
| *validate_file*(file, caller) | Input or output file validation. |

### 22.2.1 grains.abaqus.extract

grains.abaqus.**extract**(*keyword*)

Obtains Abaqus keyword and its parameters.

> **Parameters keyword** (*str*) –
>
> > **Some examples:** '*Elastic, type=ORTHOTROPIC' '*Damage Initiation, criterion=HASHIN' '*Nset, nset=Set-1, generate'
>
> **Returns separated** (*list*) – DESCRIPTION.

### 22.2.2 grains.abaqus.validate_file

grains.abaqus.**validate_file**(*file*, *caller*)

Input or output file validation.

> **Parameters**
>
> > - **file** (*str*) – Existing Abaqus .inp file or a new .inp file to be created.
> > - **caller** (*{'read', 'write', 'create'}*) – Method name that called this function.
>
> **Returns** *None.*

**class** grains.abaqus.**Geometry**

> Bases: object
>
> Geometrical operations on the mesh.
>
> **_Geometry__format**()
>
> > **Formats the material data in the Abaqus .inp format.** The internal representation of the material data in converted to a string understood by Abaqus.
> >
> > **abaqus_format** [list] List of strings, each element of the list corresponding to a line (with
> >
> > **line ending) in the Abaqus .inp file. In case of no**
> >
> > > material, an empty list is returned.

The output is a list so that further concatenation operations are easy. If you want a string, merge the elements of the list:

output = ''.join(output)

This is what the *show* method does.

**read**(*inp_file*)
Reads material data from an Abaqus .inp file.

> **Parameters inp_file** (*str*) – Abaqus input (.inp) file to be created.
>
> **Returns** *None.*

### Notes

- This method is designed to read material data. Although the logic could be used to process other properties (parts, assemblies, etc.) in an input file, they are not yet implemented in this class.

- This method assumes that the input file is valid. If it is, the material data can be extacted. If not, the behavior is undefined: the program can crash or return garbage. This is by design: the single responsibility principle dictates that the validity of the input file must be provided by other methods. If the input file was generated from within Abaqus CAE, it is guaranteed to be valid. The *write* method of this class also ensures that the resulting input file is valid. This design choice also makes the program logic simpler. For valid syntax in the input file, check the Input Syntax Rules section in the Abaqus user's guide.

- To read material data from an input file, one has to identify the structure of .inp files in Abaqus. Abaqus is driven by keywords and corresponding data. For a list of accepted keywords, consult the Abaqus Keywords Reference Guide. There are three types of input lines in Abaqus:

  - keyword line: begins with a star, followed by the name of the keyword. Parameters, if any, are separated by commas and are given as parameter-value pairs. Keywords and parameters are not case sensitive. Example:

    *ELASTIC, TYPE=ISOTROPIC, DEPENDENCIES=1

    Some keywords can only be defined once another keyword has already been defined. E.g. the keyword ELASTIC must come after MATERIAL in a valid .inp file.

  - data line: immediately follows a keyword line. All data items must be separated by commas. Example:

    -12.345, 0.01, 5.2E-2, -1.2345E1

  - **comment line: starts with \*\* and is ignored by Abaqus. Example:** \*\* This is a comment line

- Internally, the materials are stored in a dictionary. It holds the material data read from the file. The keys in this dictionary are the names of the materials, and the values are dictionaries themselves. Each such dictionary stores a behavior for the given material. E.g. an elastoplastic material is governed by an elastic and a plastic behavior. The parameters for each behavior are stored in a list.

**scale**(*factor*)
Scales the geometry by modifying the coordinates of the nodes.

> **Parameters factor** (*float*) – Each nodal coordinate is multiplied by this non-negative number.
>
> **Returns** *None.*

### Notes

The modification happens in-place.

**write**(*output_file=None*)
: Writes material data to an Abaqus .inp file.

> **Parameters output_file** (*str, optional*) – Output file name to write the modifications into. If not given, the original file name is appended with '_mod'.

> **Returns** *None.*

### Notes

- If the output file name is the same as the input file, the original .inp file will be overwritten. This is strongly not recommended.

- The whole content of the original input file is read to memory. It might be a problem for very large .inp files. In that case, a possible implementation could be the following:

  1. Remove old material data

  2. Append new material data to the proper position in the file

  Appending is automatically done at the end of the file. Moving the material description to the end of the file is not possible in general because defining materials cannot be done from any module, i.e. the *****MATERIAL keyword cannot follow an arbitrary keyword. In this case, Abaqus throws an AbaqusException with the following message:

  > It can be suboption for the following keyword(s)/level(s): model

**class** grains.abaqus.**Material**(*from_Abaqus=False*)
: Bases: `object`

Adds, removes, modifies materials. Requirements: be able to - create an empty .inp file, containing only the materials - add materials to an existing .inp file

TODO: only document public attributes! .. attribute:: materials

> materials, their behaviors and their parameters are stored here. Intended for internal representation. To view them, use the *show* method, to write them to an input file, use the *write* method.
>
> > **type** dict

**read**(*inp_file*)
: Reads material data from an Abaqus .inp file.

**write**(*output_file=None*)
: Writes material data to an Abaqus .inp file.

**remove**(*inp_file*, *output_file=None*)
: Removes material definitions from an Abaqus .inp file.

**create**(*inp_file*)
: Creates empty Abaqus .inp file.

**show**()
: Shows material data as it appears in the Abaqus .inp file.

### Notes

The aim of this class is to programmatically add, remove, modify materials in a format understandable by Abaqus. This class does not target editing materials through a GUI (that can be done in Abaqus CAE).

**`_Material__format`**`()`

> **Formats the material data in the Abaqus .inp format.** The internal representation of the material data in converted to a string understood by Abaqus.
>
> **abaqus_format** [list] List of strings, each element of the list corresponding to a line (with
>
> **line ending) in the Abaqus .inp file. In case of no**
>
>> material, an empty list is returned.
>
> The output is a list so that further concatenation operations are easy. If you want a string, merge the elements of the list:
>
>> output = ''.join(output)
>
> This is what the *show* method does.

**`static _Material__isnumeric`**`(`*x*`)`

> Decides if the input is a scalar number.
>
>> **Parameters** **x** (*any type*) – Input to be tested.
>>
>> **Returns** *bool* – True if the given object is a scalar number.

**`add_linearelastic`**`(`*material*`, `*E*`, `*nu*`)`

> Adds linear elastic behavior to a given material.
>
>> **Parameters**
>>
>> - **material** (*str*) – Name of the material the behavior belongs to.
>> - **E** (*int, float*) – Young's modulus.
>> - **nu** (*int, float*) – Poisson's ratio.
>>
>> **Returns** *None.*

**`add_material`**`(`*material*`)`

> Defines a new material by its name.
>
>> **Parameters** **material** (*str*) – Name of the material to be added. A material can have multiple behaviors (e.g. elastoplastic).
>>
>> **Returns** *None.*

**`add_plastic`**`(`*material*`, `*sigma_y*`, `*epsilon_p*`)`

> Adds metal plasticity behavior to a given material.
>
>> **Parameters**
>>
>> - **material** (*str*) – Name of the material the behavior belongs to.
>> - **sigma_y** (*int, float*) – Yield stress.
>> - **epsilon_p** (*int, float*) – Plastic strain.
>>
>> **Returns** *None.*

**`static add_sections`**`(`*inp_file*`, `*output_file=None*`)`

> Adds section definitions to an Abaqus .inp file.

Defines properties for elements by associating materials to them. The element set containing the elements for which the material behavior is being defined is assumed to have the same name as that of the material. E.g. if materials with names *mat-1* and *mat-2* exist, element sets with names *mat-1* and *mat-2* must also exist. If such element sets do not exist, Abaqus will throw a warning and the section assignment will not be successful.

> **Parameters**
>
> > - **inp_file** (*str*) – Abaqus .inp file from which the materials should be removed.
> >
> > - **output_file** (*str, optional*) – Output file name to write the modifications into. If not given, the original file name is appended with '_mod'.
>
> **Returns** *None.*

### Notes

If fine control is required for associating custom material names to custom element set names, that can be done from the Abaqus GUI. The purpose of this method is automation for large number of element sets, each associated to a (possibly distinct) material. In that case, custom element set names are not reasonable any more, and having the same names for the element sets and for the materials is completely meaningful.

**create**(*inp_file*)

Creates empty Abaqus .inp file.

> **Parameters inp_file** (*str*) – Abaqus input file to be created. If an extension is not given, the default .inp is used.
>
> **Returns** *None.*

**read**(*inp_file*)

Reads material data from an Abaqus .inp file.

> **Parameters inp_file** (*str*) – Abaqus input (.inp) file to be created.
>
> **Returns** *None.*

### Notes

- This method is designed to read material data. Although the logic could be used to process other properties (parts, assemblies, etc.) in an input file, they are not yet implemented in this class.

- This method assumes that the input file is valid. If it is, the material data can be extacted. If not, the behavior is undefined: the program can crash or return garbage. This is by design: the single responsibility principle dictates that the validity of the input file must be provided by other methods. If the input file was generated from within Abaqus CAE, it is guaranteed to be valid. The *write* method of this class also ensures that the resulting input file is valid. This design choice also makes the program logic simpler. For valid syntax in the input file, check the Input Syntax Rules section in the Abaqus user's guide.

- To read material data from an input file, one has to identify the structure of .inp files in Abaqus. Abaqus is driven by keywords and corresponding data. For a list of accepted keywords, consult the Abaqus Keywords Reference Guide. There are three types of input lines in Abaqus:

  - keyword line: begins with a star, followed by the name of the keyword. Parameters, if any, are separated by commas and are given as parameter-value pairs. Keywords and parameters are not case sensitive. Example:

    *ELASTIC, TYPE=ISOTROPIC, DEPENDENCIES=1

Some keywords can only be defined once another keyword has already been defined. E.g. the keyword ELASTIC must come after MATERIAL in a valid .inp file.

- – data line: immediately follows a keyword line. All data items must be separated by commas. Example:

  -12.345, 0.01, 5.2E-2, -1.2345E1

- – **comment line: starts with ** and is ignored by Abaqus. Example:** ** This is a comment line

- Internally, the materials are stored in a dictionary. It holds the material data read from the file. The keys in this dictionary are the names of the materials, and the values are dictionaries themselves. Each such dictionary stores a behavior for the given material. E.g. an elastoplastic material is governed by an elastic and a plastic behavior. The parameters for each behavior are stored in a list.

**static remove**(*inp_file*, *output_file=None*)
> Removes material definitions from an Abaqus .inp file.

> **Parameters**
> > - **inp_file** (*str*) – Abaqus .inp file from which the materials should be removed.
> > - **output_file** (*str, optional*) – Output file name to write the modifications into. If not given, the original file name is appended with '_mod'.

> **Returns** *None.*

**show**()
> Shows material data as it appears in the Abaqus .inp file.

> **Returns** *None.*

**write**(*output_file=None*)
> Writes material data to an Abaqus .inp file.

> **Parameters output_file** (*str, optional*) – Output file name to write the modifications into. If not given, the original file name is appended with '_mod'.

> **Returns** *None.*

### Notes

- If the output file name is the same as the input file, the original .inp file will be overwritten. This is strongly not recommended.

- The whole content of the original input file is read to memory. It might be a problem for very large .inp files. In that case, a possible implementation could be the following:

  1. Remove old material data

  2. Append new material data to the proper position in the file

  Appending is automatically done at the end of the file. Moving the material description to the end of the file is not possible in general because defining materials cannot be done from any module, i.e. the *MATERIAL keyword cannot follow an arbitrary keyword. In this case, Abaqus throws an AbaqusException with the following message:

  > It can be suboption for the following keyword(s)/level(s): model

**class** grains.abaqus.**Procedure**(*from_Abaqus=False*)
> Bases: object

Handling analysis steps during the simulation.

---

TODO: only document public attributes!

**steps**

> analysis steps, each step collecting the necessary information. Intended for internal representation. To view them, use the *show()* method, to write them to an input file, use the *write()* method.
>
> > **Type** dict

**_Procedure__format**()

> **Formats the data in the Abaqus .inp format.** The internal representation of the steps data in converted to a string understood by Abaqus.
>
> **abaqus_format** [list] List of strings, each element of the list corresponding to a line (with
>
> **line ending)**
>
> > in the Abaqus .inp file. In case of no data, an empty list is returned.
>
> The output is a list so that further concatenation operations are easy. If you want a string, merge the elements of the list: `output = ''.join(output)` This is what the *show()* method does.

