



HAL
open science

CristalX: Facilitating simulations for experimentally obtained grain-based microstructures

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

► **To cite this version:**

Zoltan Csati, Jean-Francois Witz, Vincent Magnier, Ahmed El Bartali, Nathalie Limodin, et al..
CristalX: Facilitating simulations for experimentally obtained grain-based microstructures. *SoftwareX*,
2021, 14, pp.100669. 10.1016/j.softx.2021.100669 . hal-03409334

HAL Id: hal-03409334

<https://hal.science/hal-03409334>

Submitted on 9 Nov 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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)

Preprint submitted to SoftwareX

January 31, 2021

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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 <code>environment.yml</code> file in the repository
Link to developer documentation	https://cristalx.readthedocs.io/
Support email for questions	zoltan-c@keemail.me

Code metadata

1 **1. Introduction**

2 Grain-based microstructures occur in nature (e.g. crystalline rocks) and
3 also play a role in industrial processes such as forging [1] or developing cer-
4 ments [2]. The effect of the grain size and shape, and the distribution of the
5 constituents at the micro scale determines the behaviour of the material at
6 the macro scale¹ [3]. It is therefore of interest to characterize the microstruc-
7 ture. One way to do it is to synthetically generate many of them with some
8 relevant characteristics drawn from statistical distributions and use statisti-
9 cal methods to compute the average response to given loading conditions
10 [4]. The advantage of computational microstructure generation is that one
11 can perform computational experiments by changing the parameters. When
12 a new material is under development, it is highly useful to virtually ex-
13 periment with the microstructure without having to produce samples and
14 conduct real experiments on them. It is also a faster and cheaper method
15 than having to perform time-consuming and often costly measurements. For
16 grain generation, see the recent paper [5]. The open-source tool *Neper* [6]
17 also has a module for generating polycrystalline microstructures. This ap-
18 proach works well when the microstructure can be properly captured by the
19 statistical parameters. However, when this is not the case or if a specific real
20 microstructure is of interest, measurements are necessary. The outcomes of
21 measurements are images in which the grains are to be identified. This is a
22 task for image segmentation.

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

¹For 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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

82 The quality of the mesh influences both the interpolation error and the
83 conditioning of the discretized problem [15]. In case of nonlinear problems
84 (e.g. contact, large deformation elasticity, plasticity), a high quality mesh is
85 crucial to obtain an accurate solution, or even to have a convergent solu-
86 tion at a given load increment. While generating adaptive meshes directly
87 based on an image is an ongoing research effort, image-based methods usu-
88 ally concentrate on creating meshes on the grains. However, in many cases
89 interesting phenomena, such as strain localization, happen along the grain
90 boundaries (interfaces). It is therefore useful to be able to extract the in-
91 terfaces and control the mesh generation e.g. by constructing graded meshes
92 near the interfaces. Mesh refinement is also necessary to conduct convergence
93 studies. Another problem we faced was mesh repairing. Since the mesh qual-
94 ity obtained by *OO2* was not good enough, we considered improving the
95 mesh with third-party tools. However, the new mesh obtained by those tools
96 does not respect the original microstructure, which can be detrimental for
97 small grains consisting of only a few cells. What is more, the new mesh has
98 no information about which cell belongs to which grain, which information
99 is necessary if different material properties are associated to the different
100 grains. The challenges above lead us to represent the grains as an *exact*
101 geometry, rather than an assembly of cells, which is an inherently discrete
102 approach. We note that after extracting the interfaces, non-conforming (and
103 possibly block-structured) meshes could be used, as in X-FEM. That choice
104 eases the mesh generation but at the same time requires specialized solvers
105 and custom preconditioners. Therefore, in this paper, traditional conforming
106 mesh generation is applied, which has the advantage that robust open-source

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

107 or proprietary software can be utilized for that purpose.

