CristalX: Facilitating simulations for experimentally obtained grain-based microstructures

Zoltan Csati, Jean-Francois Witz, Vincent Magnier, Ahmed El Bartali, Nathalie Limodin, Denis Najjar

CristalX: Facilitating simulations for experimentally obtained grain-based microstructures

Zoltan Csati[a,*], Jean-François Witz[a], Vincent Magnier[a], Ahmed El Bartali[a], Nathalie Limodin[a], Denis Najjar[a]

[a] *Univ. Lille, CNRS, Centrale Lille, UMR 9013 - LaMcube - Laboratoire de Mécanique, Multiphysique, Multi-échelle, Lille, F-59000, France*

Abstract

Polycrystalline microstructures occur in nature and also arise in industrial processes. Performing finite element computations on such microstructures is relevant in multiple engineering fields. The present paper describes a highly automated, possibly human-assisted, approach to create high-quality finite element meshes for polycrystalline microstructures. The input is an image, showing grains to be identified. First, the individual grains are obtained by performing several image segmentation techniques. The program allows analyzing the relevant properties of a grain, such as its diameter. The segmented image is then turned to a CAD geometry. Different parametrizations of this geometry lead to different grain smoothness. The advantage of our approach is that existing, robust meshing software can be used to create a high-quality mesh on the grain assembly in a completely automatic way. The developed methodology is illustrated on a real-world example.

Key words: meshing, microstructure, segmentation

*Corresponding author
Email address: zoltan-c@keemail.me (Zoltan Csati)

Current code version	1.0.1
Permanent link to code/repository used for this code version	`https://github.com/CsatiZoltan/CristalX`
Legal Code License	LGPL-3.0
Code versioning system used	git
Software code languages, tools, and services used	Python $\geq$ 3.6; Binder; Read the Docs; Better Code Hub
Compilation requirements, operating environments & dependencies	Linux/Windows; requirements provided in the `environment.yml` file in the repository
Link to developer documentation	`https://cristalx.readthedocs.io/`
Support email for questions	zoltan-c@keemail.me

Code metadata

1. Introduction

Grain-based microstructures occur in nature (e.g. crystalline rocks) and also play a role in industrial processes such as forging [1] or developing cermets [2]. The effect of the grain size and shape, and the distribution of the constituents at the micro scale determines the behaviour of the material at the macro scale[1] [3]. It is therefore of interest to characterize the microstructure. One way to do it is to synthetically generate many of them with some relevant characteristics drawn from statistical distributions and use statistical methods to compute the average response to given loading conditions [4]. The advantage of computational microstructure generation is that one can perform computational experiments by changing the parameters. When a new material is under development, it is highly useful to virtually experiment with the microstructure without having to produce samples and conduct real experiments on them. It is also a faster and cheaper method than having to perform time-consuming and often costly measurements. For grain generation, see the recent paper [5]. The open-source tool *Neper* [6] also has a module for generating polycrystalline microstructures. This approach works well when the microstructure can be properly captured by the statistical parameters. However, when this is not the case or if a specific real microstructure is of interest, measurements are necessary. The outcomes of measurements are images in which the grains are to be identified. This is a task for image segmentation.

In order to locate objects and boundaries, image segmentation methods partition an image into regions such that a distinct label is associated to

[1] For the sake of brevity, we do not make a distinction between micro scale and meso scale in this paper. The terms *micro scale* and *macro scale* are used to accentuate the difference in the scale in which phenomena are investigated. The notion micro scale will be used for the scale where the grain shapes can be identified.

each region (pixel group). Superpixels are groups of pixels that belong together based on color, spatial distance or other properties [7]. Superpixel segmentation subdivides an image into superpixels on which other image processing algorithms can operate efficiently. For a comparison of superpixel segmentation methods, see [8]. As superpixels over-segment the image, other segmentation techniques are necessary to obtain a properly segmented image, in which the individual grains can be distinguished. One such method is the watershed segmentation. Watershed segmentation is a common use case of the watershed transformation, also simply called watershed. Introduced in [9], the term watershed was coined after the geological definition, i.e. the set of barriers that separate the catchment basins of a topographic surface. A grayscale image can be viewed as a topographic surface where the pixel intensities correspond to hills and valleys. Intuitively, the surface is flooded starting from the valleys (local minima) with different colors of water. When two different colors would merge, a barrier is inserted. The process continues until the whole surface is under water. Since [9] many other definitions of the watershed were put forward and several algorithms were proposed to compute them. When directly applied on a grayscale image, the result of the watershed segmentation is 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.

Homogenization methods provide the theoretical background on the transition from the micro scale to the macro scale. In the presence of complex microstructural features or complicated physics, the so-called computational homogenization is prevalent [10]. In computational homogenization, the governing equations are often solved by the finite element method (FEM), which requires a mesh on the domain. Image-based approaches directly create a mesh on an image, without the need for an intermediate geometry construction step. Most methods work on an already segmented image. The simplest approach is to allocate one mesh cell for each pixel (or voxel in 3D). This not only results in prohibitively large meshes but also inaccuracies in the subsequent simulation due to the jagged surface approximation. Therefore, unstructured meshes are often preferred, in which the image features are taken into account when generating the mesh. The software *OOF2* [11] has several built-in local mesh modification algorithms to create a mesh on a segmented image. That approach was extended for tetrahedral meshing in 3D in *OOF3D* [12]. On the other hand, [13] came up with a penalty method that does not even require a segmented image. Additionally, the meshed image