**add_analysis**(*step*, *time_period=1.0*, *initial_increment=None*, *min_increment=0*, *max_increment=None*)
Adds an analysis type to a given step.

---

**Note:** Currently only static stress/displacement analysis is supported, indicated by the STATIC Abaqus keyword. No optional parameters are supported.

---

**Todo:** Check user inputs. Define exceptions (*ValueError* class) with the error messages Abaqus CAE throws.

---

> **Parameters**
>
> - **step** (*str*) – Name of the step the analysis type belongs to.
>
> - **initial_increment** (*float, optional*) – Initial time increment. This value will be modified as required if the automatic time stepping scheme is used. If this entry is zero or is not specified, a default value that is equal to the total time period of the step is assumed.
>
> - **time_period** (*float, optional*) – Time period of the step. The default is 1.0.
>
> - **min_increment** (*float, optional*) – Only used for automatic time incrementation. If ABAQUS/Standard finds it needs a smaller time increment than this value, the analysis is terminated. If this entry is zero, a default value of the smaller of the suggested initial time increment or 1e–5 times the total time period is assumed.
>
> - **max_increment** (*float, optional*) – Maximum time increment allowed. Only used for automatic time incrementation. If this value is not specified, no upper limit is imposed, i.e. `max_increment = time_period`.
>
> **Returns** *None*

## Examples

```
>>> proc = Procedure()
>>> proc.create_step('Step-1', max_increments=1000)
>>> proc.add_analysis('Step-1', initial_increment=0.001, min_increment=1e-8)
```

**add_boundary_condition**(*name*, *step*, *nodes*, *first_dof*, *last_dof=None*, *magnitude=0.0*)

Adds boundary condition to a given step.

Boundary conditions can be prescribed on node sets, using the Abaqus keyword BOUNDARY. Optional parameters are not supported.

Multiple boundary conditions can be given on the same node set. It is the responsibility of the user to ensure that the constraints are compatible.

---

**Note:** Currently, boundary conditions can only be defined in a user-defined step, and not in the initial step.

---

**Parameters**

- **name** (*str*) – Name of the boundary condition to be added. Boundary condition names must be unique.

- **step** (*str*) – Name of the step the boundary condition belongs to.

- **nodes** (*str or int*) – If a string, the label of the node set, if a positive integer, the label of a single node on which the boundary condition is prescribed.

- **first_dof** (*int*) – First degree of freedom constrained.

- **last_dof** (*int, optional*) – Last degree of freedom constrained. Not necessary to be given if only one degree of freedom is being constrained.

- **magnitude** (*float*) – Magnitude of the variable. If this magnitude is a rotation, it must be given in radians. The default value is 1.0.

**Returns** *None*

## Examples

Let us consider a two-dimensional structure, fixed on a part of its boundary (called *'left'*), and displaced on another part (called *'right'*). First, we create an analysis step (named *'Step-1'*).

```
>>> proc = Procedure()
>>> proc.create_step('Step-1', max_increments=1000)
```

Now, we prescribe the zero displacements.

```
>>> proc.add_boundary_condition('fixed', 'Step-1', 'left', first_dof=1, last_
↪dof=2)
```

Since we did not specify the magnitude of the displacement components, they are zero by default. The other boundary condition prescribes different values for the horizontal and vertical components. Hence, they are given separately.

```
>>> proc.add_boundary_condition('pulled_right', 'Step-1', 'right', first_
↪dof=1, magnitude=2)
>>> proc.add_boundary_condition('fixed_right', 'Step-1', 'right', first_dof=2,
↪ magnitude=0)
```

Note that we can also prescribe boundary conditions at a particular node, e.g.

```
>>> proc.add_boundary_condition('at_node', 'Step-1', 1, first_dof=2,␣
↪magnitude=-1.2)
```

where we set the vertical displacement component to -1.2 at node 1.

**create_step**(*name*, *nlgeom=False*, *max_increments=100*)

Defines a new step.

A *step* is the fundamental part of the simulation workflow in Abaqus. Among the available options, only the *NLGEOM* and *INC* options are supported currently.

---

**Todo:** Check in the `write()` method whether every step contains an analysis, as this is required by Abaqus.

---

**Parameters**

- **name** (*str*) – Name of the analysis step to be added. Step names must be unique.

- **nlgeom** (*bool, optional*) – Omit this parameter or set to False to perform a geometrically linear analysis during the current step. Set it to True to indicate that geometric nonlinearity should be accounted for during the step. Once the `nlgeom` option has been switched on, it will be active during all subsequent steps in the analysis. The default is False.

- **max_increments** (*int*) – The analysis will stop if the maximum number of increments is exceeded before the complete solution for the step has been obtained. The default is 100.

**Returns** *None*

**read**(*inp_file*)

Reads procedure data from an Abaqus .inp file.

**Parameters inp_file** (*str*) – Abaqus input (.inp) file to be created.

**Returns** *None.*

**Notes**

- This method is designed to read material data. Although the logic could be used to process other properties (parts, assemblies, etc.) in an input file, they are not yet implemented in this class.

- This method assumes that the input file is valid. If it is, the material data can be extacted. If not, the behavior is undefined: the program can crash or return garbage. This is by design: the single responsibility principle dictates that the validity of the input file must be provided by other methods. If the input file was generated from within Abaqus CAE, it is guaranteed to be valid. The *write* method of this class also ensures that the resulting input file is valid. This design choice also makes the program logic simpler. For valid syntax in the input file, check the Input Syntax Rules section in the Abaqus user's guide.

- To read material data from an input file, one has to identify the structure of .inp files in Abaqus. Abaqus is driven by keywords and corresponding data. For a list of accepted keywords, consult the Abaqus Keywords Reference Guide. There are three types of input lines in Abaqus:

  - keyword line: begins with a star, followed by the name of the keyword. Parameters, if any, are separated by commas and are given as parameter-value pairs. Keywords and parameters are not case sensitive. Example:

    *ELASTIC, TYPE=ISOTROPIC, DEPENDENCIES=1

    Some keywords can only be defined once another keyword has already been defined. E.g. the keyword ELASTIC must come after MATERIAL in a valid .inp file.

  - data line: immediately follows a keyword line. All data items must be separated by commas. Example:

    -12.345, 0.01, 5.2E-2, -1.2345E1

  - **comment line: starts with ** and is ignored by Abaqus. Example:** ** This is a comment line

- Internally, the materials are stored in a dictionary. It holds the material data read from the file. The keys in this dictionary are the names of the materials, and the values are dictionaries themselves. Each such dictionary stores a behavior for the given material. E.g. an elastoplastic material is governed by an elastic and a plastic behavior. The parameters for each behavior are stored in a list.

**show**()
> Shows the text for the *step* module, as it appears in the Abaqus .inp file.

> > **Returns** *None.*

**write**(*output_file=None*)
> Writes step definition data to an Abaqus .inp file.

> > **Parameters output_file** (*str, optional*) – Output file name to write the modifications into. If not given, the original file name is appended with '_mod'.

> > **Returns** *None.*

### Notes

- If the output file name is the same as the input file, the original .inp file will be overwritten. This is strongly not recommended.

- The whole content of the original input file is read to memory. It might be a problem for very large .inp files. In that case, a possible implementation could be the following:

  1. Remove old material data

  2. Append new material data to the proper position in the file

  Appending is automatically done at the end of the file. Moving the material description to the end of the file is not possible in general because defining materials cannot be done from any module, i.e. the *MATERIAL keyword cannot follow an arbitrary keyword. In this case, Abaqus throws an AbaqusException with the following message:

  > It can be suboption for the following keyword(s)/level(s): model

grains.abaqus.**extract**(*keyword*)
> Obtains Abaqus keyword and its parameters.

> > **Parameters keyword** (*str*) –

> > Some examples: '*Elastic, type=ORTHOTROPIC' '*Damage Initiation, criterion=HASHIN'
> > '*Nset, nset=Set-1, generate'

> **Returns** **separated** (*list*) – DESCRIPTION.

grains.abaqus.**validate_file**(*file*, *caller*)
    Input or output file validation.

> **Parameters**

> > - **file** (*str*) – Existing Abaqus .inp file or a new .inp file to be created.
> > - **caller** (*{'read', 'write', 'create'}*) – Method name that called this function.

> **Returns** *None.*

# DIC

This module is used to process field values obtained by digital image correlation (DIC). Plotting, comparison with numerical solutions, etc. are implemented.

## 23.1 Classes

| | |
|---|---|
| *DIC* | Performs operations on digital image correlation data. |

### 23.1.1 grains.dic.DIC

**class** grains.dic.**DIC**(*u*, *v*)

    Performs operations on digital image correlation data.

> **Parameters u, v** (*ndarray*) – The two components of the displacement field, discretized on a rectangular grid.
>
> **Raises `ValueError`** – If the displacement components are not discretized on the same grid or if the grid size is not at least 3-by-3. This latter requirement is not a problem in practice (which camera is not capable of shooting photos in 9 pixels resolution?), while it allows the evaluation of the numerical derivatives with second order accuracy on the boundaries too.

#### Notes

Throughout the class methods, the following terms are used. When not precised, *image* refers to the DIC data plotted as an image. We can perceive the image as a *pixel grid*, in which the vertices are the centres of the pixels. An image coordinate system $(X, Y)$ can be defined on this grid such that the vertices have integer coordinates, the origin $A$ is at the top left hand corner, and the $Y$-axis points towards the right, while the $X$-axis points to the bottom. When the experimental result is compared with a numerical solution (assumed to be available at the *nodes* of a *mesh*), one needs to map values defined on the DIC grid onto the nodes of the mesh, and vice versa. The mesh exists on the physical domain, for which a physical coordinate system $(x, y)$ is attached to. Hence, there is a need to express $(X, Y)$ in terms of $(x, y)$. Let $A(a_x, a_y)$ be the origin of the $(X, Y)$ coordinate system given in terms of the $x, y$ coordinates, and $s[\text{pixel/mm}]$ is the scale for the DIC image. The coordinate transformation is then given by

The DIC grid in the physical coordinate system $(x, y)$ is called *physical grid*.

It should be noted that the special alignment of $(x, y)$ with respect to $(X, Y)$ is not a major constraint. In practical applications (when a Cartesian coordinate system is used), the orientation of $(x, y)$ in this way is a convention.

__**init**__ ($u$, $v$)
>    Initialize self. See help(type(self)) for accurate signature.

#### Methods

| | |
|---|---|
| _\_\_init\_\__(u, v) | Initialize self. |
| _equivalent_strain_(strain_tensor, strain_measure) | Computes a scalar quantity from the strain tensor. |
| _plot_displacement_(component[, ax]) | Plots a component of the displacement field. |
| _plot_physicalgrid_([ax]) | Plots the DIC grid in the physical coordinate system. |
| _plot_pixelgrid_([ax]) | Plots the DIC grid in the image coordinate system. |
| _plot_strain_(component[, minval, maxval, . . . ]) | Plots a component of the infinitesimal strain tensor. |
| _plot_superimposedmesh_(mesh, *args, **kwargs) | Plots the DIC grid with a mesh superimposed on it. |
| _project_onto_grid_(nodes, nodal_values[, method]) | Project a numerically computed field onto the DIC grid. |
| _project_onto_mesh_(nodes[, method]) | Project the experimental displacement field onto a mesh. |
| _set_transformation_(origin, . . . ) | Sets the transformation rule between the pixel and the physical coordinate systems. |
| _strain_(strain_measure) | Computes the strain tensor from the displacement vector. |

## 23.2 Functions

| | |
|---|---|
| _plot_strain_(strain[, minval, maxval, . . . ]) | Plots a scalar strain field. |

### 23.2.1 grains.dic.plot_strain

grains.dic.**plot_strain** (*strain*, *minval=None*, *maxval=None*, *colorbar=True*, *label=''*)
>    Plots a scalar strain field.

>    **Parameters**

>    - **strain** (*ndarray*) – Scalar strain field sampled on an m-by-n grid, given as a 2D numpy array.