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

123 2. Software description

124 Our project, called `CristalX`, is written in pure Python², a widely-used
125 free and open-source general purpose language with a rich scientific ecosystem
126 [16]. The source code of the software is available on [https://github.com/
127 CsatiZoltan/CristalX](https://github.com/CsatiZoltan/CristalX) under the LGPL-3.0 license. Its documentation is
128 hosted on <https://cristalx.readthedocs.io/> and includes a detailed API
129 reference as well as examples. The software was tested to work with Python
130 3.6 under Ubuntu 18.04 and Windows 8.1. All the required dependencies
131 are installed by the `conda` package manager so that the installation into
132 a separate environment is automatic. Additionally, the code is accessible
133 through `Binder`, in which case no installation is needed at all.

134 2.1. Design choices

135 This software is not a black-box library such as BLAS, neither is a GUI-
136 based application intended for end-users. It is rather an easy-to-use and
137 extensible set of Python codes that provide the basic functionalities that sci-
138 entists can extend based on their needs. The software was built by adhering
139 to the following rules:

- 140 1. Driven by actual needs
141 Only implement features that are currently used. Adding extra fea-
142 tures requires more testing, possibly more dependencies and therefore

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

143 code bloat, and increases the cognitive load of the user. Instead, the
144 emphasis is on creating a stable minimum core library that can be eas-
145 ily extended according to users' demands. Consequently, application
146 code is separated from the core modules.

147 2. Build on well-established packages

148 We rely on the scientific Python stack: *NumPy* [17] for array manip-
149 ulations, *SciPy* [18] for interpolation and some other computations,
150 *Matplotlib* [19] for visualization and *scikit-image* [20] for image pro-
151 cessing. This ensures interoperability with other scientific codes and
152 that our software is hopefully bug-free.

153 3. Minimize the dependencies

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

159 4. High-quality documentation

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

163 2.2. Segmentation

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

179 The workflow provided by the `segmentation` module is the following (func-
180 tions or class methods are typed in monospace font).

181 1. Filtering (`segmentation.Segmentation.filter_image`)

182 To remove unwanted artifacts on the image, noise removal is necessary

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

185 2. Superpixel segmentation (`segmentation.Segmentation.initial_segmentation`)
186 In many practical cases, the interfaces do not appear clearly separated
187 from the grains on the image. Therefore, simple thresholding methods
188 are not successful in those cases. Instead, the Quick Shift [21] method
189 is used, which is a superpixel segmentation algorithm.

190 3. Decrease the number of superpixels (`segmentation.Segmentation.merge_-`
191 `clusters`)
192 As the superpixel segmentation in the previous step results in an over-
193 segmented image, the region adjacency graph is constructed and used
194 to merge some of the neighboring superpixels based on their similarity
195 with respect to mean color [22].

196 4. Find the grain boundaries (`segmentation.Segmentation.find_grain_-`
197 `boundaries`)
198 A binary image is returned with true values indicating the boundaries
199 among the labelled regions.

200 5. Construct the skeleton (`segmentation.Segmentation.create_skeleton`)
201 Use thinning on the grain boundary to obtain a single-pixel wide skele-
202 ton. If the automatic segmentation carried out so far is not good
203 enough, the user can manually edit the grain boundaries as a graph
204 in *ImagePy* [23]. The combination of the automatic segmentation with
205 human supervision is a powerful way to achieve good results in a rela-
206 tively short amount of time.

207 6. Watershed segmentation (`segmentation.Segmentation.watershed_-`
208 `segmentation`)
209 Given the – possibly manually modified – skeleton, we want to obtain
210 the regions (grains) they define. We use the watershed segmentation
211 for that purpose. The Euclidean distance transform is computed on the
212 skeleton to determine the catchment basins. It gives how far a point is
213 from the closest skeleton pixel. The local minima of the negative of this
214 distance function could be used as markers for the watershed segmen-
215 tation. However, that results in an oversegmented image because each
216 minimum acts as a catchment basin. Therefore, the markers are set to
217 be the *extended minima* of the negative distance function, where the
218 extended minimum is the regional minimum of the *h*-minima transfor-
219 mation (see Chapter 6 in [24]). The extended minima thus define the
220 mask required for the marker-based watershed segmentation.