can be perceived as a lower-resolution representation of the original image. Hence, their method is not only suitable for meshing an image but can also serve as an initial segmentation, similarly to the superpixel segmentation algorithms. In this category, we mention *Neper* [6], a tool for generating and meshing polycrystals, modeled as a Voronoi tessellation. The strategies above are usually complex because the unstructured mesh must satisfy two criteria at the same time. It must consist of good quality cells and the cells should be as homogeneous as possible, i.e. a cell should ideally belong to a single labeled region. The X-FEM coupled with the level set method circumvents this requirement [14]. Here, the level set is constructed on the high-resolution image. However, a simple structured or block-structured mesh can be created independently of the boundaries of the segmented image. This way, the boundaries cut the mesh cells and the level set values are projected onto the mesh. By coupling the X-FEM with the level set method, the quality of the function approximation (determined by the mesh density) is decoupled from the resolution of the geometry (determined by the number of pixels/voxels).

The quality of the mesh influences both the interpolation error and the conditioning of the discretized problem [15]. In case of nonlinear problems (e.g. contact, large deformation elasticity, plasticity), a high quality mesh is crucial to obtain an accurate solution, or even to have a convergent solution at a given load increment. While generating adaptive meshes directly based on an image is an ongoing research effort, image-based methods usually concentrate on creating meshes on the grains. However, in many cases interesting phenomena, such as strain localization, happen along the grain boundaries (interfaces). It is therefore useful to be able to extract the interfaces and control the mesh generation e.g. by constructing graded meshes near the interfaces. Mesh refinement is also necessary to conduct convergence studies. Another problem we faced was mesh repairing. Since the mesh quality obtained by *OOF2* was not good enough, we considered improving the mesh with third-party tools. However, the new mesh obtained by those tools does not respect the original microstructure, which can be detrimental for small grains consisting of only a few cells. What is more, the new mesh has no information about which cell belongs to which grain, which information is necessary if different material properties are associated to the different grains. The challenges above lead us to represent the grains as an *exact* geometry, rather than an assembly of cells, which is an inherently discrete approach. We note that after extracting the interfaces, non-conforming (and possibly block-structured) meshes could be used, as in X-FEM. That choice eases the mesh generation but at the same time requires specialized solvers and custom preconditioners. Therefore, in this paper, traditional conforming mesh generation is applied, which has the advantage that robust open-source

or proprietary software can be utilized for that purpose.

The paper is structured as follows. Section 2 is the main part of the article, containing the reusable components of our program. Section 2.1 describes the philosophy behind the design choices. These were the most relevant requirements we targeted from the very beginning, and we stuck to them as the project evolved. Sections 2.2–2.3 detail the two main ingredients of obtaining a geometrical representation of a microstructure. These two steps comprise the core of our software. Additional capabilities of the program are described in Section 2.4. Section 2 ends with the means of contributing to this project (Section 2.5). After having discussed the fundamentals of the program in Section 2, its usage is demonstrated in Section 3 by a practical example our group is currently working on. That section shows how to use the available building blocks and provides additional Python modules for extending the core in order to meet the demands. The impact of our software on the material science community is highlighted in Section 4. Some perspectives for future extension are given in Section 5.

2. Software description

Our project, called `CristalX`, is written in pure Python[2], a widely-used free and open-source general purpose language with a rich scientific ecosystem [16]. The source code of the software is available on `https://github.com/CsatiZoltan/CristalX` under the LGPL-3.0 license. Its documentation is hosted on `https://cristalx.readthedocs.io/` and includes a detailed API reference as well as examples. The software was tested to work with Python 3.6 under Ubuntu 18.04 and Windows 8.1. All the required dependencies are installed by the *conda* package manager so that the installation into a separate environment is automatic. Additionally, the code is accessible through *Binder*, in which case no installation is needed at all.

2.1. Design choices

This software 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 software was built by adhering to the following rules:

1. Driven by actual needs
   Only implement features that are currently used. Adding extra features requires more testing, possibly more dependencies and therefore

[2]Whenever we write Python, we mean the CPython implementation.

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.

2. Build on well-established packages
   We rely on the scientific Python stack: *NumPy* [17] for array manipulations, *SciPy* [18] for interpolation and some other computations, *Matplotlib* [19] for visualization and *scikit-image* [20] for image processing. This ensures interoperability with other scientific codes and that our software is hopefully bug-free.
3. 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.
4. 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.

2.2. Segmentation

Image segmentation is performed on the image of the microstructure. The segmentation results in a so-called labeled image, an image having the same size as the input image. Distinct positive integers, called labels, are associated to groups of pixels in the labeled image and each pixel belongs to one and only one such group. The image segmentation is successful if the pixel groups correspond to grains in the original image. Depending on the microstructure and on the image quality (noise, etc.), image segmentation can be very challenging and cannot be fully automated in general. Our approach is to create a workflow that we found to give satisfying results, but also expose relevant parameters to the user so that the procedure can be fine-tuned for custom images. This workflow is detailed below. For the rest of this paper, the term *grain* will not only be used in the physical sense but it will also refer to the various representations of a grain (some connected pixels of an image, vertices of a polygon, planar surface bounded by splines). Its usage will be apparent from the context.

The workflow provided by the `segmentation` module is the following (functions or class methods are typed in `monospace font`).

