MicroStructPy Documentation

Release 1.5.6

Kenneth Hart

Sep 02, 2022

CONTENTS

1 Welcome **1**
- 1.1 Summary . . . 1
- 1.2 Examples . . . 1
- 1.3 Quick Start . . . 3
- 1.4 Publications . . . 4
- 1.5 License and Attribution . . . 4

2 Getting Started **7**
- 2.1 Download & Installation . . . 7
- 2.2 Running Demonstrations . . . 7
- 2.3 Development . . . 8

3 Examples **9**
- 3.1 Input File Introduction . . . 9
  - 3.1.1 Basic Example . . . 9
  - 3.1.2 Quality Controls . . . 12
  - 3.1.3 Size & Shape . . . 14
  - 3.1.4 Oriented Grains . . . 19
  - 3.1.5 Plot Controls . . . 22
  - 3.1.6 Culmination . . . 26
- 3.2 CLI Examples . . . 30
  - 3.2.1 Elliptical Grains . . . 30
  - 3.2.2 Minimal Example . . . 34
  - 3.2.3 Picritic Basalt . . . 35
  - 3.2.4 Two Phase 3D Example . . . 40
  - 3.2.5 Colormap . . . 48
- 3.3 Python Package Examples . . . 50
  - 3.3.1 Standard Voronoi Diagram . . . 50
  - 3.3.2 Uniform Seeding Voronoi Diagram . . . 53
  - 3.3.3 Foam . . . 56
  - 3.3.4 MicroStructPy Logo . . . 60
  - 3.3.5 Grain Neighborhoods . . . 65
  - 3.3.6 Microstructure from Image . . . 67
  - 3.3.7 Microstructure Mesh Process . . . 72

4 Command Line Guide **79**
- 4.1 Introduction . . . 79
  - 4.1.1 Using the Command Line Interface . . . 79
  - 4.1.2 Command Line Procedure . . . 79
  - 4.1.3 Example Input File . . . 80

4.1.4 Including References to Other Input Files . . . 81
4.2 <material> - Material Phases . . . 83
4.2.1 Single Material . . . 83
4.2.2 Multiple Materials . . . 83
4.2.3 Grain Size Distributions . . . 84
4.2.4 Grain Geometries . . . 86
4.2.5 Material Type . . . 90
4.2.6 Grain Position Distribution . . . 91
4.2.7 Other Material Settings . . . 92
4.3 <domain> - Microstructure Domain . . . 93
4.3.1 Box . . . 93
4.3.2 Circle . . . 94
4.3.3 Cube . . . 94
4.3.4 Ellipse . . . 95
4.3.5 Rectangle . . . 96
4.3.6 Square . . . 97
4.4 <settings> - Settings . . . 98
4.4.1 Defaults . . . 98
4.4.2 File and Console I/O . . . 99
4.4.3 Run Settings . . . 101
4.4.4 Plot Controls . . . 104

5 Python Package Guide **107**
5.1 The Standard Workflow . . . 108
5.2 Meshing Methods . . . 110
5.2.1 Laguerre-Voronoi Tessellation . . . 110
5.2.2 Unstructured Meshing . . . 110
5.3 File I/O & Plot Methods . . . 110
5.4 Phase Dictionaries . . . 111

6 Output File Formats **113**
6.1 List of Seeds . . . 113
6.2 Polygonal Mesh . . . 114
6.2.1 Text File . . . 114
6.2.2 Additional Formats . . . 115
6.3 Triangular Mesh . . . 116
6.3.1 Text File . . . 116
6.3.2 Additional Formats . . . 117

7 API **119**
7.1 microstructpy.cli . . . 119
7.2 microstructpy.geometry . . . 123
7.2.1 microstructpy.geometry.Box . . . 124
7.2.2 microstructpy.geometry.Circle . . . 125
7.2.3 microstructpy.geometry.Cube . . . 128
7.2.4 microstructpy.geometry.Ellipse . . . 129
7.2.5 microstructpy.geometry.Ellipsoid . . . 133
7.2.6 microstructpy.geometry.n_box.NBox . . . 137
7.2.7 microstructpy.geometry.n_sphere.NSphere . . . 138
7.2.8 microstructpy.geometry.Rectangle . . . 140
7.2.9 microstructpy.geometry.Sphere . . . 143
7.2.10 microstructpy.geometry.Square . . . 145
7.2.11 microstructpy.geometry.factory . . . 148
7.3 microstructpy.meshing . . . 149