>    - **minval, maxval** (*float, optional*) – Set the color scaling for the image by fixing the values that map to the colormap color limits. If `minval` is not provided, the limit is determined from the minimum value of the data. Similarly for `maxval`.

>    - **colorbar** (*bool, optional*) – If True, a horizontal colorbar is placed below the strain map. The default is True.

>    - **label** (*str, optional*) – Label describing the strain field. LaTeX code is also accepted, e.g. *r'$varepsilon_{yy}$'*. If not given, no text is displayed.

>    **Returns** *None*

**See also:**

*DIC.strain()*

Deprecated since version 1.1.0: This will be removed in 1.3.0. This function will be a static method of the `DIC` class. See also the deprecation warning of *DIC.plot_strain()*.

**class** grains.dic.**DIC**(*u*, *v*)

Bases: object

Performs operations on digital image correlation data.

> **Parameters u, v** (*ndarray*) – The two components of the displacement field, discretized on a rect-angular grid.

> **Raises ValueError** – If the displacement components are not discretized on the same grid or if the grid size is not at least 3-by-3. This latter requirement is not a problem in practice (which camera is not capable of shooting photos in 9 pixels resolution?), while it allows the evaluation of the numerical derivatives with second order accuracy on the boundaries too.

**Notes**

Throughout the class methods, the following terms are used. When not precised, *image* refers to the DIC data plotted as an image. We can perceive the image as a *pixel grid*, in which the vertices are the centres of the pixels. An image coordinate system $(X, Y)$ can be defined on this grid such that the vertices have integer coordinates, the origin $A$ is at the top left hand corner, and the $Y$-axis points towards the right, while the $X$-axis points to the bottom. When the experimental result is compared with a numerical solution (assumed to be available at the *nodes* of a *mesh*), one needs to map values defined on the DIC grid onto the nodes of the mesh, and vice versa. The mesh exists on the physical domain, for which a physical coordinate system $(x, y)$ is attached to. Hence, there is a need to express $(X, Y)$ in terms of $(x, y)$. Let $A(a_x, a_y)$ be the origin of the $(X, Y)$ coordinate system given in terms of the $x, y$ coordinates, and $s[\text{pixel/mm}]$ is the scale for the DIC image. The coordinate transformation is then given by

The DIC grid in the physical coordinate system $(x, y)$ is called *physical grid*.

It should be noted that the special alignment of $(x, y)$ with respect to $(X, Y)$ is not a major constraint. In practical applications (when a Cartesian coordinate system is used), the orientation of $(x, y)$ in this way is a convention.

**static equivalent_strain**(*strain_tensor*, *strain_measure*)

Computes a scalar quantity from the strain tensor.

> **Parameters**

> - **strain_tensor** (*ndarray*) – Components of the strain tensor $\varepsilon$ as an $m \times n \times 3$ array. The first two dimensions correspond to the grid points they are determined at, the third dimension gives the components of the tensor in the following order: $\varepsilon_{11}, \varepsilon_{12}, \varepsilon_{22}$.

> - **strain_measure** (*{'von Mises'}*) – One of the following strain measures.

>   – 'von Mises'

$$\varepsilon_M := \sqrt{\frac{2}{3}\varepsilon^{\text{dev}} : \varepsilon^{\text{dev}}} = \sqrt{\frac{2}{3}\left(\varepsilon : \varepsilon - \frac{1}{3}(\text{tr}\varepsilon)^2\right)}$$

where $\varepsilon^{\mathrm{dev}}$ is the deviatoric part of the strain tensor. This can further be simplified (see the notes below) to

$$\varepsilon_M = \frac{2}{3}\sqrt{\varepsilon_{11}^2 + \varepsilon_{22}^2 + 3\varepsilon_{12}^2 - \varepsilon_{11}\varepsilon_{22}}$$

**Returns** *ndarray* – Equivalent strain at the grid points, given as an $m \times n$ numpy array.

See also:

*strain()*

### Notes

Under plane stress conditions, $\varepsilon_{33}$ is not zero, but it is computed as

$$\varepsilon_{33} = -\frac{\nu}{E}(\sigma_{11} + \sigma_{22})$$

Since the stresses are not known from the DIC, one must settle with plane strain conditions where $\varepsilon_{33} = 0$. This is what we follow in the *DIC* class. We remark that in stereo-DIC multiple cameras are used, which allows the measurement of $\varepsilon_{33}$.

**plot_displacement**(*component*, *ax=None*)
Plots a component of the displacement field.

**Parameters**

- **component** (*{1, 2}*) – Component to be plotted, where 1 corresponds to the first, 2 corresponds to the second component of the displacement field.

- **ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the grid resides in. The default is None, in which case a new *Axes* within a new figure is created.

**Returns** *None*

See also:

*plot_strain()*

**plot_physicalgrid**(*ax=None*)
Plots the DIC grid in the physical coordinate system.

This method is only available once the relation between the image coordinate system and the physical coordinate system has been set up by the *set_transformation()* method.

**Parameters ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the grid resides in. The default is None, in which case a new *Axes* within a new figure is created.

**Returns ax** (*matplotlib.axes.Axes*)

See also:

*plot_pixelgrid()*

**Notes**

See in the `plot_pixelgrid()` method.

**plot_pixelgrid**(*ax=None*)

Plots the DIC grid in the image coordinate system.

> **Parameters ax** (*matplotlib.axes.Axes, optional*) – The *Axes* instance the grid resides in. The default is None, in which case a new *Axes* within a new figure is created.
>
> **Returns ax** (*matplotlib.axes.Axes*)

**See also:**

`plot_physicalgrid()`

**Notes**

For a DIC image with $m \times n$ number of pixels, the number of grid lines is $(m+1)+(n+1)$. Plotting all these lines would not only slow down the program, it would also make the grid practically indistinguishable from a filled rectangle. Therefore, for high resolution images, only grid lines around the boundary of the image are plotted. The target resolution above which this strategy is used can be given in the class constructor.

**plot_strain**(*component*, *minval=None*, *maxval=None*, *colorbar=True*)

Plots a component of the infinitesimal strain tensor.

The partial derivatives of the displacement field are computed with numerical differentiation.

> **Parameters**
>
> - **component** (*tuple, {(1,1), (1,2), (2,1), (2,2)}*) – Component to be plotted, where
>   - (1,1) denotes $\varepsilon_{11}$
>   - (1,2) and (2,1) denote $\varepsilon_{12} = \varepsilon_{21}$
>   - (2,2) denotes $\varepsilon_{22}$
>
>   for the infinitesimal strain tensor
>
> $$\varepsilon = \begin{pmatrix} \varepsilon_{11} & \varepsilon_{12} \\ \varepsilon_{21} & \varepsilon_{22} \end{pmatrix}$$
>
> - **minval, maxval** (*float, optional*) – Set the color scaling for the image by fixing the values that map to the colormap color limits. If `minval` is not provided, the limit is determined from the minimum value of the data. Similarly for `maxval`.
>
> - **colorbar** (*bool, optional*) – If True, a horizontal colorbar is placed below the strain map. The default is True.
>
> **Returns** *None*

**See also:**

`plot_displacement()`, `numpy.gradient()`

Deprecated since version 1.1.0: This will be removed in 1.3.0. This method will be modified to perform the plotting only. The computation of the various strain measures will be done in the `strain()` method. The function that replaces this function will keep the same name and is currently implemented outside this class.

**plot_superimposedmesh**(*mesh*, *\*args*, *\*\*kwargs*)
    Plots the DIC grid with a mesh superimposed on it.

    Since the finite element mesh represents a physical domain, the DIC grid is plotted in the physical coordinate system, which is determined by the `set_transformation()` method.

---

**Note:** Maybe no need to couple this module with the *geometry* module just for the sake of this function. It is probably easier to simply pass the nodes and the elements, and call *triplot*.

---

> **Parameters**
>
> > - **mesh** (*Mesh*) – A subclass of `grains.geometry.Mesh` to be plotted along with the physical DIC grid.
> >
> > - **args** (*optional*) – The same as for `matplotlib.axes.Axes.plot()`. The settings given here influence the mesh plotting.
> >
> > - **kwargs** (*optional*) – The same as for `matplotlib.axes.Axes.plot()`. The settings given here influence the mesh plotting.
>
> **Returns** **ax** (*matplotlib.axes.Axes*) – The Axes object on which the plot is drawn.

**See also:**

`grains.geometry.TriMesh.plot()`

### Examples

Let us create a grid and a random mesh (the same as in the *Examples* section of the `project_onto_mesh()`).

```
>>> from grains.dic import DIC
>>> from grains.geometry import TriMesh
>>> x_grid, y_grid = np.mgrid[-1:1:100j, 1:2:50j]
>>> exact_solution = lambda x, y: 1 - (x + y**2) * np.sign(x)
>>> grid = DIC(np.random.rand(*np.shape(x_grid.T)), np.random.rand(*np.
↪shape(x_grid.T)))
>>> grid.set_transformation((-1, 2), 50)
>>> n_nodes = 100   # modify this to see how good the interpolated solution is
>>> msh = TriMesh(*TriMesh.sample_mesh(1, n_nodes))
>>> grid.plot_superimposedmesh(msh, linewidth=3, markerfacecolor='b')
>>> plt.show()
```

**project_onto_grid**(*nodes*, *nodal_values*, *method='linear'*)
    Project a numerically computed field onto the DIC grid.

---

**Todo:** Use the example code of this method to create the plotting method of this class. When done, rewrite the example with the methods.

---

> **Parameters**
>
> > - **nodes** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a node of the mesh, and the two columns giving the Cartesian coordinates of the nodes.
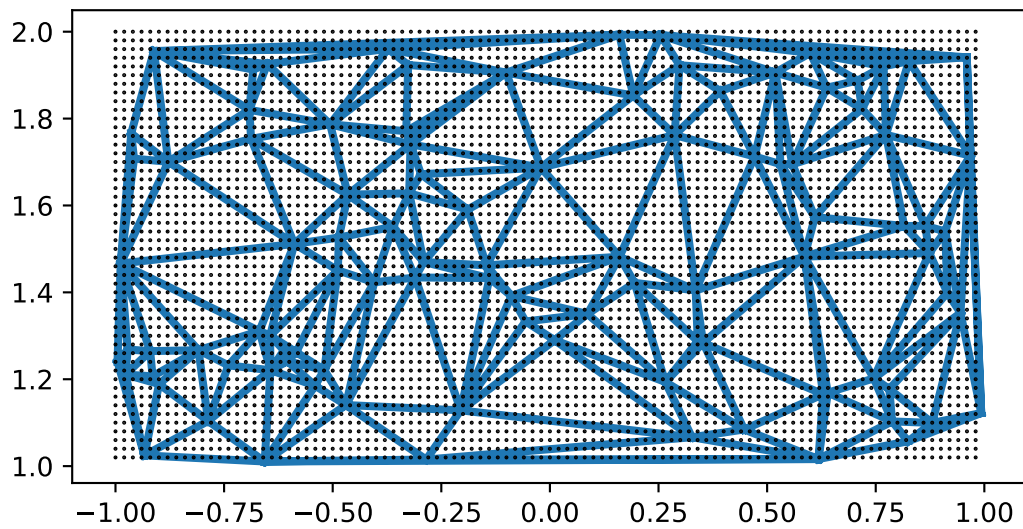
- **nodal_values** (*ndarray*) – 1D numpy array (vector), the numerically computed field at the nodes of the mesh.

- **method** (*{'nearest', 'linear', 'cubic'}, optional*) – Type of the interpolation. The default is linear.