1. Filtering (`segmentation.Segmentation.filter_image`)
   To remove unwanted artifacts on the image, noise removal is necessary

as a preprocessing step. Since the interfaces are important for us, we use median filtering for smoothing as it preserves the contours.

2. Superpixel segmentation (`segmentation.Segmentation.initial_segmentation`)
   In many practical cases, the interfaces do not appear clearly separated from the grains on the image. Therefore, simple thresholding methods are not successful in those cases. Instead, the Quick Shift [21] method is used, which is a superpixel segmentation algorithm.
3. Decrease the number of superpixels (`segmentation.Segmentation.merge_clusters`)
   As the superpixel segmentation in the previous step results in an oversegmented image, the region adjacency graph is constructed and used to merge some of the neighboring superpixels based on their similarity with respect to mean color [22].
4. Find the grain boundaries (`segmentation.Segmentation.find_grain_boundaries`)
   A binary image is returned with true values indicating the boundaries among the labelled regions.
5. Construct the skeleton (`segmentation.Segmentation.create_skeleton`)
   Use thinning on the grain boundary to obtain a single-pixel wide skeleton. If the automatic segmentation carried out so far is not good enough, the user can manually edit the grain boundaries as a graph in *ImagePy* [23]. The combination of the automatic segmentation with human supervision is a powerful way to achieve good results in a relatively short amount of time.
6. Watershed segmentation (`segmentation.Segmentation.watershed_segmentation`)
   Given the – possibly manually modified – skeleton, we want to obtain the regions (grains) they define. We use the watershed segmentation for that purpose. The Euclidean distance transform is computed on the skeleton to determine the catchment basins. It gives how far a point is from the closest skeleton pixel. The local minima of the negative of this distance function could be used as markers for the watershed segmentation. However, that results in an oversegmented image because each minimum acts as a catchment basin. Therefore, the markers are set to be the *extended minima* of the negative distance function, where the extended minimum is the regional minimum of the h-minima transformation (see Chapter 6 in [24]). The extended minima thus define the mask required for the marker-based watershed segmentation.

The steps above will be exemplified in Section 3.

Remark 1. *The image segmentation part of our software is useful by itself;*

for scientists who only work with images, without the need for subsequent meshing.

2.3. Geometry reconstruction

As explained in Section 1, to produce a high-quality mesh, we represent the grains as geometrical objects. For the proper representation of the geometry, the grains must form a tessellation of the domain. That is, there must be no gap or overlapping among them. It implies that an *interface* (common boundary between two grains) corresponds to the very same object whether we consider it as part of the first or the second neighboring grain. Therefore, the interfaces need to be identified first and then the grains must be constructed as surfaces bounded by the interfaces. The steps to obtain a geometrical description of the labeled image is now detailed.

1. Build the skeleton of the segmented image (`cad.build_skeleton`)
   The interfaces form the *skeleton* of the segmented (labeled) image and have a single-pixel width. Built-in functions of *scikit-image* are used to obtain the skeleton of the segmented image. First, the labeled image is surrounded by an artificial pixel region. Creating this extra region defines boundary interfaces for the boundary grains, making it possible to apply the same algorithms whether a grain lies along the boundary of the region or it is inside. The *connectivity graph* gives how the skeleton pixels are connected. For this purpose, the Python package *skan* is used [25]. *skan* stores the pixel connectivity in a sparse matrix, and provides for each skeleton pixel the number of neighboring skeleton pixels (*degree*). The degree allows classifying the skeleton pixels into various categories. If the degree is one, a particular skeleton pixel connects to only one other skeleton pixel, hence it is called an *end point*. Usually most skeleton pixels are internal, having degree two. If the degree is three or more, it is called a *junction*. As *skan* correctly states, *branches* can emerge between two junctions, an end point and a junction, between two end points, and can even indicate an isolated cycle that consists of degree two skeleton pixels only. However, in our case, it is easy to realize that only branches with junction-to-junction connections can form grain boundaries. Hence, we will use the term branch in this restricted sense for the rest.
2. Determine the grains bounded by the set of branches (`cad.skeleton2regions`)
   This problem could possibly be solved by a graph theoretic approach. Consider a graph in which the vertices are the branch end points and the edges are the branches. Properly selected cycles in this graph would give the branches that bound the grains. While this method is elegant and abstract, hence efficient and robust third-party graph libraries

could be used, it comes with several drawbacks. First, enumerating all the elementary cycles in a graph has a prohibitive time complexity [26]. Second, most of the elementary cycles do not enclose individual grains (e.g. a cycle could correspond to the union of two grains), hence a selection must be performed to filter out the non-physical cycles. Another technique could be to determine the minimum cycle basis for the graph. Unfortunately, the minimum cycle basis is not unique [27]. Therefore, it is not guaranteed that such a cycle will correspond to a grain. An additional difficulty arises when a small grain is located between two larger grains, resulting in a double edge in the corresponding graph. Many graph algorithms work under the premise that the graph contains no loops and multiple edges.

To alleviate this, an alternative solution is proposed, in which we use the information available in the segmented image. Specifically, we superimpose the skeletonized binary image on the labeled image and detect the labels around a given skeleton pixel. Performing this neighbor search on the skeleton pixels of a branch, and choosing the two most common labels, provides the neighboring grains to a branch.