221 The steps above will be exemplified in Section 3.

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

225 *2.3. Geometry reconstruction*

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

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

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

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

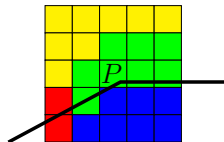


Figure 1: Skeleton pixel P and its neighborhood

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

296 of P , the algorithm identifies the neighboring regions (grains). The
 297 tables report whether the identification is correct. A natural choice
 298 is to consider the immediate neighbors of P , let it be 4-connectivity
 299 (von Neumann) or 8-connectivity (Moore), see the first two columns of
 300 Tab. 1. However, there are situations, as shown in Fig. 1, when a slender
 301 part of another grain (pixels in green) gets inserted between two grains
 302 that actually neighbor an interface. To mitigate this issue, generalized
 303 von Neumann and Moore neighborhoods can be considered, in which
 304 the neighbor search is extended for a larger range n , thereby decreasing
 305 the effect of the immediate neighbors. In this example, $n = 2$ is not
 306 sufficient to obtain the desired neighbors (see the first two columns of
 307 Tab. 2). Although $n = 3$ solves the problem in this example, setting
 308 n too large introduces another issue: small grains remain unidentified.
 309 Another strategy is to make the mask hollow. This choice discards
 310 the close neighbors, thereby protecting against slender grain insertions
 311 (cf. columns 3–4 in Tables 1–2). One can conclude from this example
 312 that (i) the best strategy is configuration-dependent and (ii) it is easy
 313 to switch among the strategies, or implement new ones, by modifying
 314 the mask. So instead of constructing complex and costly methods to
 315 detect the neighbors, parametrizable heuristic algorithms are provided
 316 in the software. We accept that perfect results are nearly impossible
 317 to achieve for arbitrary labeled images, therefore we allow the user to
 318 choose the algorithm that fits the best for a given image. The hollow
 319 neighborhood strategy with $n = 2$ proved to be the winning choice for
 320 the microstructure investigated in Section 3, resulting in a topologically
 321 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			
	0	1	0	1
	4	6	3	5
	0	0	0	0
	1	2	1	2
Neighbors	 - 	 - 	 - 	 - 
Correct	no	no	no	no

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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.

322 To find the branches that bound a grain, the branch–grain connec-
323 tivities determined above are inverted. The `cad.skeleton2regions`
324 function can be perceived as an intermediate step between a skeleton
325 network and completely geometrical representation of the grains. That
326 is, it keeps the key topological information required to create a fully ge-
327 ometrical description, but it also contains the coordinates of the grain
328 boundaries. The outputs of this function can be used to build different
329 grain representations (e.g. polygonal or spline surface).
330 3. Find oriented grain boundaries (`cad.branches2boundary`)
331 The previous part of the reconstruction algorithm determined which
332 branches bound a grain. In order to obtain a surface representation of
333 a grain, the boundary must be oriented and hence the branches must
334 be connected in the appropriate order. The `cad.branches2boundary`
335 function uses a simple brute-force method to interlace the branches
336 based on their common junctions. Both clockwise and counter-clockwise
337 orientations are supported. By the end of this step, we obtain a fully
338 geometrical description because each grain is now given by a series of
339 points along its boundary.
340 4. Represent each grain as a planar surface
341 Two representations are provided by the software. The simpler one
342 considers the points on the boundary of a grain as vertices of a poly-
343 gon. This straightforward method, implemented in `cad.region_as_-`
344 `polygon`, has the disadvantage that the generated mesh on the polygon
345 will not be adaptable, i.e the mesh cannot be controlled. Depending on