**Returns** *ndarray* – 2D numpy array, the projected field values at the physical DIC grid points.

**See also:**

*project_onto_mesh()*, scipy.interpolate.griddata()

## Examples

The DIC grid and the nodal finite element values (interpolated in-between, which is an errenous way to visualize discontinuous functions) can be seen in the upper left and lower left figures, respectively.

```python
>>> from grains.dic import DIC
>>> from grains.geometry import TriMesh
>>> x_grid, y_grid = np.mgrid[-1:1:100j, 1:2:50j]
>>> exact_solution = lambda x, y: 1 - (x + y**2) * np.sign(x)
>>> grid = DIC(exact_solution(x_grid, y_grid).T, exact_solution(x_grid, y_
→grid).T)
>>> grid.set_transformation((-1, 2), 50)
```

The finite element solution is obtained at the nodes (upper right figure) of the mesh. Here, we assumed that the nodes are sampled from a uniformly random distribution on $[-1, 1] \times [1, 2]$.

```python
>>> n_nodes = 100   # modify this to see how good the interpolated solution is
```

For the sake of this example, we also assume that the nodal values are "exact", i.e. they admit the function values at the nodes. In reality, of course, this will not be the case, but this allows us to investigate the effect moving from the FE mesh to the DIC grid.

```python
>>> mesh = TriMesh(*TriMesh.sample_mesh(1, n_nodes))
>>> fe_values = exact_solution(mesh.vertices[:, 0], mesh.vertices[:, 1])
>>> mesh.associate_field(fe_values)
>>> interpolated = grid.project_onto_grid(mesh.vertices, fe_values, method=
→'linear')
```

The FE field available at the nodes are interpolated at the DIC grid points, as shown in the lower right figure. Note that no extrapolation is performed, so values are not computed at the grid points lying outside the convex hull of the finite element nodes.

```python
>>> ax = []
>>> ax.append(plt.subplot(221))
>>> plt.plot(x_grid, y_grid, 'k.', ms=1)
>>> plt.title('DIC grid')
>>> ax.append(plt.subplot(222))
>>> plt.plot(mesh.vertices[:, 0], mesh.vertices[:, 1], 'k.', ms=1)
>>> plt.title('FE nodes')
>>> ax.append(plt.subplot(223))
>>> mesh.plot_field(0, ax=ax[-1])
>>> plt.title('FE field')
>>> ax.append(plt.subplot(224))
>>> plt.imshow(interpolated, extent=(-1, 1, 1, 2), origin='lower', vmin=-4,
→vmax = 5)
```

(continues on next page)

```
>>> plt.title('FE field interpolated at the DIC grid')
>>> plt.tight_layout()
>>> for a in ax:
...     a.set_aspect('equal', 'box')
>>> plt.show()
```
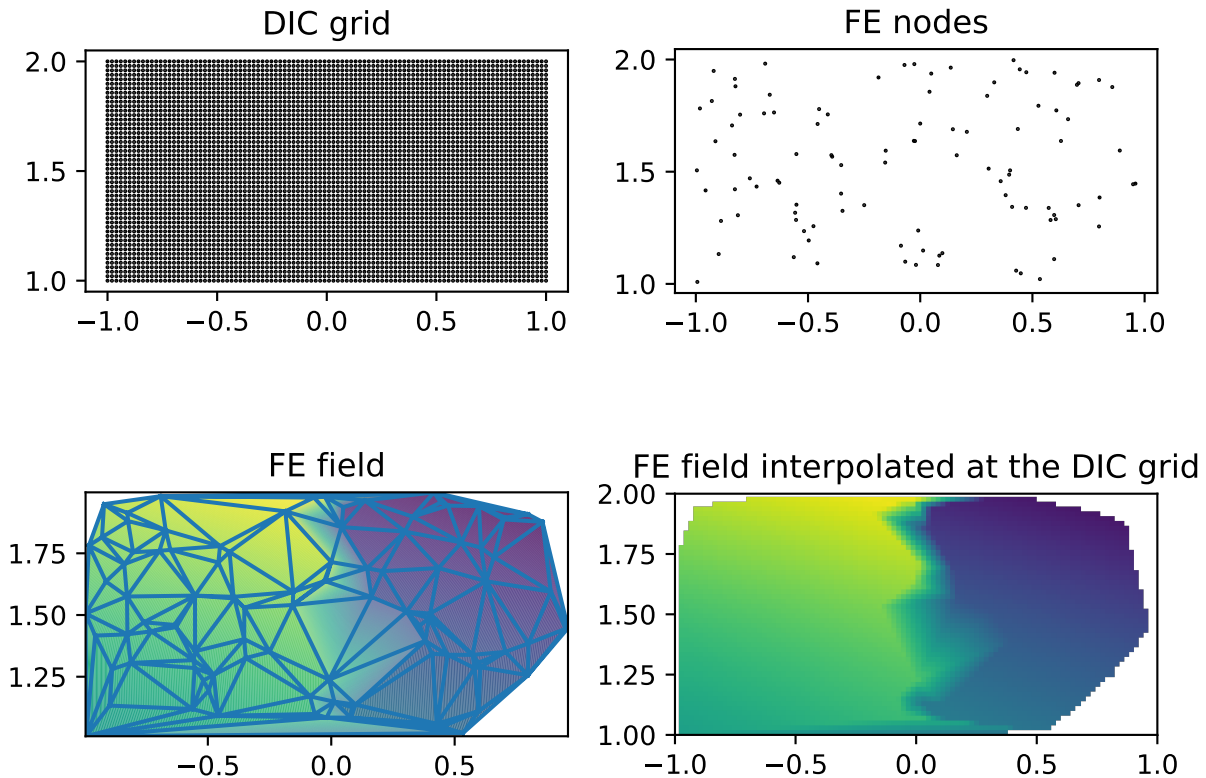


You are invited to modify this example to simulate a finer mesh by increasing `n_nodes`. Also try different interpolation techniques.

**project_onto_mesh** (*nodes*, *method='linear'*)
Project the experimental displacement field onto a mesh.

> **Parameters** **nodes** (*ndarray*) – 2D numpy array with 2 columns, each row corresponding to a node of the mesh, and the two columns giving the Cartesian coordinates of the nodes.
>
> **Returns**
>
> - **u_nodes, v_nodes** (*ndarray*) – 1D numpy arrays (vectors), the components of the projected displacement field at the given `nodes`.
>
> - **method** (*{'nearest', 'linear', 'cubic'}, optional*) – Type of the interpolation. The default is linear.
>
> **See also:**
>
> *project_onto_grid()*

### Examples

Suppose we have measured field data, obtained by DIC. The experimental field values are obtained as an image. The pixel centers of this image form a grid, each grid point having an associated scalar value, the value of the measured field. In this example, we pretend that the measured field is given as a function. The DIC grid and the measured field values can be seen in the upper left and lower left figures, respectively.

```
>>> from grains.dic import DIC
>>> from grains.geometry import TriMesh
>>> x_grid, y_grid = np.mgrid[-1:1:100j, 1:2:50j]
>>> exact_solution = lambda x, y: 1 - (x + y**2) * np.sign(x)
>>> grid = DIC(exact_solution(x_grid, y_grid).T, exact_solution(x_grid,y_
↪grid).T)
```

The finite element solution is obtained at the nodes (upper right figure) of the mesh. Here, we assumed that the nodes are sampled from a uniformly random distribution on $[-1, 1] \times [1, 2]$.

```
>>> n_nodes = 100  # modify this to see how good the interpolated solution is
>>> mesh = TriMesh(*TriMesh.sample_mesh(1, n_nodes))
```

For the sake of this example, we also assume that the nodal values are "exact", i.e. they are the function values at the nodes. In reality, of course, this will not be the case, but this allows us to investigate the effect moving from the FE mesh to the DIC grid.

```
>>> grid.set_transformation((-1, 2), 50)
>>> fe_values = exact_solution(mesh.vertices[:, 0], mesh.vertices[:, 1])
>>> interpolated = grid.project_onto_mesh(mesh.vertices, 'nearest')
```

The FE solution available at the nodes are interpolated at the DIC grid points, as shown in the lower right figure. Interpolation with continuous functions cannot resolve well the discontinuity present in the "exact solution", i.e. in the measurement data. A discontinuous manufactured solution was intentionally prepared to illustrate this.

```
>>> ax = []
>>> ax.append(plt.subplot(221))
>>> grid.plot_physicalgrid(ax[0])
>>> plt.title('DIC grid')
>>> ax.append(plt.subplot(222))
>>> mesh.plot('k.', ax=ax[1], markersize=1)
>>> plt.title('FE nodes')
>>> ax.append(plt.subplot(223))
>>> plt.imshow(exact_solution(x_grid, y_grid).T, extent=(-1, 1, 1, 2),
↪origin='lower',
...            vmin=-4, vmax = 5)
>>> plt.title('Exact solution on the DIC grid')
>>> ax.append(plt.subplot(224))
>>> interpolated[0][np.isnan(interpolated[0])] = 0
>>> mesh.associate_field(interpolated[0])
>>> mesh.plot_field(0, show_mesh=False, ax=ax[3])
>>> plt.title('FE solution interpolated at the DIC grid')
>>> plt.show()
```

You are invited to modify this example to simulate a finer mesh by increasing `n_nodes`. Also try different interpolation techniques.

**set_transformation**(*origin*, *pixels_per_physicalunit*)
    Sets the transformation rule between the pixel and the physical coordinate systems.
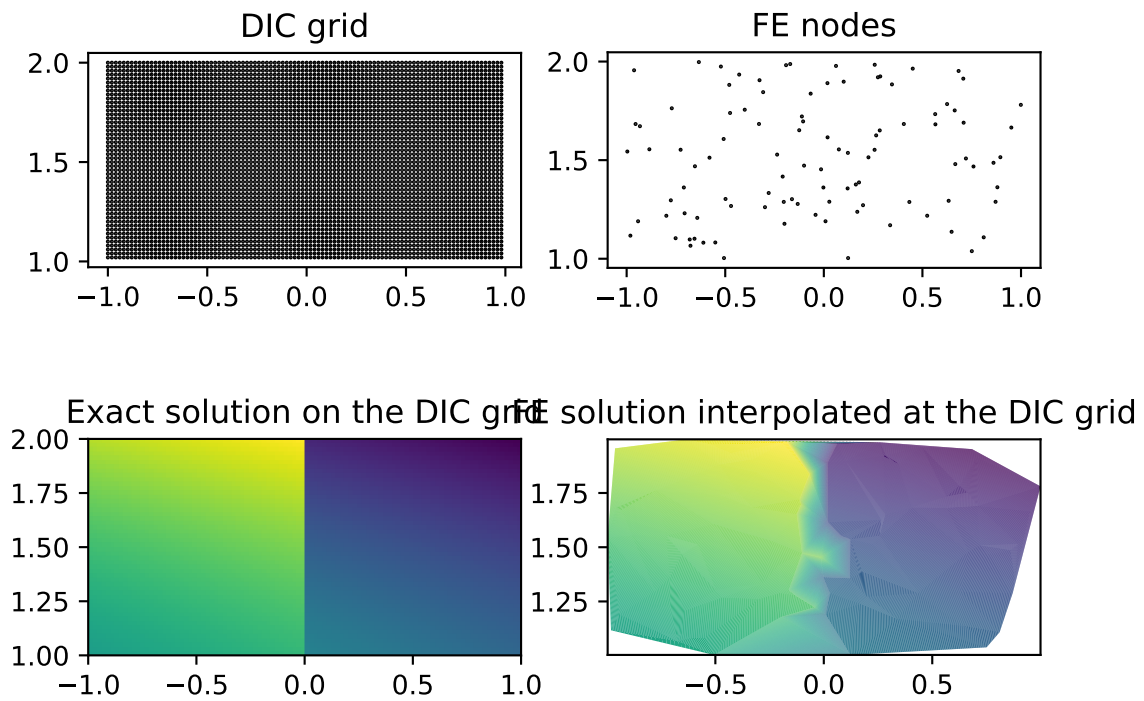
To determine the position and the size of the DIC image in the physical space, a transformation rule needs to be given, as described in the *Notes* section of the `DIC` class.

> **Parameters**
>
> - **origin** (*tuple*) – 2-tuple of float, the coordinates of the origin of the DIC grid (upper left corner) in the physical coordinate system.
>
> - **pixels_per_physicalunit** (*float*)
>
> **Returns** *None*

**strain**(*strain_measure*)

Computes the strain tensor from the displacement vector.

The symmetric strain tensor $\varepsilon$ with components

$$\varepsilon = \begin{pmatrix} \varepsilon_{11} & \varepsilon_{12} \\ \varepsilon_{21} & \varepsilon_{22} \end{pmatrix}$$

is computed for the given strain measure. The partial derivatives of the displacement field $(u, v)$, available on an $m \times n$ grid, are computed with numerical differentiation with second order accuracy.