What remains to be specified is what is considered as *neighborhood* of a skeleton pixel. Figure 1 shows a branch in boldface with a chosen skeleton pixel P on it, and the surroundings when superimposed on the corresponding labeled image. The labeled regions are distinguished by distinct colors. This configuration is taken from an actual microstructure we are working with and will be used as an example to demonstrate the effect of choosing various neighborhood definitions.

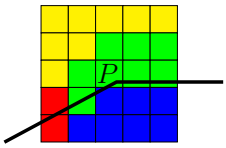


Figure 1: Skeleton pixel P and its neighborhood

Instead of creating two branches that separate the yellow region from the green and the green from the blue, *skan* creates only a single branch. From the full labeled image, of which a part is shown in Fig 1, a human observer can easily decide that the yellow and the blue regions must be the two neighbors of the branch. Nevertheless, the automated identification of the "correct" neighbors, relying on local information only, is not straightforward. Tables 1–2 collect the different strategies we implemented. Based on the two most common pixels in the neighborhood

of P, the algorithm identifies the neighboring regions (grains). The tables report whether the identification is correct. A natural choice is to consider the immediate neighbors of P, let it be 4-connectivity (von Neumann) or 8-connectivity (Moore), see the first two columns of Tab. 1. However, there are situations, as shown in Fig. 1, when a slender part of another grain (pixels in green) gets inserted between two grains that actually neighbor an interface. To mitigate this issue, generalized von Neumann and Moore neighborhoods can be considered, in which the neighbor search is extended for a larger range n, thereby decreasing the effect of the immediate neighbors. In this example, $n = 2$ is not sufficient to obtain the desired neighbors (see the first two columns of Tab. 2). Although $n = 3$ solves the problem in this example, setting n too large introduces another issue: small grains remain unidentified. Another strategy is to make the mask hollow. This choice discards the close neighbors, thereby protecting against slender grain insertions (cf. columns 3–4 in Tables 1–2). One can conclude from this example that (i) the best strategy is configuration-dependent and (ii) it is easy to switch among the strategies, or implement new ones, by modifying the mask. So instead of constructing complex and costly methods to detect the neighbors, parametrizable heuristic algorithms are provided in the software. We accept that perfect results are nearly impossible to achieve for arbitrary labeled images, therefore we allow the user to choose the algorithm that fits the best for a given image. The hollow neighborhood strategy with $n = 2$ proved to be the winning choice for the microstructure investigated in Section 3, resulting in a topologically admissible geometry and leaving only one tiny grain unidentified.

Connectivity	von Neumann	Moore	von Neumann, hollow	Moore, hollow
Mask				
Grain pixel	Occurrence within the mask			
yellow	0	1	0	1
green	4	6	3	5
red	0	0	0	0
blue	1	2	1	2
Neighbors	green - blue	green - blue	green - blue	green - blue
Correct	no	no	no	no

Table 1: Neighborhood definitions with range $n = 1$ and their assessment for the configuration depicted in Fig. 1.

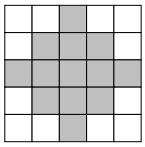
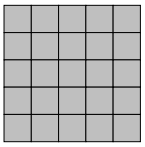
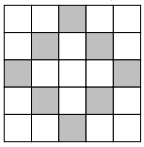
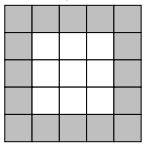
Connectivity	von Neumann	Moore	von Neumann, hollow	Moore, hollow
Mask				
Grain pixel	Occurrence within the mask			
■	3	8	3	7
■	7	8	3	2
■	0	2	0	2
■	3	7	2	5
Neighbours	■ - ■ / ■ - ■	■ - ■	■ - ■	■ - ■
Correct	no/no	no	no	yes

Table 2: Neighborhood definitions with range $n = 2$ and their assessment for the configuration depicted in Fig. 1.

To find the branches that bound a grain, the branch–grain connectivities determined above are inverted. The `cad.skeleton2regions` function can be perceived as an intermediate step between a skeleton network and completely geometrical representation of the grains. That is, it keeps the key topological information required to create a fully geometrical description, but it also contains the coordinates of the grain boundaries. The outputs of this function can be used to build different grain representations (e.g. polygonal or spline surface).

3. Find oriented grain boundaries (`cad.branches2boundary`)
   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 `cad.branches2boundary` function uses a simple brute-force method to interlace the branches based on their common junctions. Both clockwise and counter-clockwise orientations are supported. By the end of this step, we obtain a fully geometrical description because each grain is now given by a series of points along its boundary.
4. Represent each grain as a planar surface
   Two representations are provided by the software. The simpler one considers the points on the boundary of a grain as vertices of a polygon. This straightforward method, implemented in `cad.region_as_polygon`, has the disadvantage that the generated mesh on the polygon will not be adaptable, i.e the mesh cannot be controlled. Depending on

the resolution of the labeled image, the polygon can consist of hundreds of vertices. A mesh generator respects the tiny edges of the polygon and therefore the mesh will be refined or coarsened based on the boundary vertices of the grains, and not on other, more adequate, criteria. Furthermore, the resulting mesh is unnecessarily dense. As a partial solution, the Douglas-Peucker algorithm could be used to simplify the branches before forming the polygon. However, the order of simplification we want to achieve depends on a characteristic length, which will make the process tedious to use in practice.

A better solution is to use CAD tools to control the geometry.