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

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

- 356 (a) Create B-spline passing through points (`cad.fit_spline`)
357 *NURBS-Python* is an easy-to-use pure Python package for spline
358 manipulation, with no dependency on top of the standard library
359 [28]. Its approximation capabilities would be sufficient for our task
360 but, as far as we know, creating a surface bounded by splines is
361 not possible with it. Also, at the time of writing, it cannot export
362 spline surfaces to a lossless CAD format, such as STEP. Therefore,
363 we rely on *PythonOCC* [29], which is a Python wrapper around
364 the mature and free *Open CASCADE Technology (OCCT)*. The
365 user can set different parameters to control the approximation.
366 These parameters are the minimum and maximum degree of the
367 spline, its continuity and a tolerance. The tolerance instructs
368 *OCCT* to construct the spline such that the distance from any
369 data point to the spline is smaller than the prescribed tolerance.
370 As the tolerance tends to zero, one obtains a spline interpolant.
- 371 (b) Approximate each branch with a B-spline (`cad.branches2splines`)
372 The spline approximation done by `cad.fit_spline` is performed
373 for every branch.
- 374 (c) Represent each grain as a spline surface (`cad.region_as_splinegon`)
375 For every grain, the splines forming its boundary are combined
376 into a closed contour. Then a planar surface is created, bounded
377 by this contour.

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

386 *2.4. Additional features*

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

- 389 • **geometry**
390 Implements computational geometry algorithms, with the emphasis be-
391 ing on minimalism and not on feature-richness. The `TriMesh` class is
392 a representation of an unstructured mesh with triangular cells. Hun-
393 dreds of thousands of cells can be handled efficiently, which is often
394 sufficient for 2D problems. The `Polygon` class is an abstraction for
395 simple, possibly concave, polygons. Since it has member functions for
396 computing the area, the centroid and the diameter of polygons, this
397 class proves to be useful when analyzing grains that are approximated
398 as polygons. Polygons also naturally arise when grains are discretized
399 to an assembly of cells.
 - 400 • **med**
401 The MED data model is used by Salome as an exchange format to
402 encompass various simulation codes in a framework [30]. It is an ex-
403 tended version of HDF5, supporting parallel meshes and fields. Sa-
404 lome’s *MEDCoupling* tool provides C++ and Python APIs for inter-
405 acting with meshes and fields. Although MEDCoupling is powerful,
406 its Python API is available only from the Salome kernel. Moreover,
407 it may lack certain mesh processing functionalities a user might need.
408 Since meshes consisting of cells of the same type (e.g. triangles) can
409 be represented as homogeneous and contiguous arrays, converting the
410 mesh from MED to NumPy arrays seems a reasonable choice. This is
411 what our `med` module does: it provides a thin wrapper around MED-
412 Coupling to extract the mesh and the defined groups (cell and vertex
413 groups) from the MED file and convert them to NumPy arrays. This
414 way, the user who deals with numerical modeling can implement their
415 mesh processing algorithms based on NumPy arrays, which is fast and
416 straightforward. Furthermore, the person who performs the CAD op-
417 erations and has Salome installed, can use our `med` module to export
418 the mesh to NumPy arrays so that the numerical analyst can directly
419 work on it without having to have Salome installed and without any
420 knowledge on the MEDCoupling API.
 - 421 • **utils**
422 Depends only on the standard library and NumPy, and contains utility
423 functions that do not fit to other categories.
- 424 For other helper functions, e.g. the ones enabling profiling the code, see the
425 source code and its documentation.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

426 *2.5. Collaboration*

427 New ideas, bug reports and critiques on the code and on the documenta-
428 tion are welcome, and they are tackled on the GitHub issue tracker of the
429 project. The `contributing.md` file contains the details on how to submit
430 proposals.

431 **3. An illustrative example**

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

- 439 1. Input: microstructure as a photo

440 We will consider the microstructure in Fig. 2a available in the thesis
441 [33]. To make the segmentation easier, the central part is cropped.

- 442 2. Identify the individual grains (`segmentation.py`)

443 After performing the steps described in Section 2.2, we arrive at the
444 segmented image shown in Fig. 2b. For better results, we could have
445 used *ImagePy*, but for the purpose of presentation we stayed with the
446 automatic workflow. Nevertheless, Fig. 3 compares the skeleton obtained
447 automatically (Fig. 3a) and with manual editing (Fig. 3b). So
448 the segmented image in Fig. 2b is based on the skeleton in Fig. 3a.
449 At first glance, the identified grains in Fig. 2b do not correspond to
450 the actual grains we can see in Fig. 2a. In fact, there are more seg-
451 mented regions than grains in reality. However, the contours of the
452 “real” grains are well identified. This is relevant from the point of view
453 of the invested time because the extraneous regions can be easily and
454 quickly merged afterwards, while the manual grain boundary detection
455 would take a lot of time.