7.3.1 microstructpy.meshing.PolyMesh . . . 149
7.3.2 microstructpy.meshing.RasterMesh . . . 151
7.3.3 microstructpy.meshing.TriMesh . . . 154
7.4 microstructpy.seeding . . . 156
7.4.1 microstructpy.seeding.Seed . . . 156
7.4.2 microstructpy.seeding.SeedList . . . 158
7.5 microstructpy.verification . . . 161

8 Troubleshooting **165**
8.1 Installation . . . 165
8.1.1 Missing library for pygmsh on Linux . . . 165
8.1.2 MeshPy fails to install . . . 166
8.2 Command Line Interface . . . 166
8.2.1 Command not found on Linux . . . 166
8.2.2 'tkinter' not found on Linux . . . 166
8.2.3 Program quits/segfaults while calculating Voronoi diagram . . . 167

Python Module Index **169**

Index **171**

CHAPTER

ONE

WELCOME

1.1 Summary

MicroStructPy is a microstructure mesh generator written in Python. Features of MicroStructPy include:

- 2D and 3D microstructures
- Grain size, shape, orientation, and position control
- Polycrystals, amorphous phases, and voids
- Mesh verification
- Visualizations
- Output to common file formats
- Customizable workflow

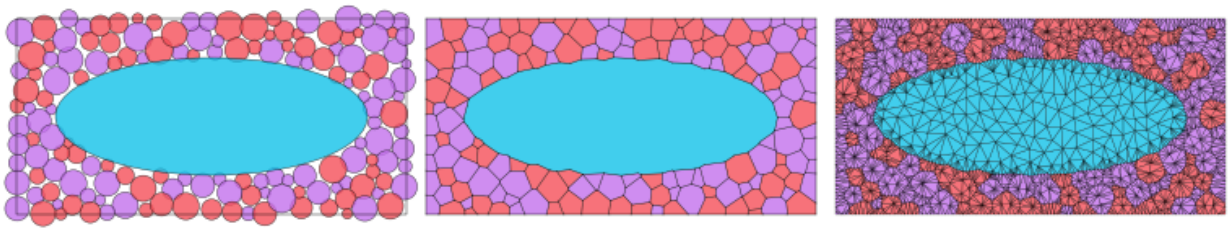

Fig. 1.1: The primary steps to create a microstructure. 1) seed the domain with particles, 2) create a Voronoi power diagram, and 3) convert the diagram into an unstructured mesh.

1.2 Examples

These images were created using MicroStructPy. For more examples, see the *Examples* section.