> **Parameters strain_measure** (*{'infinitesimal', 'Green-Lagrange'}*) – One of the following strain measures.
>
> - 'infinitesimal'

$$\varepsilon_{11} = \frac{\partial u}{\partial x}$$
$$\varepsilon_{12} = \frac{1}{2} \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right)$$
$$\varepsilon_{22} = \frac{\partial v}{\partial y}$$

> - 'Green-Lagrange'

$$\varepsilon_{11} = \frac{1}{2} \left( 2\frac{\partial u}{\partial x} + \left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 \right)$$
$$\varepsilon_{12} = \frac{1}{2} \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} + \frac{\partial u}{\partial x}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\frac{\partial v}{\partial y} \right)$$
$$\varepsilon_{22} = \frac{1}{2} \left( 2\frac{\partial v}{\partial y} + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 \right)$$

> **Returns strain_tensor** (*ndarray*) – Components of the strain tensor as an $m \times n \times 3$ array. The first two dimensions correspond to the grid points they are determined at, the third dimension gives the components of the tensor in the following order: $\varepsilon_{11}, \varepsilon_{12}, \varepsilon_{22}$.

**See also:**

`equivalent_strain()`, `plot_strain()`, `numpy.gradient()`

**CristalX, Release 1.1.0**

### Notes

The derivatives of the displacement field are computed in the the physical coordinate system.

### Examples

Consider a rectangular body with a linear displacement function in horizontal direction and no displacement vertically. The deformation of the body is such that only the horizontal strain is nonzero.

```
>>> u = np.array([[0, 1e-3, 2e-3, 3e-3], [0, 1e-3, 2e-3, 3e-3], [0, 1e-3, 2e-
↪3, 3e-3]])
>>> v = np.zeros((3,4))
>>> d = DIC(u, v)
>>> E = d.strain('infinitesimal')
>>> E[:, :, 0]
array([[0.001, 0.001, 0.001, 0.001],
       [0.001, 0.001, 0.001, 0.001],
       [0.001, 0.001, 0.001, 0.001]])
>>> np.allclose(E[:, :, 2], 0)
True
>>> np.allclose(E[:, :, 1], 0)
True
```

The Green-Lagrange strain tensor is slightly different. Comparing with the infinitesimal strain tensor shows how good the assumption of small deformations is.

```
>>> E_gl = d.strain('Green-Lagrange')
>>> E_gl[:, :, 0] - E[:, :, 0]
array([[5.e-07, 5.e-07, 5.e-07, 5.e-07],
       [5.e-07, 5.e-07, 5.e-07, 5.e-07],
       [5.e-07, 5.e-07, 5.e-07, 5.e-07]])
```

The other two strain components remain zero.

```
>>> np.allclose(E_gl[:, :, 1:2], 0)
True
```

grains.dic.**plot_strain**(*strain*, *minval=None*, *maxval=None*, *colorbar=True*, *label=''*)
    Plots a scalar strain field.

**Parameters**

- **strain** (*ndarray*) – Scalar strain field sampled on an m-by-n grid, given as a 2D numpy array.

- **minval, maxval** (*float, optional*) – Set the color scaling for the image by fixing the values that map to the colormap color limits. If `minval` is not provided, the limit is determined from the minimum value of the data. Similarly for `maxval`.

- **colorbar** (*bool, optional*) – If True, a horizontal colorbar is placed below the strain map. The default is True.

- **label** (*str, optional*) – Label describing the strain field. LaTeX code is also accepted, e.g. *r'$varepsilon_{yy}$'*. If not given, no text is displayed.

**Returns** *None*

**See also:**

*DIC.strain()*

Deprecated since version 1.1.0: This will be removed in 1.3.0. This function will be a static method of the `DIC` class. See also the deprecation warning of *DIC.plot_strain()*.

# SIMULATION

## 24.1 Functions

### 24.1.1 grains.simulation.data_Pierre

grains.simulation.**data_Pierre**()

Yield stresses and average grain diameters from Pierre's thesis.

> **Returns**
>
> > • **sigma_y** (*list of floats*) – Yield stresses.
> >
> > • **d** (*list of floats*) – Diameters of the grains.

### 24.1.2 grains.simulation.hallpetch_constants

grains.simulation.**hallpetch_constants**(*sigma_y*, *d*)

Determines the two Hall-Petch constants. Given available measurements for the grains sizes and the yield stresses, the two constants in the Hall-Petch formula are computed.

> **Parameters**
>
> > • **sigma_y** (*list of floats*) – Yield stresses.
> >
> > • **d** (*list of floats*) – Diameters of the grains.
>
> **Returns**
>
> > • **sigma_0** (*float*) – Starting stress for dislocation movement (material constant).
> >
> > • **k** (*float*) – Strengthening coefficient (material constant).

**Notes**

If two data points are given in the inputs (corresponding to two measurements), the output parameters have unique values:

$$k = (\text{sigma\_y}[0] - \text{sigma\_y}[1]) / (1/\text{sqrt}(d[0]) - 1/\text{sqrt}(d[1])) \quad \text{sigma\_0} = \text{sigma\_y}[0] - k/\text{sqrt}(d[0])$$

If there are more than two measurements, the resulting linear system is overdetermined. In both cases, the outputs are determined using least squares fitting.

### 24.1.3 grains.simulation.hallpetch

grains.simulation.**hallpetch**(*sigma_0*, *k*, *d*)

Computes the yield stress from the Hall-Petch relation.

> **Parameters**
>
> > - **sigma_0** (*float*) – Starting stress for dislocation movement (material constant).
> >
> > - **k** (*float*) – Strengthening coefficient (material constant).
> >
> > - **d** (*float or list of floats*) – Diameter of the grain.
>
> **Returns** **sigma_y** (*float or list of floats*) – Yield stress.

### 24.1.4 grains.simulation.hallpetch_plot

grains.simulation.**hallpetch_plot**(*sigma_y*, *d*, *units=('MPa', 'mm')*)

Plots the Hall-Petch formula for given grain sizes and yield stresses.

> **Parameters**
>
> > - **sigma_y** (*ndarray*) – Yield stresses.
> >
> > - **d** (*ndarray*) – Grain diameters.
> >
> > - **units** (*2-tuple of str, optional*) – Units for the yield stress and the grain diameters. The default is ("MPa","mm").
>
> **Returns** **fig** (*matplotlib.figure.Figure*) – The figure object is returned in case further manipulations are necessary.

### 24.1.5 grains.simulation.change_domain

grains.simulation.**change_domain**(*image*, *left*, *right*, *bottom*, *top*, *padding_value=nan*)

Extends or crops the domain an image fills.

The original image is extended, cropped or left unchanged in the left, right, bottom and top directions by padding the corresponding 2D or 3D array with a given value or removing existing elements. Non-integer image length is avoided by rounding up. This choice prevents ending up with an empty image during cropping or no added pixel during extension.

> **Parameters**
>
> > - **image** (*ndarray*) – 2D or 3D numpy array representing the original image.
> >
> > - **left, right** (*float*) – When positive, the domain is extended, when negative, the domain is cropped by the given value relative to the image width.

- **bottom, top** (*float*) – When positive, the domain is extended, when negative, the domain is cropped by the given value relative to the image height.

- **padding_value** (*float, optional*) – Value for the added pixels. The default is numpy.nan.

**Returns** **changed_image** (*ndarray*) – 2D or 3D numpy array representing the modified domain.

### Examples

Crop an image at the top, extended it at the bottom and on the left, and leave it unchanged on the right. Note the rounding for non-integer pixels.

```
>>> import numpy as np
>>> image = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])
>>> modified = change_domain(image, 0.5, 0, 1/3, -1/3, 0)
>>> image  # no in-place modification, the original image is not overwritten
array([[0.1, 0.2, 0.3],
       [0.4, 0.5, 0.6],
       [0.7, 0.8, 0.9]])
>>> modified
array([[0. , 0. , 0.4, 0.5, 0.6],
       [0. , 0. , 0.7, 0.8, 0.9],
       [0. , 0. , 0. , 0. , 0. ]])
```

## 24.1.6 grains.simulation.nature_of_deformation

grains.simulation.**nature_of_deformation**(*microstructure*, *strain_field*, *interface_width=3*, *visualize=True*)

Characterizes the intergranular/intragranular deformations.

To decide whether the strain localization is intergranular (happens along grain boundaries, also called interfaces) or intragranular in a given microstructure, the strain field is projected on the microstructure. Here, by strain field we mean a scalar field, often called *equivalent strain* that is derived from a strain tensor.

It is irrelevant for this function whether the strain field is obtained from a numerical simulation or from a (post-processed) full-field measurement. All what matters is that the strain field be available on a grid of the same size as the microstructure.

The strain field is assumed to be localized on an interface if its neighborhood, with band width defined by the user, contains higher strain values than what is outside the band (i.e. the grain interiors). A too large band width identifies small grains to have boundary only, without any interior. This means that even if the strain field in reality localizes inside such small grains, the localization is classified as intergranular. However, even for extreme deformations, one should not expect that the strain localizes on an interface with a single-point width. Moreover, using a too small band width is susceptible to the exact position of the interfaces, which are extracted from the grain microstructure. A judicial balance needs to be achieved in practice.

**Parameters**

- **microstructure** (*ndarray*) – Labelled image corresponding to the segmented grains in a grain microstructure.

- **strain_field** (*ndarray*) – Discrete scalar field of the same size as the `microstructure`.

- **interface_width** (*int, optional*) – Thickness of the band around the interfaces.

- **visualize** (*bool, optional*) – If True, three plots are created. Two of them show the deformation field within the bands and outside the bands. They are linked together, so when

you pan or zoom on one, the other plot will follow. The third plot contains two histograms on top of each other, giving the frequency of the strain values within the bands and outside the bands.

> **Returns**
>
> - **boundary_strain** (*ndarray*) – Copy of `strain_field`, but values outside the band are set to NaN.
>
> - **bulk_strain** (*ndarray*) – Copy of `strain_field`, but values within the band are set to NaN.
>
> - **bands** (*ndarray*) – Boolean array of the same size as `strain_field`, with True values corresponding to the band.

**See also:**

*`grains.dic.DIC.strain()`* Computes a strain tensor from the displacement field.

*`grains.dic.DIC.equivalent_strain()`* Extracts a scalar quantity from a strain tensor.

`matplotlib.pyplot.hist()` Plots a histogram.

### Notes

1. From the modelling viewpoint, it is important to know whether the strain localizes to the grain boundaries or it is dominant within the grains as well. In the former case, simplifications in the models save computational time in the simulations.

2. In dynamics, the evolution of the strain field is relevant. E.g. an initially intergranular deformation can turn into diffuse localization that occurs within the grains as well. In that case, a strain field must be obtained at each time step, and this function can be called for each such instance.

### Examples

The following figure was created by this function with `visualize` set to *True* and `band_width` chosen to be 3.

grains.simulation.**change_domain**(*image*, *left*, *right*, *bottom*, *top*, *padding_value=nan*)
Extends or crops the domain an image fills.

The original image is extended, cropped or left unchanged in the left, right, bottom and top directions by padding the corresponding 2D or 3D array with a given value or removing existing elements. Non-integer image length is avoided by rounding up. This choice prevents ending up with an empty image during cropping or no added pixel during extension.

> **Parameters**
>
> - **image** (*ndarray*) – 2D or 3D numpy array representing the original image.
>
> - **left, right** (*float*) – When positive, the domain is extended, when negative, the domain is cropped by the given value relative to the image width.
>
> - **bottom, top** (*float*) – When positive, the domain is extended, when negative, the domain is cropped by the given value relative to the image height.
>
> - **padding_value** (*float, optional*) – Value for the added pixels. The default is numpy.nan.

> **Returns**  **changed_image** (*ndarray*) – 2D or 3D numpy array representing the modified domain.