- 456 3. Obtain planar B-spline surfaces for the grains (`cad.py`)

457 With the algorithms in Section 2.3, all but a very tiny grain (consisting
458 of a few pixels only) could be identified. The missing grain acts as a
459 hole in the domain.

- 460 4. Manual reparation of the CAD geometry in Salome for filling the hole

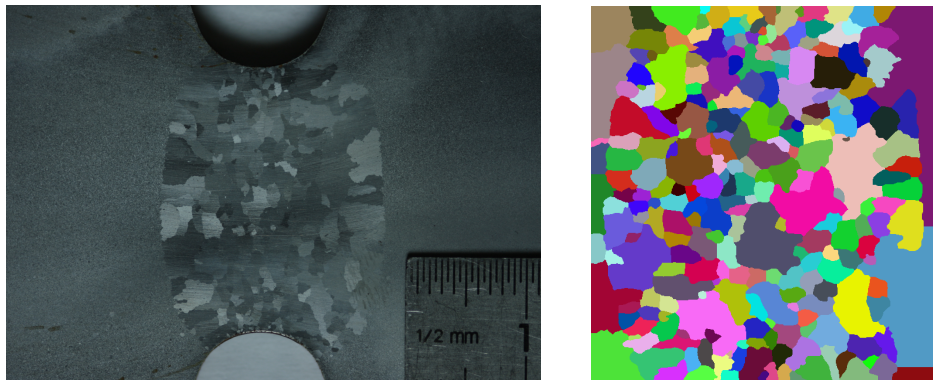
- 461 5. Mesh generation

462 Thanks to the CAD geometry, excellent mesh quality is achieved by
463 the *Netgen* mesh generator [31] without any user intervention. Figure 4
464 compares the mesh obtained by *OOF2* (Fig. 4a) with the one that is

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

- 473 6. Loading the mesh from a .med file (`med.py`)
474 The mesh is exported from Salome as a MED file. Using the `med` module
475 of `CristalX`, the triangular mesh cells for each grain and the boundary
476 nodes are extracted as NumPy arrays.
- 477 7. Mesh manipulation utilities (`geometry.py`)
478 The mesh is scaled to define the computational domain based on phys-
479 ical units instead of pixel units.
- 480 8. Abaqus input file (`abaqus.py`)
481 The material parameters, the boundary conditions and the mesh is
482 written to a text file that the finite element program *Abaqus* can inter-
483 pret.
- 484 9. Projection between a Cartesian grid and an unstructured mesh (`dic.py`)
485 The experimentally obtained displacement field, obtained by the digital
486 image correlation (DIC) technique, is available known on a Cartesian
487 grid. To compare the field values with the numerical data (available at
488 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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