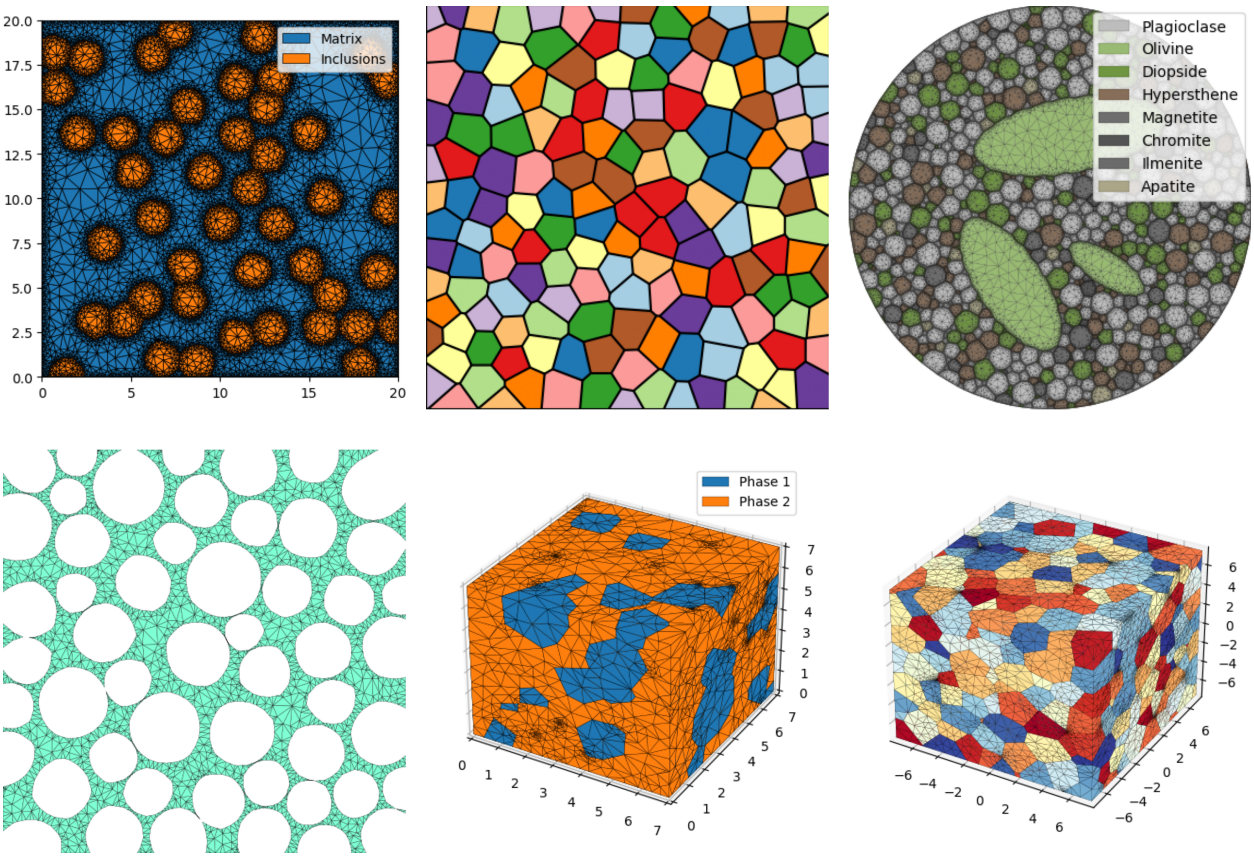

Fig. 1.2: Examples created using MicroStructPy.

1.3 Quick Start

To install MicroStructPy, download it from PyPI using:

```
pip install microstructpy
```

If there is an error with the install, try `pip install pybind11` first, then install MicroStructPy. This will create a command line executable and python package both named `microstructpy`. To use the command line interface, create a file called `input.xml` and copy this into it:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 0.15 </size>
    </material>

    <domain>
        <shape> square </shape>
    </domain>
</input>
```

Next, run the file from the command line:

```
microstructpy input.xml
```

This will produce three text files and three image files: `seeds.txt`, `polymesh.txt`, `trimesh.txt`, `seeds.png`, `polymesh.png`, and `trimesh.png`. The text files contain all of the data related to the seed geometries and meshes. The image files contain:

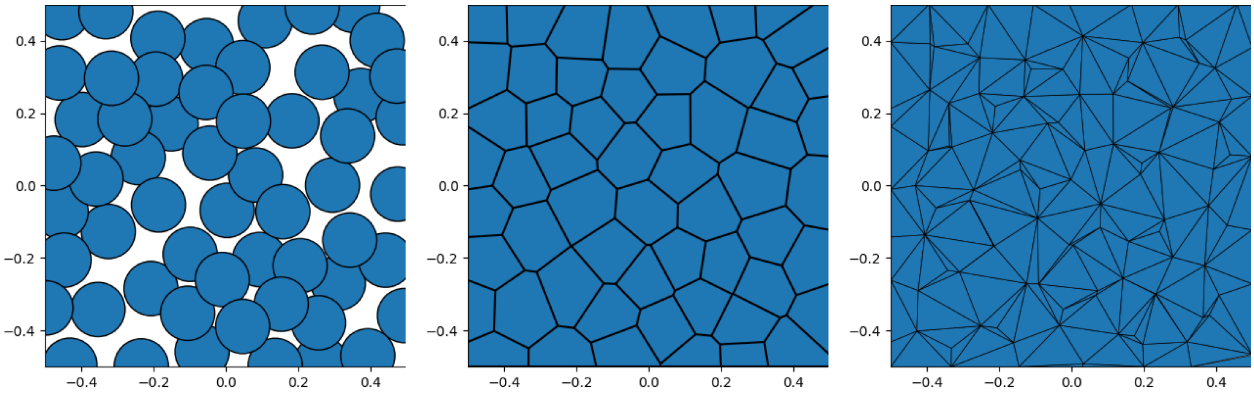

Fig. 1.3: The output plots are: 1) seed geometries, 2) polygonal mesh, and 3) triangular mesh.

The same results can be produced using this script:

```
import matplotlib.pyplot as plt
import microstructpy as msp


phase = {'shape': 'circle', 'size': 0.15}
domain = msp.geometry.Square()

```

(continues on next page)

(continued from previous page)

```
# Unpositioned list of seeds
seeds = msp.seeding.SeedList.from_info(phase, domain.area)

# Position seeds in domain
seeds.position(domain)

# Create polygonal mesh
polygon_mesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

# Create triangular mesh
triangle_mesh = msp.meshing.TriMesh.from_polymesh(polygon_mesh)

# Plot outputs
for output in [seeds, polygon_mesh, triangle_mesh]:
    plt.figure()
    output.plot(edgecolor='k')
    plt.axis('image')
    plt.axis([-0.5, 0.5, -0.5, 0.5])
    plt.show()
```

1.4 Publications

If you use MicroStructPy in you work, please consider including these citations in your bibliography:

K. A. Hart and J. J. Rimoli, Generation of statistically representative microstructures with direct grain geometry control, *Computer Methods in Applied Mechanics and Engineering*, 370 (2020), 113242. (BibTeX) (DOI)

K. A. Hart and J. J. Rimoli, MicroStructPy: A statistical microstructure mesh generator in Python, *SoftwareX*, 12 (2020), 100595. (BibTeX) (DOI)

The news article AE Doctoral Student Kenneth A. Hart Presents MicroStructPy to the World, written by the School of Aerospace Engineering at Georgia Tech, describes MicroStructPy for a general audience.

1.5 License and Attribution

MicroStructPy is open source and freely available. Copyright for MicroStructPy is held by Georgia Tech Research Corporation. MicroStructPy is a major part of Kenneth (Kip) Hart's doctoral thesis, advised by Prof. Julian Rimoli.

CHAPTER

TWO

GETTING STARTED

2.1 Download & Installation

To install MicroStructPy, download it from PyPI using:

```
pip install microstructpy
```

This installs both the `microstructpy` Python package and the `microstructpy` command line interface (CLI). If there is an error with the install, try to install `pybind11` first. You may need to add the `--user` flag, depending on your permissions.

To verify installation of the package, run the following commands:

```
python -c 'import microstructpy'
microstructpy --help
```

This verifies that 1) the python package has installed correctly and 2) the command line interface (CLI) has been found by the shell. If there is an issue with the CLI, the install location may not be in the PATH variable. The most likely install location is `~/.local/bin` for Mac or Linux machines. For Windows, it may be in a path similar to `~\AppData\Roaming\Python\Python36\Scripts\`.

Note: If the install fails and the last several error messages reference `pybind11`, run `pip install pybind11` first then install MicroStructPy.

2.2 Running Demonstrations

MicroStructPy comes with several demonstrations to familiarize users with its capabilities and options. A demonstration can be run from the command line by:

```
microstructpy --demo=minimal.xml
```

When a demo is run, the XML input file is copied to the current working directory. See *Examples* for a full list of available examples and demostrations.

2.3 Development

Contributions to MicroStructPy are most welcome. To download and install the source code for the project:

```
git clone https://github.com/kip-hart/MicroStructPy.git
pip install -e MicroStructPy
```

MicroStructPy uses tox to run tests and build the documentation. To perform these tests:

```
pip install tox
cd MicroStructPy
tox
```

Please use the issue and pull request features of the GitHub repository to report bugs and modify the code.

CHAPTER

THREE

EXAMPLES

The following contains examples of MicroStructPy. These examples include an introduction to the XML input files, some more advanced input files used on the CLI, and scripts that use the Python package.

3.1 Input File Introduction

3.1.1 Basic Example

XML Input File

The basename for this file is `intro_1_basic.xml`. The file can be run using this command:

```
microstructpy --demo=intro_1_basic.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Matrix </name>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>
        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
    </material>

    <material>
        <name> Inclusions </name>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <diameter> 2 </diameter>
    </material>

    <domain>
        <shape> square </shape>
```

(continues on next page)

(continued from previous page)

```
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>

    <settings>
        <verbose> True </verbose>
        <directory> intro_1_basic </directory>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a matrix, which is represented with small circles. The size and shape of matrix grain particles are not critical, since the boundaries between them will be removed before triangular meshing. The second material consists of circular inclusions with diameter 2.