(a) Create B-spline passing through points (`cad.fit_spline`)
NURBS-Python is an easy-to-use pure Python package for spline manipulation, with no dependency on top of the standard library [28]. Its approximation capabilities would be sufficient for our task but, as far as we know, creating a surface bounded by splines is not possible with it. Also, at the time of writing, it cannot export spline surfaces to a lossless CAD format, such as STEP. Therefore, we rely on *PythonOCC* [29], which is a Python wrapper around the mature and free *Open CASCADE Technology* (*OCCT*). The user can set different parameters to control the approximation. These parameters are the minimum and maximum degree of the spline, its continuity and a tolerance. The tolerance instructs *OCCT* to construct the spline such that the distance from any data point to the spline is smaller than the prescribed tolerance. As the tolerance tends to zero, one obtains a spline interpolant.

(b) Approximate each branch with a B-spline (`cad.branches2splines`)
The spline approximation done by `cad.fit_spline` is performed for every branch.

(c) Represent each grain as a spline surface (`cad.region_as_splinegon`)
For every grain, the splines forming its boundary are combined into a closed contour. Then a planar surface is created, bounded by this contour.

5. Export the geometry (`cad.regions2step`)
It is important that the surface of each grain is given in a lossless format, e.g. in STEP (**ST**andardized **E**xchange of **P**roduct) and not as a mesh representation. The `cad.regions2step` function builds a compound surface from the individual grain surfaces, which is then written to a STEP file by `cad.write_step_file`. The exported STEP file is used as an input to a mesh generator, such as *Salome* [30], *Netgen* [31] or *Gmsh* [32].

2.4. Additional features

The software contains convenience functions and classes. Currently, the following Python modules are the most relevant ones:

- `geometry`
  Implements computational geometry algorithms, with the emphasis being on minimalism and not on feature-richness. The `TriMesh` class is a representation of an unstructured mesh with triangular cells. Hundreds of thousands of cells can be handled efficiently, which is often sufficient for 2D problems. The `Polygon` class is an abstraction for simple, possibly concave, polygons. Since it has member functions for computing the area, the centroid and the diameter of polygons, this class proves to be useful when analyzing grains that are approximated as polygons. Polygons also naturally arise when grains are discretized to an assembly of cells.
- `med`
  The MED data model is used by Salome as an exchange format to encompass various simulation codes in a framework [30]. It is an extended version of HDF5, supporting parallel meshes and fields. Salome's *MEDCoupling* tool provides C++ and Python APIs for interacting with meshes and fields. Although MEDCoupling is powerful, its Python API is available only from the Salome kernel. Moreover, it may lack certain mesh processing functionalities a user might need. 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 modeling 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.
- `utils`
  Depends only on the standard library and NumPy, and contains utility functions that do not fit to other categories.

For other helper functions, e.g. the ones enabling profiling the code, see the source code and its documentation.

2.5. Collaboration

New ideas, bug reports and critiques on the code and on the documentation are welcome, and they are tackled on the GitHub issue tracker of the project. The `contributing.md` file contains the details on how to submit proposals.

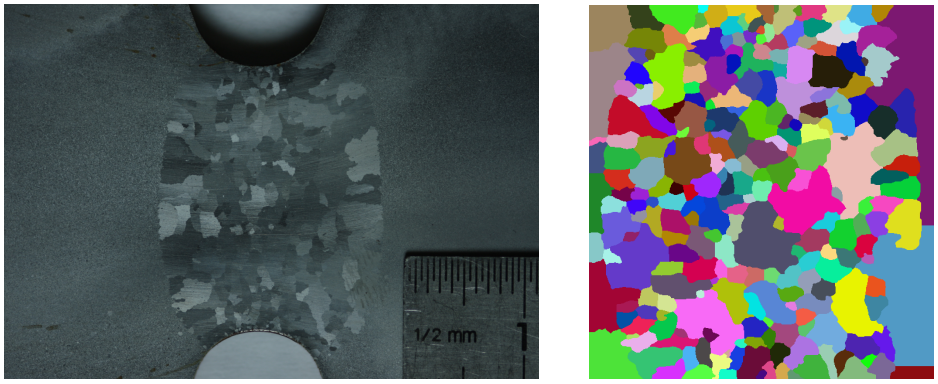
3. An illustrative example

`CristalX` was built around the following application. We are interested in how the size gradient of grains, and the material they are made up of influence the resistance of train wheels and axles to fatigue loading. Compared to [1], we take into account the effect of each individual grain in the microstructure. A high-level overview to tackle this problem is drafted below. The code is available as a Jupyter notebook in the repository, while the physical problem and the numerical solution scheme will be detailed in another article.