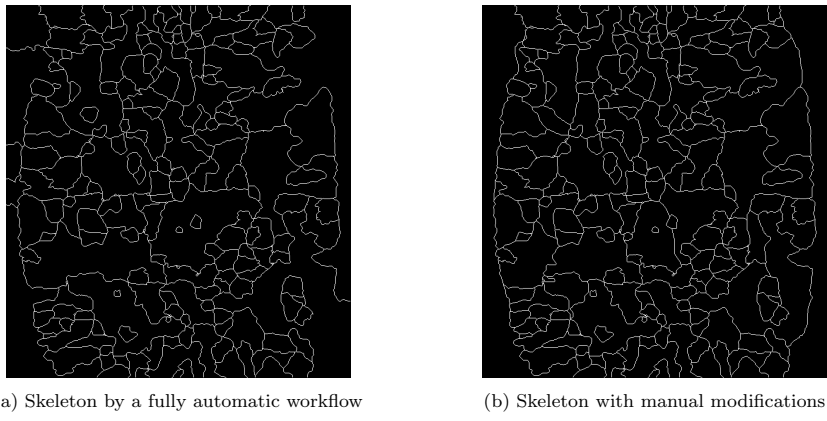


Figure 3: Skeleton of the image after the initial Quick Shift segmentation

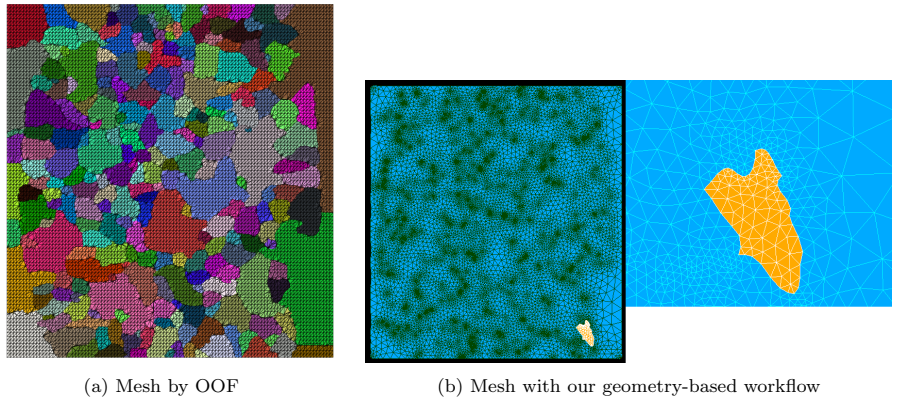


Figure 4: Generated mesh on the segmented image

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

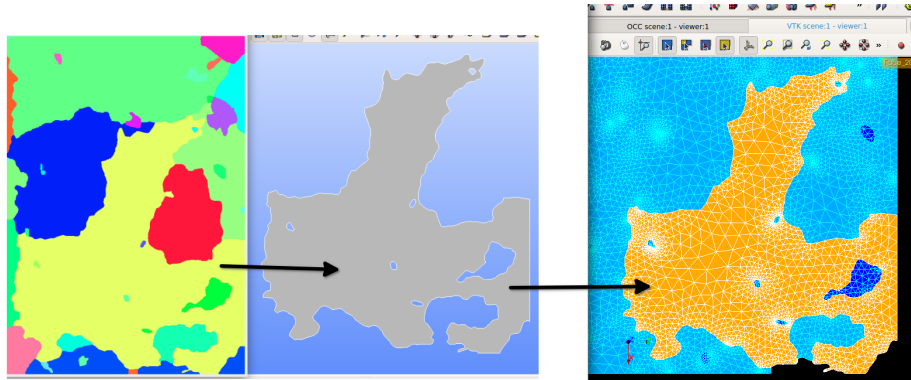


Figure 5: The image on the left shows a part of the segmented microstructure, in which the colors correspond to the grains. Each segmented region is then transformed to an explicitly given geometry, the boundaries of which are parametrized by B-splines (center). This kind of description of the geometry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

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

509 5. Conclusions and future work

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

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

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

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

527 The current version of the software is 1.0.1.

528 **Conflict of interest**

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

533 **Acknowledgments**

534 The support of Xiaolong Yan, one of the creators of *ImagePy*, is ac-
535 knowledged for giving advice on image segmentation and in particular for
536 his help with *ImagePy*. We also thank Pierre Baudoin for providing us the
537 microstructure we used to illustrate the workflow in this paper.

538 This work has been carried out within the CNRS SWIT’lab joint laboratory
539 (LaMcube CNRS 9013, LAMIH CNRS 8201, MG Valdunes company and
540 CNRS) and has also been supported by the ELSAT2020 research project.
541 SWIT’lab and ELSAT2020 are co-financed by the European Union with the
542 European Regional Development Fund, the French state and the Hauts-de-
543 France Region Council.

43 **References**

- 45 [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.
- 52 [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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- [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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- [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, Í. 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. Goullart, T. Yu, scikit-image: image processing in Python, *PeerJ* 2 (e453) (Jun. 2014). doi:10.7717/peerj.453.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

[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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- [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).