Domain Geometry

The domain of the microstructure is a square with its bottom-left corner fixed to the origin. The side length is 20, which is 10x the size of the inclusions to ensure that the microstructure is statistically representative.

Settings

Many settings have been left to their defaults, with the exceptions being the verbose mode and output directory.

By default, MicroStructPy does not print status updates to the command line. Switching the verbose mode on will regularly print the status of the code.

The output directory is a filepath for writing text and image files. By default, MicroStructPy outputs texts files containing data on the seeds, polygon mesh, and triangular mesh as well as the corresponding image files, saved in PNG format.

Note: The `<directory>` field can be an absolute or relative filepath. If it is relative, outputs are written relative to the **input file**, not the current working directory.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.1 - Fig. 3.3.

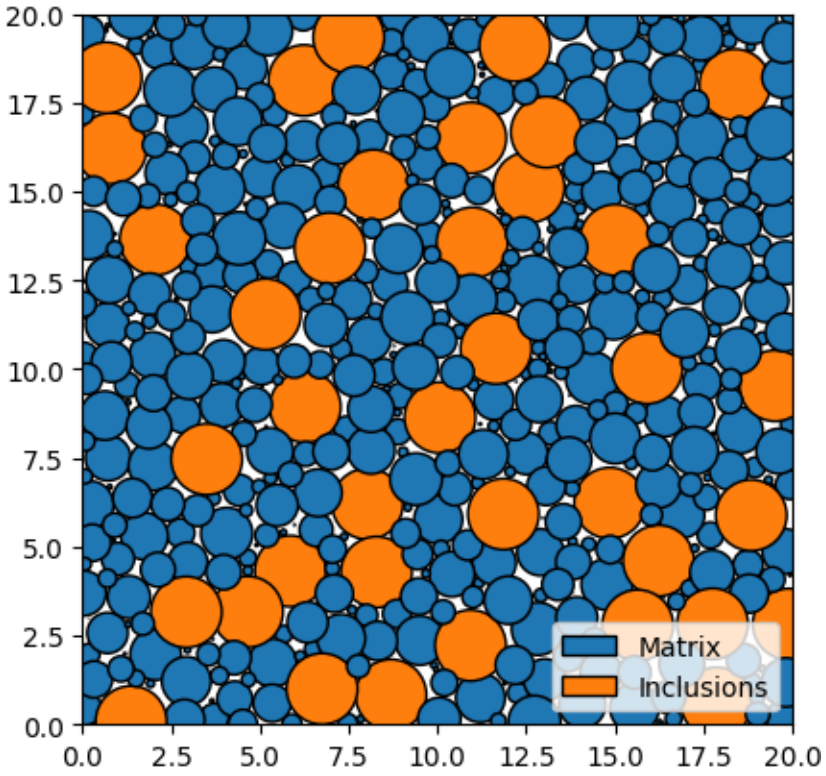

Fig. 3.1: Introduction 1 - seed geometries.