1. Input: microstructure as a photo
   We will consider the microstructure in Fig. 2a available in the thesis [33]. To make the segmentation easier, the central part is cropped.
2. Identify the individual grains (`segmentation.py`)
   After performing the steps described in Section 2.2, we arrive at the segmented image shown in Fig. 2b. For better results, we could have used *ImagePy*, but for the purpose of presentation we stayed with the automatic workflow. Nevertheless, Fig. 3 compares the skeleton obtained automatically (Fig. 3a) and with manual editing (Fig. 3b). So the segmented image in Fig. 2b is based on the skeleton in Fig. 3a. At first glance, the identified grains in Fig. 2b do not correspond to the actual grains we can see in Fig. 2a. In fact, there are more segmented regions than grains in reality. However, the contours of the "real" grains are well identified. This is relevant from the point of view of the invested time because the extraneous regions can be easily and quickly merged afterwards, while the manual grain boundary detection would take a lot of time.
3. Obtain planar B-spline surfaces for the grains (`cad.py`)
   With the algorithms in Section 2.3, all but a very tiny grain (consisting of a few pixels only) could be identified. The missing grain acts as a hole in the domain.
4. Manual reparation of the CAD geometry in Salome for filling the hole
5. Mesh generation
   Thanks to the CAD geometry, excellent mesh quality is achieved by the *Netgen* mesh generator [31] without any user intervention. Figure 4 compares the mesh obtained by *OOF2* (Fig. 4a) with the one that is

based on the intermediate geometry representation (Fig. 4b). The low-quality cells generated by *OOF2* hampers the convergence of Abaqus solvers in several loading steps. Notice the finer mesh in the vicinity of the grain boundaries in Fig. 4b. This mesh adaptivity will be important in computations as large deformation is expected to happen along the grain boundaries. We note that the colors are associated randomly to the grains, this is why they do not match in Fig. 2b and in Fig. 4a, though they depict the same microstructure.

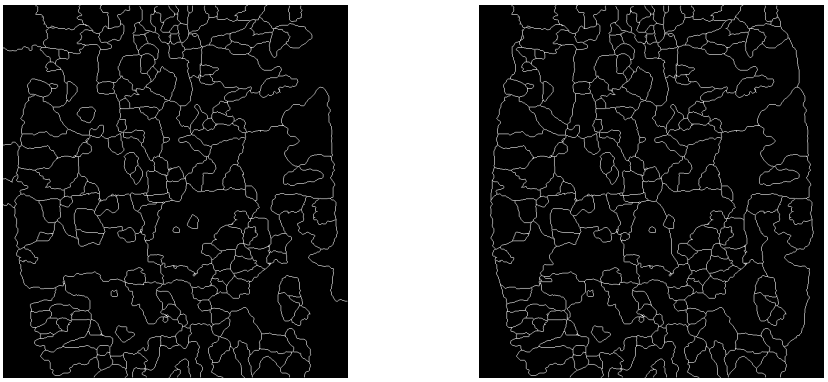
6. Loading the mesh from a .med file (`med.py`)
   The mesh is exported from Salome as a MED file. Using the `med` module of `CristalX`, the triangular mesh cells for each grain and the boundary nodes are extracted as NumPy arrays.
7. Mesh manipulation utilities (`geometry.py`)
   The mesh is scaled to define the computational domain based on physical units instead of pixel units.
8. Abaqus input file (`abaqus.py`)
   The material parameters, the boundary conditions and the mesh is written to a text file that the finite element program *Abaqus* can interpret.
9. Projection between a Cartesian grid and an unstructured mesh (`dic.py`)
   The experimentally obtained displacement field, obtained by the digital image correlation (DIC) technique, is available known on a Cartesian grid. To compare the field values with the numerical data (available at the nodes of the mesh), scattered interpolation is implemented.



(a) Microstructure (source: [33])

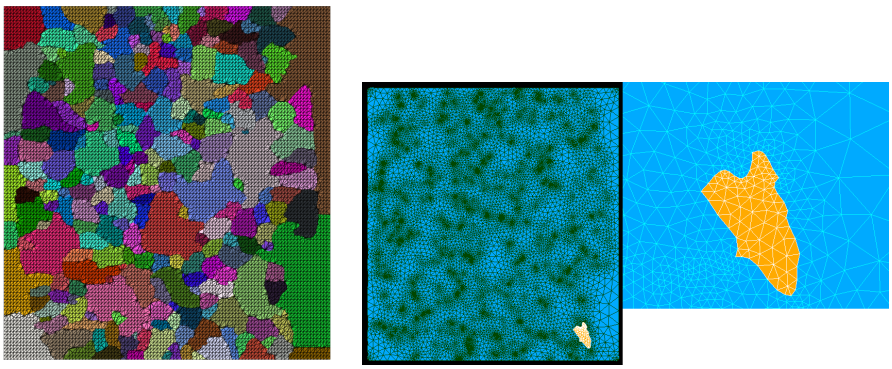
(b) Segmented microstructure

Figure 2: Close-up of a tensile specimen and its image segmentation



(a) Skeleton by a fully automatic workflow (b) Skeleton with manual modifications

Figure 3: Skeleton of the image after the initial Quick Shift segmentation



(a) Mesh by OOF (b) Mesh with our geometry-based workflow

Figure 4: Generated mesh on the segmented image

4. Impact

Our tool is flexible because it consists of loosely coupled modules for the typical steps of microstructure identification and handling. Therefore, these components can be developed independently by researchers, depending on their needs. The detailed documentation, the contribution guidelines, and the fact that we exclusively rely on open-source tools promote collaboration. Indeed, we are interested in incorporating research ideas from external contributors to `CristalX`. Our geometry reconstruction algorithms do not assume polygonal grains. Convex, concave, non-simply connected domains

can all be identified successfully as Fig. 5 demonstrates. While microstructure generator tools have the freedom to restrict their attention to relatively simple grain shapes, real microstructures often consist of very complex grains (as the one in Fig. 5). Hence, for experimentally obtained microstructures, our algorithms are crucial.