**Examples**

Crop an image at the top, extended it at the bottom and on the left, and leave it unchanged on the right. Note the rounding for non-integer pixels.

```
>>> import numpy as np
>>> image = np.array([[0.1, 0.2, 0.3], [0.4, 0.5, 0.6], [0.7, 0.8, 0.9]])
>>> modified = change_domain(image, 0.5, 0, 1/3, -1/3, 0)
>>> image  # no in-place modification, the original image is not overwritten
array([[0.1, 0.2, 0.3],
       [0.4, 0.5, 0.6],
       [0.7, 0.8, 0.9]])
>>> modified
array([[0. , 0. , 0.4, 0.5, 0.6],
       [0. , 0. , 0.7, 0.8, 0.9],
       [0. , 0. , 0. , 0. , 0. ]])
```

grains.simulation.**data_Pierre**()
    Yield stresses and average grain diameters from Pierre's thesis.

> **Returns**
>
> > - **sigma_y** (*list of floats*) – Yield stresses.
> >
> > - **d** (*list of floats*) – Diameters of the grains.

grains.simulation.**hallpetch**(*sigma_0*, *k*, *d*)
    Computes the yield stress from the Hall-Petch relation.

> **Parameters**
>
> > - **sigma_0** (*float*) – Starting stress for dislocation movement (material constant).
> >
> > - **k** (*float*) – Strengthening coefficient (material constant).
> >
> > - **d** (*float or list of floats*) – Diameter of the grain.
>
> **Returns** **sigma_y** (*float or list of floats*) – Yield stress.

grains.simulation.**hallpetch_constants**(*sigma_y*, *d*)
    Determines the two Hall-Petch constants. Given available measurements for the grains sizes and the yield stresses, the two constants in the Hall-Petch formula are computed.

> **Parameters**
>
> > - **sigma_y** (*list of floats*) – Yield stresses.
> >
> > - **d** (*list of floats*) – Diameters of the grains.
>
> **Returns**
>
> > - **sigma_0** (*float*) – Starting stress for dislocation movement (material constant).
> >
> > - **k** (*float*) – Strengthening coefficient (material constant).

**Notes**

If two data points are given in the inputs (corresponding to two measurements), the output parameters have unique values:

k = (sigma_y[0] - sigma_y[1]) / (1/sqrt(d[0]) - 1/sqrt(d[1])) sigma_0 = sigma_y[0] - k/sqrt(d[0])

If there are more than two measurements, the resulting linear system is overdetermined. In both cases, the outputs are determined using least squares fitting.

grains.simulation.**hallpetch_plot**(*sigma_y*, *d*, *units=('MPa', 'mm')*)

Plots the Hall-Petch formula for given grain sizes and yield stresses.

> **Parameters**
>
> - **sigma_y** (*ndarray*) – Yield stresses.
>
> - **d** (*ndarray*) – Grain diameters.
>
> - **units** (*2-tuple of str, optional*) – Units for the yield stress and the grain diameters. The default is ("MPa","mm").
>
> **Returns** **fig** (*matplotlib.figure.Figure*) – The figure object is returned in case further manipulations are necessary.

grains.simulation.**nature_of_deformation**(*microstructure*, *strain_field*, *interface_width=3*, *visualize=True*)

Characterizes the intergranular/intragranular deformations.

To decide whether the strain localization is intergranular (happens along grain boundaries, also called interfaces) or intragranular in a given microstructure, the strain field is projected on the microstructure. Here, by strain field we mean a scalar field, often called *equivalent strain* that is derived from a strain tensor.

It is irrelevant for this function whether the strain field is obtained from a numerical simulation or from a (post-processed) full-field measurement. All what matters is that the strain field be available on a grid of the same size as the microstructure.

The strain field is assumed to be localized on an interface if its neighborhood, with band width defined by the user, contains higher strain values than what is outside the band (i.e. the grain interiors). A too large band width identifies small grains to have boundary only, without any interior. This means that even if the strain field in reality localizes inside such small grains, the localization is classified as intergranular. However, even for extreme deformations, one should not expect that the strain localizes on an interface with a single-point width. Moreover, using a too small band width is susceptible to the exact position of the interfaces, which are extracted from the grain microstructure. A judicial balance needs to be achieved in practice.

> **Parameters**
>
> - **microstructure** (*ndarray*) – Labelled image corresponding to the segmented grains in a grain microstructure.
>
> - **strain_field** (*ndarray*) – Discrete scalar field of the same size as the `microstructure`.
>
> - **interface_width** (*int, optional*) – Thickness of the band around the interfaces.
>
> - **visualize** (*bool, optional*) – If True, three plots are created. Two of them show the deformation field within the bands and outside the bands. They are linked together, so when you pan or zoom on one, the other plot will follow. The third plot contains two histograms on top of each other, giving the frequency of the strain values within the bands and outside the bands.
>
> **Returns**
>
> - **boundary_strain** (*ndarray*) – Copy of `strain_field`, but values outside the band are set to NaN.

- **bulk_strain** (*ndarray*) – Copy of `strain_field`, but values within the band are set to NaN.

- **bands** (*ndarray*) – Boolean array of the same size as `strain_field`, with True values corresponding to the band.

**See also:**

`grains.dic.DIC.strain()` Computes a strain tensor from the displacement field.

`grains.dic.DIC.equivalent_strain()` Extracts a scalar quantity from a strain tensor.

`matplotlib.pyplot.hist()` Plots a histogram.

## Notes

1. From the modelling viewpoint, it is important to know whether the strain localizes to the grain boundaries or it is dominant within the grains as well. In the former case, simplifications in the models save computational time in the simulations.

2. In dynamics, the evolution of the strain field is relevant. E.g. an initially intergranular deformation can turn into diffuse localization that occurs within the grains as well. In that case, a strain field must be obtained at each time step, and this function can be called for each such instance.

## Examples

The following figure was created by this function with `visualize` set to *True* and `band_width` chosen to be 3.

# TWENTYFIVE

# UTILITIES

This module provides *general* utility functions used by the **grains** package. The specific helper functions reside in the proper module. For example, a function that works on a general list goes here, but a computational geometry algorithm goes to the **geometry** module. The functions in the **utils** module can be interesting for other projects too, partly because of their general scope, and partly because of the few dependencies.

## 25.1 Functions

| | |
|---|---|
| *duplicates* | Set of duplicate elements in a sequence. |
| *toggle* | Return True for False values and False for True values in a list. |
| *index_list* | Index a list by another list. |
| *flatten_list* | Merge a list of lists to a single list. |
| *argsorted* | Return the indices that would sort a list or a tuple. |
| *map_inplace* | Apply a function to each member of an iterable in-place. |
| *non_unique* | Finds indices of non-unique elements in a 1D or 2D ndarray. |
| *parse_kwargs* | Compares keyword arguments with defaults. |
| *compress* | Creates a zip archive from a single file. |
| *decompress* | Decompresses the contents of a zip archive into the current directory. |
| *decompress_inmemory* | Decompresses the contents of a zip archive into a dictionary. |
| *neighborhood* | Neighboring points to a grid point. |

### 25.1.1 grains.utils.duplicates

grains.utils.**duplicates**(*sequence*)

 Set of duplicate elements in a sequence.

> **Parameters sequence** (*sequence types (list, tuple, string, etc.)*) – Sequence possibly containing repeating elements.
>
> **Returns** *set* – Set of unique values.

**Notes**

Copied from https://stackoverflow.com/a/9836685/4892892

**Examples**

Note that the order of the elements in the resulting set does not matter.

```
>>> a = [1, 2, 3, 2, 1, 5, 6, 5, 5, 5]  # list
>>> duplicates(a)
{1, 2, 5}
>>> a = (1, 1, 0, -1, -1, 0)  # tuple
>>> duplicates(a)
{0, 1, -1}
>>> a = 'abbcdkc'  # string
>>> duplicates(a)
{'c', 'b'}
```

## 25.1.2 grains.utils.toggle

grains.utils.**toggle**(*lst*)

Return True for False values and False for True values in a list.

> **Parameters  lst** (*list*) – An arbitrary list, possibly containing other lists.
>
> **Returns**  *list* – Element-wise logical not operator applied on the input list.

**Notes**

Solution taken from https://stackoverflow.com/a/51122372/4892892.

**Examples**

```
>>> toggle([True, False])
[False, True]
>>> toggle(['h', 0, 2.3, -2, 5, []])
[False, True, False, False, False, True]
```

## 25.1.3 grains.utils.index_list

grains.utils.**index_list**(*lst*, *indices*)

Index a list by another list.

> **Parameters**
>
> > • **lst** (*list*) – List to be indexed.
> >
> > • **indices** (*list*) – Indices of the original list that will form the new list.
>
> **Returns**  *list* – Members of *lst*, selected by *indices*.

### Examples

```
>>> index_list(['c', ['nested', 'list'], 13], [1, 2])
[['nested', 'list'], 13]
```

## 25.1.4 grains.utils.flatten_list

grains.utils.**flatten_list**(*nested_list*)

    Merge a list of lists to a single list.

> **Parameters**   **nested_list** (*list*) – List containing other lists.
>
> **Returns**   *list* – Flattened list.

### Notes

- Only a single level (i.e. list of lists) is handled, see the second example.

- Several methods, such as list comprehension, monoid and loops, are proposed in https://stackoverflow.com/questions/952914/how-to-make-a-flat-list-out-of-list-of-lists. Here, the list comprehension approach is used.

### Examples

```
>>> nested_list = [['some'], ['items']]
>>> flatten_list(nested_list)
['some', 'items']
>>> multiply_nested_list = [[['item'], 'within', 'item']]
>>> flatten_list(multiply_nested_list)
[['item'], 'within', 'item']
```

## 25.1.5 grains.utils.argsorted

grains.utils.**argsorted**(*sequence*, *reverse=False*)

    Return the indices that would sort a list or a tuple.

    Implementation is taken from https://stackoverflow.com/a/6979121/4892892.

> **Parameters**
>
> - **sequence** (*list, tuple*) – Input sequence in which the sorted indices to be found.
>
> - **reverse** (*bool*) – If set to True, then the elements are sorted as if each comparison was reversed.
>
> **Returns**   *list* – List of indices that would sort the input list/tuple.

**See also:**

sorted(), numpy.argsort()

**Examples**

```
>>> argsorted([2, 1.1, 1.1])
[1, 2, 0]
>>> argsorted([2, 1.1, 1.1], reverse=True)
[0, 1, 2]
>>> argsorted(())
[]
```

## 25.1.6 grains.utils.map_inplace

grains.utils.**map_inplace**(*function*, *__iterable*)
    Apply a function to each member of an iterable in-place.

> **Parameters**
>
> > * **function** (*function object*) – Function to be applied to the entries of the iterable.
> >
> > * **__iterable** (*iterable*) – Iterable.

> **Notes**

Comprehensions or functional tools work on iterators, thereby not modifying the original container (https://stackoverflow.com/a/4148523/4892892). For in-place modification, the conventional for loop approach is used (https://stackoverflow.com/a/4148525/4892892).

> **Examples**

```
>>> lst = ['2', 2]; func = lambda x: x*2
>>> map_inplace(func, lst); lst
['22', 4]
>>> lifespan = {'cat': 15, 'dog': 12}; die_early = lambda x: x/2
>>> map_inplace(die_early, lifespan); lifespan
{'cat': 7.5, 'dog': 6.0}
```

## 25.1.7 grains.utils.non_unique

grains.utils.**non_unique**(*array*, *axis=None*)
    Finds indices of non-unique elements in a 1D or 2D ndarray.

> **Parameters**
>
> > * **array** (*ndarray*) – Array in which the non-unique elements are searched.
> >
> > * **axis** (*{None, 0, 1}, optional*) – The axis to operate on. If None, *array* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.
>
> **Returns**
>
> > * **nonunique_values** (*list*) – Unique (individual, row or column) entries.

- **nonunique_indices** (*list*) – Each element of the list corresponds to non-unique elements, whose indices are given in a 1D numpy array.