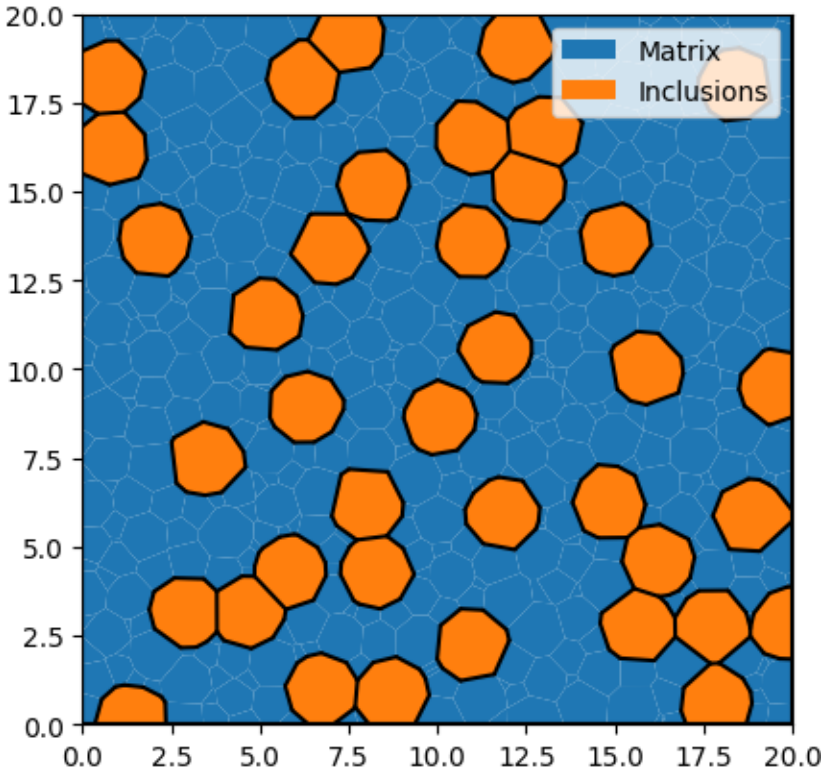

Fig. 3.2: Introduction 1 - polygonal mesh.

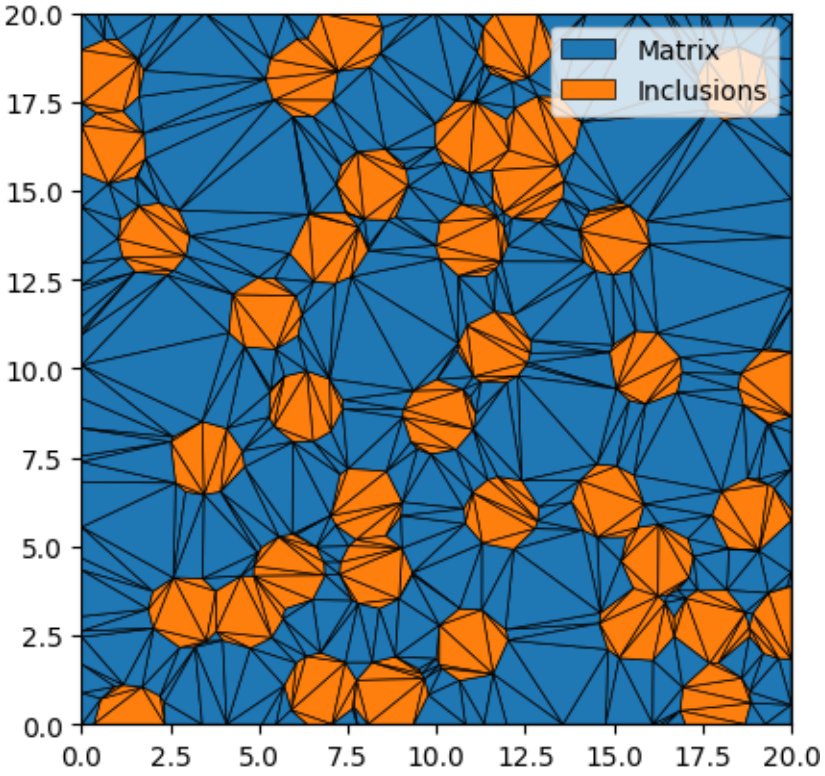

Fig. 3.3: Introduction 1 - triangular mesh.

3.1.2 Quality Controls

XML Input File

The basename for this file is `intro_2_quality.xml`. The file can be run using this command:

```
microstructpy --demo=intro_2_quality.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Matrix </name>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>

        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
    </material>

    <material>
```

(continues on next page)