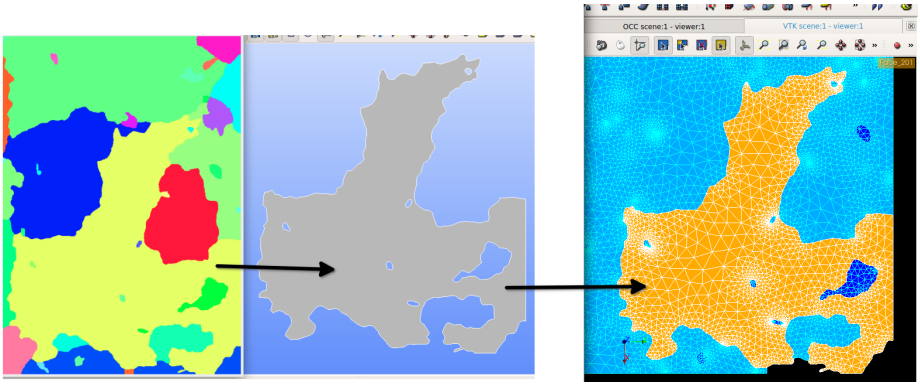


Figure 5: The image on the left shows a part of the segmented microstructure, in which the colors correspond to the grains. Each segmented region is then transformed to an explicitly given geometry, the boundaries of which are parametrized by B-splines (center). This kind of description of the geometry makes it possible to generate high quality meshes (right).

We expect that `CristalX` will be helpful for material scientists who want to characterize grain-based microstructures or intend to perform finite element computations on high-quality meshes. Finally, we mention that although `CristalX` was developed for microstructural analysis in mind, other applications in which a geometrical representation of a tessellation is needed can benefit from the method described here.

5. Conclusions and future work

We argued how a direct geometrical representation of grains results in a better mesh quality, necessary for applications involving nonlinearities. Our contribution is twofold. First, an image segmentation workflow is tailored for the identification of granular microstructures. Second, the segmented image is turned to a CAD geometry format with customizable smoothness. These two core modules were used as a foundation to develop other modules that allow us to solve a relevant industrial problem. The building blocks of our software are general enough so that they can serve as a point of departure for other areas, in which grain-like shapes tessellate a domain.

A collection of heuristics could be built to search for neighboring pixels when identifying the branch–grain connectivities (cf. Section 2.3). The growing collection of such heuristics would typically be application-driven. When an existing strategy fails, one searches for another one and save that prevailing strategy for further use.

It would be reasonable to support a less strict image segmentation workflow by allowing the user to select among different initial segmentation methods, not just the Quick Shift segmentation.

The current version of the software is 1.0.1.

Conflict of interest

No conflict of interest exists: We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgments

The support of Xiaolong Yan, one of the creators of *ImagePy*, is acknowledged for giving advice on image segmentation and in particular for his help with *ImagePy*. We also thank Pierre Baudoin for providing us the microstructure we used to illustrate the workflow in this paper.
This work has been carried out within the CNRS SWIT'lab joint laboratory (LaMcube CNRS 9013, LAMIH CNRS 8201, MG Valdunes company and CNRS) and has also been supported by the ELSAT2020 research project. SWIT'lab and ELSAT2020 are co-financed by the European Union with the European Regional Development Fund, the French state and the Hauts-de-France Region Council.

References

[1] P. Baudoin, V. Magnier, A. E. Bartali, J.-F. Witz, P. Dufrenoy, F. Demilly, É. Charkaluk, Numerical investigation of fatigue strength of grain size gradient materials under heterogeneous stress states in a notched specimen, Int. J. Fatigue 87 (2016) 132–142. doi:10.1016/j.ijfatigue.2016.01.022.

[2] H. Xiong, S. Chu, P. Lei, Z. Li, K. Zhou, Ti(C,N)-based cermets containing uniformly dispersed ultrafine rimless grains: Effect of VC additions on the microstructure and mechanical properties, Ceram. Int. 46 (12) (2020) 19904–19911. doi:10.1016/j.ceramint.2020.05.055.

[3] C. Sabbarese, F. Ambrosino, A. D'Onofrio, Development of radon transport model in different types of dwellings to assess indoor activity concentration, J. Environ. Radioact. 227 (2021) 106501. doi:10.1016/j.jenvrad.2020.106501.

[4] J. Fu, S. Cui, S. Cen, C. Li, Statistical characterization and reconstruction of heterogeneous microstructures using deep neural network, Comput. Methods Appl. Mech. Engrg. 373 (2021) 113516. doi:10.1016/j.cma.2020.113516.

[5] K. A. Hart, J. J. Rimoli, Generation of statistically representative microstructures with direct grain geometry control, Comput. Methods Appl. Mech. Eng. 370 (Oct. 2020). doi:10.1016/j.cma.2020.113242.

[6] R. Quey, P. R. Dawson, F. Barbe, Large-scale 3D random polycrystals for the finite element method: Generation, meshing and remeshing, Comput. Methods Appl. Mech. Eng. 200 (17-20) (2011) 1729–1745. doi:10.1016/j.cma.2011.01.002.

[7] Ren, Malik, Learning a classification model for segmentation, in: Proceedings Ninth IEEE International Conference on Computer Vision, IEEE, 2003. doi:10.1109/iccv.2003.1238308.