**Examples**

In a 1D array, the repeated values and their indices are found by

```
>>> val, idx = non_unique(np.array([1, -1, 0, -1, 2, 5, 0, -1]))
>>> val
[-1, 0]
>>> idx
[array([1, 3, 7]), array([2, 6])]
```

In the matrix below, we can see that rows 0 and 2 are identical, as well as rows 1 and 4.

```
>>> val, idx = non_unique(np.array([[1, 3], [2, 4], [1, 3], [-1, 0], [2, 4]]),␣
↪axis=0)
>>> val
[array([1, 3]), array([2, 4])]
>>> idx
[array([0, 2]), array([1, 4])]
```

By transposing the matrix above, the same holds for the columns.

```
>>> val, idx = non_unique(np.array([[1, 2, 1, -1, 2], [3, 4, 3, 0, 4]]), axis=1)
>>> val
[array([1, 3]), array([2, 4])]
>>> idx
[array([0, 2]), array([1, 4])]
```

If the dimensions along which to find the duplicates are not given, the input is flattened and the indexing happens in C-order (row-wise).

```
>>> val, idx = non_unique(np.array([[1, 2, 1, -1, 2], [3, 4, 3, 0, 4]]))
>>> val
[1, 2, 3, 4]
>>> idx
[array([0, 2]), array([1, 4]), array([5, 7]), array([6, 9])]
```

## 25.1.8 grains.utils.parse_kwargs

grains.utils.**parse_kwargs**(*kwargs*, *defaults*)
    Compares keyword arguments with defaults.

    Allows processing keyword arguments supplied to a function by the user by comparing them with an admissible set of options defined by the developer. There are three cases:

    1. The keyword given by the user is present in the set the developer provides. Then the value belonging to the keyword overrides the default value.

    2. The keyword given by the user is *not* present in the set the developer provides. In this case, the unrecognized keyword, along with its value, is saved separately.

    3. The keyword existing in the set the developer provides is not given by the user. Then the default value is used.

    **Parameters**

- **kwargs** (*dict*) – Keyword arguments (parameter-value pairs) passed to a function.
- **defaults** (*dict*) – Default parameter-value pairs.

  **Returns**

- **parsed** (*dict*) – Dictionary with the same keys as `defaults`, the parsed parameter-value pairs.
- **unknown** (*dict*) – Dictionary containing the parameter-value pairs not present in `defaults`.

### Notes

The default values, given in the input dictionary `defaults`, are never overwritten.

### Examples

```
>>> default_options = {'opt1': 1, 'opt2': 'string', 'opt3': [-1, 0]}
>>> user_options = {'opt3': [2, 3, -1], 'opt2': 'string', 'opt4': -2.14}
>>> parsed_options, unknown_options = parse_kwargs(user_options, default_options)
>>> parsed_options
{'opt1': 1, 'opt2': 'string', 'opt3': [2, 3, -1]}
>>> unknown_options
{'opt4': -2.14}
```

## 25.1.9 grains.utils.compress

grains.utils.**compress** (*filename*, *level=9*)

   Creates a zip archive from a single file.

   **Parameters**

- **filename** (*str*) – Name of the file to be compressed.
- **level** (*int, optional*) – Level of compression. Integers 0 through 9 are accepted. The default is 9.

   **Returns** *None.*

   See also:

   `zipfile.ZipFile`

## 25.1.10 grains.utils.decompress

grains.utils.**decompress** (*filename*, *path=None*)

   Decompresses the contents of a zip archive into the current directory.

   **Parameters**

- **filename** (*str*) – Name of the zip archive.
- **path** (*str, optional*) – Directory to extract to. The default is the directory the function is called from.

**See also:**

`zipfile.ZipFile`

### 25.1.11 grains.utils.decompress_inmemory

`grains.utils.`**`decompress_inmemory`** (*filename*)

Decompresses the contents of a zip archive into a dictionary.

> **Parameters filename** (*str*) – Name of the zip archive.

> **Returns data** (*dict*) – The keys of the dictionary are the compressed file names (without extension), the corresponding values are their contents.

**See also:**

`zipfile.ZipFile`

### 25.1.12 grains.utils.neighborhood

`grains.utils.`**`neighborhood`** (*center*, *radius*, *norm*, *method='ball'*, *bounds=None*)

Neighboring points to a grid point.

Given a point in a subspace of $\mathbb{Z}^n$, the neighboring points are determined based on the specified rules.

---

**Todo:** Currently, the neighbors are deterministically but not systematically ordered. Apart from testing purposes, this does not seem to be a big issue. Still, a logical ordering is desirable. E.g. ordering in increasing coordinate values, first in the first dimension and lastly in the last dimension.

---

> **Parameters**

> - **center** (*tuple of int*) – Point $x_0$ around which the neighborhood is searched. It must be an n-tuple of integers, where $n$ is the spatial dimension.

> - **radius** (*int, positive*) – Radius of the ball or sphere in which the neighbors are searched. If the radius is 1, the immediate neighbors are returned.

> - **norm** (*{1, inf}*) – Type of the vector norm $\|x - x_0\|$, where $x$ is a point whose distance is searched from the center $x_0$.

> | Type | Name | Definition |
> |------|------|------------|
> | 1 | 1-norm | $\|x\|_1 = \sum_{i=1}^{n} |x_i|$ |
> | numpy.inf | maximum norm | $\|x\|_\infty = \max\{|x_1|, \ldots, |x_n|\}$ |

> where inf means numpy's np.inf object.

> - **method** (*{'ball', 'sphere'}, optional*) – Specifies the criterion of how to decide whether a point $x$ is in the neighborhood of $x_0$. The default is 'ball'.

> For 'ball':

$$\|x - x_0\| \leq r$$

> For 'sphere':

$$\|x - x_0\| = r$$

where $r$ is the radius passed as the `radius` parameter and the type of the norm is taken based on the `norm` input parameter.

- **bounds** (*list of tuple, optional*) – Restricts the neighbors within a box. The dimensions of the n-dimensional box are given as a list of 2-tuples: [(x_1_min, x_1_max), ... , (x_n_min, x_n_max)]. The default value is an unbounded box in all dimensions. Use np.inf to indicate unbounded values.

**Returns  neighbors** (*tuple of ndarray*) – Tuple of length n, each entry being a 1D numpy array: the integer indices of the points that are in the neighborhood of `center`.

### Notes

1. The von Neumann neighborhood with range $r$ is a special case when `radius=r`, `norm=1` and `method='ball'`.

2. The Moore neighborhood with range $r$ is a special case when `radius=r`, `norm=np.inf` and `method='ball'`.

### Examples

Find the Moore neighborhood with range 2 the point (1) on the half-line [0, inf).

```
>>> neighborhood((1,), 2, np.inf, bounds=[(0, np.inf)])
(array([0, 1, 2, 3]),)
```

Find the von Neumann neighborhood with range 2 around the point (2, 2), restricted on the domain [0, 4] x [0, 3].

```
>>> neighborhood((2, 2), 2, 1, bounds=[(0, 4), (0, 3)])
(array([2, 1, 2, 3, 0, 1, 2, 3, 4, 1, 2, 3]), array([0, 1, 1, 1, 2, 2, 2, 2, 2, 3,
↪ 3, 3]))
```

Find the Moore neighborhood with range 1 around the point (0, -4) such that the neighbors lie on the half-plane [0, 2] x (-inf, -4].

```
>>> neighborhood((0, -4), 1, np.inf, bounds=[(0, 2), (-np.inf, -4)])
(array([0, 1, 0, 1]), array([-5, -5, -4, -4]))
```

Find the sphere of radius 2, measured in the 1-norm, around the point (-1, 0, 3), within the half-space {(x,y,z) in Z^3 | y>=0}.

```
>>> neighborhood((-1, 0, 3), 2, 1, 'sphere', [(-np.inf, np.inf), (0, np.inf), (-
↪np.inf, np.inf)])
(array([-3, -2, -2, -1, -1,  0,  0,  1, -2, -1, -1,  0, -1]),
 array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2]),
 array([3, 2, 4, 1, 5, 2, 4, 3, 3, 2, 4, 3, 3]))
```

`grains.utils.`**`argsorted`**(*sequence*, *reverse=False*)

Return the indices that would sort a list or a tuple.

Implementation is taken from https://stackoverflow.com/a/6979121/4892892.

**Parameters**

- **sequence** (*list, tuple*) – Input sequence in which the sorted indices to be found.

- **reverse** (*bool*) – If set to True, then the elements are sorted as if each comparison was reversed.

    **Returns** *list* – List of indices that would sort the input list/tuple.

    See also:

    `sorted()`, `numpy.argsort()`

    **Examples**

    ```
    >>> argsorted([2, 1.1, 1.1])
    [1, 2, 0]
    >>> argsorted([2, 1.1, 1.1], reverse=True)
    [0, 1, 2]
    >>> argsorted(())
    []
    ```

`grains.utils.`**`compress`** (*filename*, *level=9*)
    Creates a zip archive from a single file.

    **Parameters**

    - **filename** (*str*) – Name of the file to be compressed.

    - **level** (*int, optional*) – Level of compression. Integers 0 through 9 are accepted. The default is 9.

    **Returns** *None.*

    See also:

    `zipfile.ZipFile`

`grains.utils.`**`decompress`** (*filename*, *path=None*)
    Decompresses the contents of a zip archive into the current directory.

    **Parameters**

    - **filename** (*str*) – Name of the zip archive.

    - **path** (*str, optional*) – Directory to extract to. The default is the directory the function is called from.

    See also:

    `zipfile.ZipFile`

`grains.utils.`**`decompress_inmemory`** (*filename*)
    Decompresses the contents of a zip archive into a dictionary.

    **Parameters** **filename** (*str*) – Name of the zip archive.

    **Returns** **data** (*dict*) – The keys of the dictionary are the compressed file names (without extension), the corresponding values are their contents.

    See also:

    `zipfile.ZipFile`

`grains.utils.`**`duplicates`** (*sequence*)
    Set of duplicate elements in a sequence.

> **Parameters sequence** (*sequence types (list, tuple, string, etc.)*) – Sequence possibly containing repeating elements.
>
> **Returns** *set* – Set of unique values.

#### Notes

Copied from https://stackoverflow.com/a/9836685/4892892

#### Examples

Note that the order of the elements in the resulting set does not matter.

```
>>> a = [1, 2, 3, 2, 1, 5, 6, 5, 5, 5]  # list
>>> duplicates(a)
{1, 2, 5}
>>> a = (1, 1, 0, -1, -1, 0)  # tuple
>>> duplicates(a)
{0, 1, -1}
>>> a = 'abbcdkc'  # string
>>> duplicates(a)
{'c', 'b'}
```

grains.utils.**flatten_list**(*nested_list*)

Merge a list of lists to a single list.

> **Parameters nested_list** (*list*) – List containing other lists.
>
> **Returns** *list* – Flattened list.

#### Notes

- Only a single level (i.e. list of lists) is handled, see the second example.
- Several methods, such as list comprehension, monoid and loops, are proposed in https://stackoverflow.com/questions/952914/how-to-make-a-flat-list-out-of-list-of-lists. Here, the list comprehension approach is used.

#### Examples

```
>>> nested_list = [['some'], ['items']]
>>> flatten_list(nested_list)
['some', 'items']
>>> multiply_nested_list = [[['item'], 'within', 'item']]
>>> flatten_list(multiply_nested_list)
[['item'], 'within', 'item']
```

grains.utils.**index_list**(*lst*, *indices*)

Index a list by another list.

> **Parameters**
>
> - **lst** (*list*) – List to be indexed.
> - **indices** (*list*) – Indices of the original list that will form the new list.

**Returns** *list* – Members of *lst*, selected by *indices*.

**Examples**

```
>>> index_list(['c', ['nested', 'list'], 13], [1, 2])
[['nested', 'list'], 13]
```

grains.utils.**map_inplace**(*function*, *__iterable*)

Apply a function to each member of an iterable in-place.

> **Parameters**
>
> > - **function** (*function object*) – Function to be applied to the entries of the iterable.
> >
> > - **__iterable** (*iterable*) – Iterable.
>
> **Notes**
>
> Comprehensions or functional tools work on iterators, thereby not modifying the original container (https://stackoverflow.com/a/4148523/4892892). For in-place modification, the conventional for loop approach is used (https://stackoverflow.com/a/4148525/4892892).
>
> **Examples**
>
> ```
> >>> lst = ['2', 2]; func = lambda x: x*2
> >>> map_inplace(func, lst); lst
> ['22', 4]
> >>> lifespan = {'cat': 15, 'dog': 12}; die_early = lambda x: x/2
> >>> map_inplace(die_early, lifespan); lifespan
> {'cat': 7.5, 'dog': 6.0}
> ```

grains.utils.**neighborhood**(*center*, *radius*, *norm*, *method='ball'*, *bounds=None*)

Neighboring points to a grid point.