(continued from previous page)

```
        <name> Inclusions </name>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <diameter> 2 </diameter>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>

    <settings>
        <directory> intro_2_quality </directory>
        <verbose> True </verbose>

        <!-- Mesh Quality Settings -->
        <mesh_min_angle> 25 </mesh_min_angle>
        <mesh_max_volume> 1 </mesh_max_volume>
        <mesh_max_edge_length> 0.1 </mesh_max_edge_length>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a matrix, which is represented with small circles. The second material consists of circular inclusions with diameter 2.

Domain Geometry

These two materials fill a square domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The side length is 20, which is 10x the size of the inclusions to ensure that microstructure is statistically representative.

Settings

The first two settings determine the output directory and whether to run the program in verbose mode. The following settings determine the quality of the triangular mesh.

The minimum interior angle of the elements is 25 degrees, ensuring lower aspect ratios compared to the first example. The maximum area of the elements is also limited to 1, which populates the matrix with smaller elements. Finally, The maximum edge length of elements at interfaces is set to 0.1, which increasing the mesh density surrounding the inclusions and at the boundary of the domain.

Note that the edge length control is currently unavailable in 3D.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.4 - Fig. 3.6.

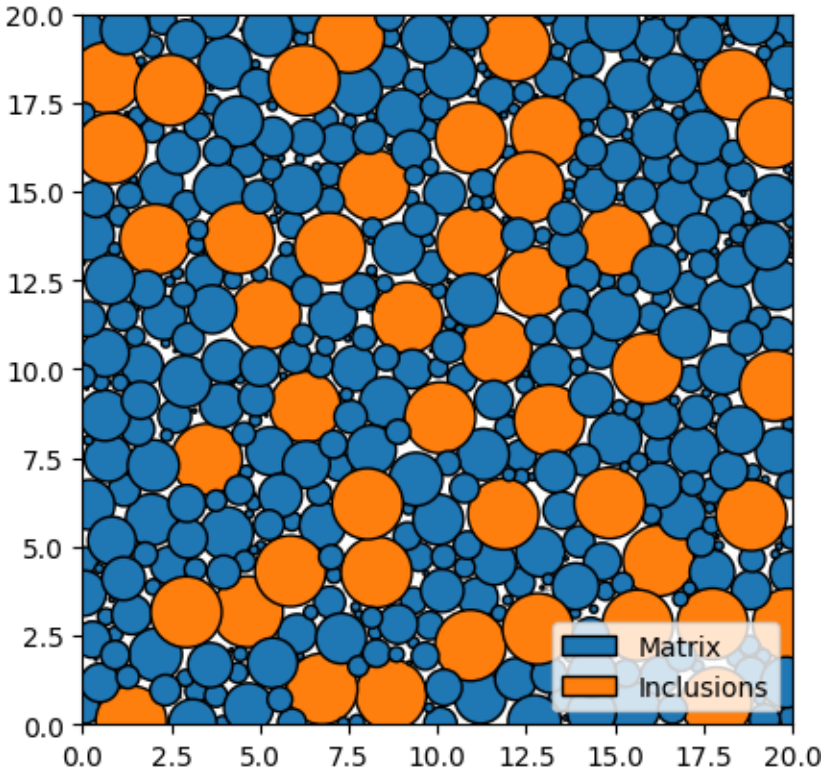

Fig. 3.4: Introduction 2 - seed geometries.

3.1.3 Size & Shape

XML Input File

The basename for this file is `intro_3_size_shape.xml`. The file can be run using this command:

```
microstructpy --demo=intro_3_size_shape.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Matrix </name>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>