[8] D. Stutz, A. Hermans, B. Leibe, Superpixels: An evaluation of the state-of-the-art, Comput. Vision Image Understanding 166 (2018) 1–27. doi:10.1016/j.cviu.2017.03.007.

[9] S. Beucher, C. Lantuejoul, Use of watersheds in contour detection, in: International Workshop on image processing: Real-time Edge and Motion detection/estimation, Rennes, France, 1979.
URL http://www.cmm.mines-paristech.fr/~beucher/publi/watershed.pdf

[10] M. G. D. Geers, V. G. Kouznetsova, W. A. M. Brekelmans, Multi-scale computational homogenization: Trends and challenges, J. Comput. Appl. Math. 234 (7) (2010) 2175–2182. doi:10.1016/j.cam.2009.08.077.

[11] A. C. E. Reid, S. A. Langer, R. C. Lua, V. R. Coffman, S.-I. Haan, R. E. García, Image-based finite element mesh construction for material microstructures, Comput. Mater. Sci. 43 (4) (2008) 989–999. doi:10.1016/j.commatsci.2008.02.016.

[12] V. R. Coffman, A. C. E. Reid, S. A. Langer, G. Dogan, OOF3D: An image-based finite element solver for materials science, Math. Comput. Simulation 82 (12) (2012) 2951–2961. doi:10.1016/j.matcom.2012.03.003.

[13] O. Goksel, S. E. Salcudean, Image-Based Variational Meshing, IEEE Transactions on Medical Imaging 30 (1) (2011) 11–21. doi:10.1109/tmi.2010.2055884.

[14] G. Legrain, P. Cartraud, I. Perreard, N. Moës, An X-FEM and level set computational approach for image-based modelling: Application to homogenization, Internat. J. Numer. Methods Engrg. 86 (7) (2010) 915–934. doi:10.1002/nme.3085.

[15] J. R. Shewchuk, What is a Good Linear Element? - Interpolation, Conditioning, and Quality Measures (2002).
URL https://people.eecs.berkeley.edu/~jrs/papers/elemj.pdf

[16] T. E. Oliphant, Python for Scientific Computing, Computing in Science & Engineering 9 (3) (2007) 10–20. doi:10.1109/mcse.2007.58.

[17] S. van der Walt, S. C. Colbert, G. Varoquaux, The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering 13 (2) (2011) 22–30. doi:10.1109/mcse.2011.37.

[18] P. Virtanen, , R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0: fundamental algorithms for scientific computing in Python, Nat. Methods 17 (3) (2020) 261–272. doi:10.1038/s41592-019-0686-2.

[19] J. D. Hunter, Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering 9 (3) (2007) 90–95. doi:10.1109/mcse.2007.55.

[20] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, scikit-image: image processing in Python, PeerJ 2 (e453) (Jun. 2014). doi:10.7717/peerj.453.

[21] A. Vedaldi, S. Soatto, Quick Shift and Kernel Methods for Mode Seeking, in: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 705–718. doi:10.1007/978-3-540-88693-8_52.

[22] A. Tremeau, P. Colantoni, Regions adjacency graph applied to color image segmentation, IEEE Transactions on Image Processing 9 (4) (2000) 735–744. doi:10.1109/83.841950.

[23] A. Wang, X. Yan, Z. Wei, ImagePy: an open-source, Python-based and platform-independent software package for bioimage analysis, Bioinformatics 34 (18) (2018) 3238–3240. doi:10.1093/bioinformatics/bty313.

[24] P. Soille, Morphological Image Analysis: Principles and Applications, Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-662-05088-0.

[25] J. Nunez-Iglesias, A. J. Blanch, O. Looker, M. W. Dixon, L. Tilley, A new Python library to analyse skeleton images confirms malaria parasite remodelling of the red blood cell membrane skeleton, PeerJ 6 (2018) e4312. doi:10.7717/peerj.4312.

[26] D. B. Johnson, Finding All the Elementary Circuits of a Directed Graph, SIAM Journal on Computing 4 (1) (1975) 77–84. doi:10.1137/0204007.

[27] P. Vismara, Union of all the Minimum Cycle Bases of a Graph, The Electronic Journal of Combinatorics 4 (1) (Jan. 1997). doi:10.37236/1294.

[28] O. R. Bingol, A. Krishnamurthy, NURBS-Python: An open-source object-oriented NURBS modeling framework in Python, SoftwareX 9 (2019) 85–94. doi:10.1016/j.softx.2018.12.005.

[29] T. Paviot, PythonOCC – Python package for 3D CAD/BIM/PLM/CAM, https://github.com/tpaviot/pythonocc-core, accessed January 31, 2021(Aug. 2014).
URL https://github.com/tpaviot/pythonocc-core

[30] A. Ribes, C. Caremoli, Salome platform component model for numerical simulation, in: 31st Annual International Computer Software and Applications Conference - Vol. 2 - (COMPSAC 2007), IEEE, 2007. doi:10.1109/compsac.2007.185.

[31] J. Schöberl, NETGEN an advancing front 2D/3D-mesh generator based on abstract rules, Computing and Visualization in Science 1 (1) (1997) 41–52. doi:10.1007/s007910050004.

[32] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, Int. J. Numer. Methods Eng. 79 (11) (2009) 1309–1331. doi:10.1002/nme.2579.

[33] P. Baudoin, Caractérisation et identification de propriétés de matériaux métalliques à gradients de microstructure, phdthesis, Université de Lille I (2015).