Given a point in a subspace of $\mathbb{Z}^n$, the neighboring points are determined based on the specified rules.

---

**Todo:** Currently, the neighbors are deterministically but not systematically ordered. Apart from testing purposes, this does not seem to be a big issue. Still, a logical ordering is desirable. E.g. ordering in increasing coordinate values, first in the first dimension and lastly in the last dimension.

---

> **Parameters**
>
> > - **center** (*tuple of int*) – Point $x_0$ around which the neighborhood is searched. It must be an n-tuple of integers, where $n$ is the spatial dimension.
> >
> > - **radius** (*int, positive*) – Radius of the ball or sphere in which the neighbors are searched. If the radius is 1, the immediate neighbors are returned.
> >
> > - **norm** (*{1, inf}*) – Type of the vector norm $\|x - x_0\|$, where $x$ is a point whose distance is searched from the center $x_0$.

| Type | Name | Definition |
|------|------|------------|
| 1 | 1-norm | $\|x\|_1 = \sum_{i=1}^{n} \|x_i\|$ |
| numpy.inf | maximum norm | $\|x\|_\infty = \max\{\|x_1\|, \ldots, \|x_n\|\}$ |

where inf means numpy's np.inf object.

- **method** (*{'ball', 'sphere'}, optional*) – Specifies the criterion of how to decide whether a point $x$ is in the neighborhood of $x_0$. The default is 'ball'.

  For 'ball':

  $$\|x - x_0\| \leq r$$

  For 'sphere':

  $$\|x - x_0\| = r$$

  where $r$ is the radius passed as the `radius` parameter and the type of the norm is taken based on the `norm` input parameter.

- **bounds** (*list of tuple, optional*) – Restricts the neighbors within a box. The dimensions of the n-dimensional box are given as a list of 2-tuples: [(x_1_min, x_1_max), ... , (x_n_min, x_n_max)]. The default value is an unbounded box in all dimensions. Use np.inf to indicate unbounded values.

**Returns neighbors** (*tuple of ndarray*) – Tuple of length n, each entry being a 1D numpy array: the integer indices of the points that are in the neighborhood of `center`.

### Notes

1. The von Neumann neighborhood with range $r$ is a special case when `radius=r`, `norm=1` and `method='ball'`.

2. The Moore neighborhood with range $r$ is a special case when `radius=r`, `norm=np.inf` and `method='ball'`.

### Examples

Find the Moore neighborhood with range 2 the point (1) on the half-line [0, inf).

```
>>> neighborhood((1,), 2, np.inf, bounds=[(0, np.inf)])
(array([0, 1, 2, 3]),)
```

Find the von Neumann neighborhood with range 2 around the point (2, 2), restricted on the domain [0, 4] x [0, 3].

```
>>> neighborhood((2, 2), 2, 1, bounds=[(0, 4), (0, 3)])
(array([2, 1, 2, 3, 0, 1, 2, 3, 4, 1, 2, 3]), array([0, 1, 1, 1, 2, 2, 2, 2, 2, 3,
↪ 3, 3]))
```

Find the Moore neighborhood with range 1 around the point (0, -4) such that the neighbors lie on the half-plane [0, 2] x (-inf, -4].

```
>>> neighborhood((0, -4), 1, np.inf, bounds=[(0, 2), (-np.inf, -4)])
(array([0, 1, 0, 1]), array([-5, -5, -4, -4]))
```

Find the sphere of radius 2, measured in the 1-norm, around the point (-1, 0, 3), within the half-space {(x,y,z) in Z^3 | y>=0}.

```
>>> neighborhood((-1, 0, 3), 2, 1, 'sphere', [(-np.inf, np.inf), (0, np.inf), (-
→np.inf, np.inf)])
(array([-3, -2, -2, -1, -1,  0,  0,  1, -2, -1, -1,  0, -1]),
 array([0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2]),
 array([3, 2, 4, 1, 5, 2, 4, 3, 3, 2, 4, 3, 3]))
```

grains.utils.**non_unique**(*array*, *axis=None*)

>    Finds indices of non-unique elements in a 1D or 2D ndarray.

>    **Parameters**

>    - **array** (*ndarray*) – Array in which the non-unique elements are searched.
>    - **axis** (*{None, 0, 1}, optional*) – The axis to operate on. If None, *array* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.

>    **Returns**

>    - **nonunique_values** (*list*) – Unique (individual, row or column) entries.
>    - **nonunique_indices** (*list*) – Each element of the list corresponds to non-unique elements, whose indices are given in a 1D numpy array.

**Examples**

In a 1D array, the repeated values and their indices are found by

```
>>> val, idx = non_unique(np.array([1, -1, 0, -1, 2, 5, 0, -1]))
>>> val
[-1, 0]
>>> idx
[array([1, 3, 7]), array([2, 6])]
```

In the matrix below, we can see that rows 0 and 2 are identical, as well as rows 1 and 4.

```
>>> val, idx = non_unique(np.array([[1, 3], [2, 4], [1, 3], [-1, 0], [2, 4]]),
→axis=0)
>>> val
[array([1, 3]), array([2, 4])]
>>> idx
[array([0, 2]), array([1, 4])]
```

By transposing the matrix above, the same holds for the columns.

```
>>> val, idx = non_unique(np.array([[1, 2, 1, -1, 2], [3, 4, 3, 0, 4]]), axis=1)
>>> val
[array([1, 3]), array([2, 4])]
>>> idx
[array([0, 2]), array([1, 4])]
```

If the dimensions along which to find the duplicates are not given, the input is flattened and the indexing happens in C-order (row-wise).

```
>>> val, idx = non_unique(np.array([[1, 2, 1, -1, 2], [3, 4, 3, 0, 4]]))
>>> val
[1, 2, 3, 4]
>>> idx
[array([0, 2]), array([1, 4]), array([5, 7]), array([6, 9])]
```

grains.utils.**parse_kwargs**(*kwargs*, *defaults*)

Compares keyword arguments with defaults.

Allows processing keyword arguments supplied to a function by the user by comparing them with an admissible set of options defined by the developer. There are three cases:

1. The keyword given by the user is present in the set the developer provides. Then the value belonging to the keyword overrides the default value.

2. The keyword given by the user is *not* present in the set the developer provides. In this case, the unrecognized keyword, along with its value, is saved separately.

3. The keyword existing in the set the developer provides is not given by the user. Then the default value is used.

> **Parameters**
>
> - **kwargs** (*dict*) – Keyword arguments (parameter-value pairs) passed to a function.
> - **defaults** (*dict*) – Default parameter-value pairs.
>
> **Returns**
>
> - **parsed** (*dict*) – Dictionary with the same keys as `defaults`, the parsed parameter-value pairs.
> - **unknown** (*dict*) – Dictionary containing the parameter-value pairs not present in `defaults`.

### Notes

The default values, given in the input dictionary `defaults`, are never overwritten.

### Examples

```
>>> default_options = {'opt1': 1, 'opt2': 'string', 'opt3': [-1, 0]}
>>> user_options = {'opt3': [2, 3, -1], 'opt2': 'string', 'opt4': -2.14}
>>> parsed_options, unknown_options = parse_kwargs(user_options, default_options)
>>> parsed_options
{'opt1': 1, 'opt2': 'string', 'opt3': [2, 3, -1]}
>>> unknown_options
{'opt4': -2.14}
```

grains.utils.**toggle**(*lst*)

Return True for False values and False for True values in a list.

> **Parameters** **lst** (*list*) – An arbitrary list, possibly containing other lists.
>
> **Returns** *list* – Element-wise logical not operator applied on the input list.

**Notes**

Solution taken from https://stackoverflow.com/a/51122372/4892892.

**Examples**

```
>>> toggle([True, False])
[False, True]
>>> toggle(['h', 0, 2.3, -2, 5, []])
[False, True, False, False, False, True]
```

# PROFILING

This module implements profiling facilities so that user code can be tested for speed.

## 26.1 Functions

| | |
|---|---|
| [*profile*](#)([output_format, output_filename, . . . ]) | Profiles a block of code with Pyinstrument. |

### 26.1.1 grains.profiling.profile

grains.profiling.**profile**(*output_format='html'*, *output_filename=None*, *html_open=False*)
    Profiles a block of code with Pyinstrument.

    Intended to be used in a *with* statement.

> **Parameters**
> - **output_format** (*{'html', 'text'}, optional*) – Shows the result either as text or in an HTML file. If output_format = 'html', the file is saved according to the output_filename parameter. The default is 'html'.
> - **output_filename** (*str, optional*) – Only taken into account if output_format = 'html'. If not given (default), the html output is saved to the same directory the caller resides. The name of the html file is the same as that of the caller.
> - **html_open** (*bool, optional*) – Only taken into account if output_format = 'html'. If True, the generated HTML file is opened in the default browser. The default is False.

> **Notes**

> In the implementation, we move two levels up in the stack frame, one for exiting the context manager and one for exiting this generator. This assumes that [*profile()*](#) was called as a context manager. As a good practice, provide the output_filename input argument.

### Examples

Measure the time needed to generate 1 million uniformly distributed random numbers.

```
>>> import random
>>> with profile('html') as p:
...     for _ in range(1000000):
...         rand_num = random.uniform(1, 2.2)
```

grains.profiling.**profile**(*output_format='html'*, *output_filename=None*, *html_open=False*)

Profiles a block of code with Pyinstrument.

Intended to be used in a *with* statement.

> **Parameters**
>
> - **output_format** (*{'html', 'text'}, optional*) – Shows the result either as text or in an HTML file. If `output_format = 'html'`, the file is saved according to the `output_filename` parameter. The default is 'html'.
>
> - **output_filename** (*str, optional*) – Only taken into account if `output_format = 'html'`. If not given (default), the html output is saved to the same directory the caller resides. The name of the html file is the same as that of the caller.
>
> - **html_open** (*bool, optional*) – Only taken into account if `output_format = 'html'`. If True, the generated HTML file is opened in the default browser. The default is False.

### Notes

In the implementation, we move two levels up in the stack frame, one for exiting the context manager and one for exiting this generator. This assumes that *profile()* was called as a context manager. As a good practice, provide the `output_filename` input argument.

### Examples

Measure the time needed to generate 1 million uniformly distributed random numbers.

```
>>> import random
>>> with profile('html') as p:
...     for _ in range(1000000):
...         rand_num = random.uniform(1, 2.2)
```

# TWENTYSEVEN

# INDEX

## 27.1 Index

# CHANGELOG

All notable changes to this project will be documented in this file. The format is based on Keep a Changelog, and this project **does not** adhere to Semantic Versioning. We refer to GitHub issues by their numbers. If there is no issue associated to the change, the commit hashes implementing the change are linked.

## 28.1 Unreleased

## 28.2 1.1.0 - 2021-03-04

### 28.2.1 Added

- Grains embedded into other grains are now identified in the splinegon creation algorithm. 4cef7ec, 6c4ad0a

- Computation of the infinitesimal and Green-Lagrange strain tensors for DIC displacement field. 7eaa0b5, 9e5466e

- Computation of the equivalent von Mises strain. 2b9b0be

- Characterization of the strain localization (intergranular/intragranular) in the microstructure. a74f954, 13576f7

- The README file shows how to cite our paper. #30

- Added a new segmented microstructure (6b539ba) and the corresponding DIC measurements (3c845e7).

### 28.2.2 Deprecated

- The content of `dic.DIC.plot_strain` will be replaced with that of `dic.plot_strain` in version 1.3.0. The latter function will be removed. b8633c2

## 28.3 1.0.1 - 2020-11-19

### 28.3.1 Added

- Document on describing the versioning scheme we will follow. e74474c

- Added a changelog to the project. b48821b, f6ca9c1

### 28.3.2 Removed

- The documentation no longer shows the recent git commits. 1fd3a50

## 28.4 1.0.0 - 2020-11-16

This is the initial release of *CristalX*.

# LICENSE

**GNU LESSER GENERAL PUBLIC LICENSE**  Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates

the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser

General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License,

other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided

by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an

Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the

Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the

object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License

without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a

facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from

a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that,

taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the

Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

> a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

> b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions

of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the

Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide

whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

# PYTHON MODULE INDEX

## g