        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
```

(continues on next page)

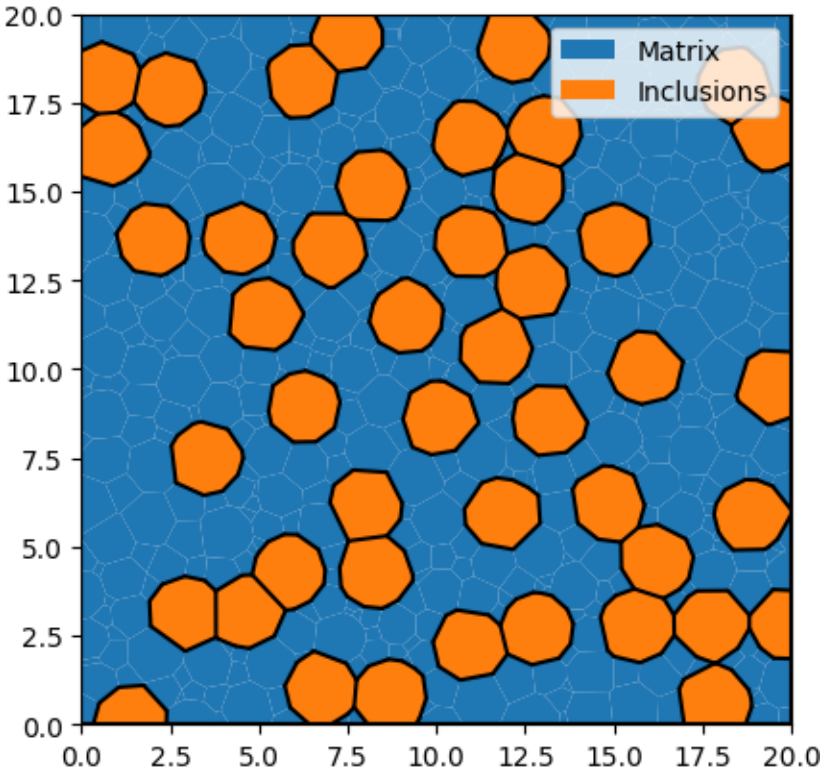

Fig. 3.5: Introduction 2 - polygonal mesh.

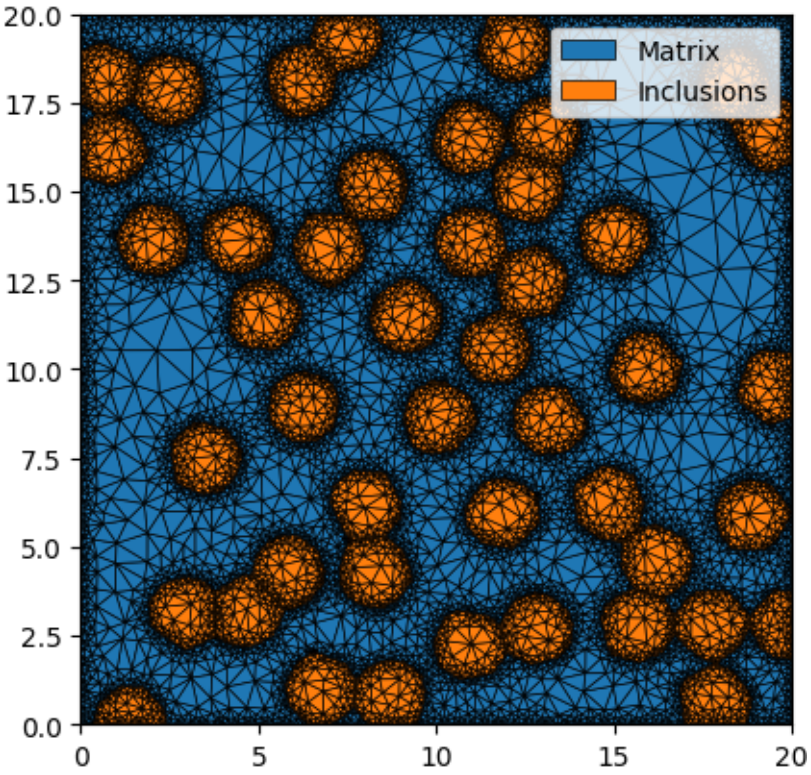

Fig. 3.6: Introduction 2 - triangular mesh.

(continued from previous page)

```
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
    </material>

    <material>
        <name> Inclusions </name>
        <fraction> 1 </fraction>
        <shape> ellipse </shape>
        <size>
            <dist_type> triang </dist_type>
            <loc> 0 </loc>
            <scale> 2 </scale>
            <c> 1 </c>
        </size>
        <aspect_ratio>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 2 </scale>
        </aspect_ratio>
        <angle> random </angle>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>

    <settings>
        <directory> intro_3_size_shape </directory>
        <verbose> True </verbose>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a matrix, which is represented with small circles.

The second material consists of elliptical inclusions with size ranging from 0 to 2 and aspect ratio ranging from 1 to 3. Note that the size is defined as the diameter of a circle with equivalent area. The orientation angle of the inclusions are random, specifically they are uniformly distributed from 0 to 360 degrees.

Domain Geometry

These two materials fill a square domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The side length is 20, which is 10x the size of the inclusions to ensure that the microstructure is statistically representative.

Settings

Many settings have been left to their defaults, with the exceptions being the verbose mode and output directory.

By default, MicroStructPy does not print status updates to the command line. Switching the verbose mode on will regularly print the status of the code.

The output directory is a filepath for writing text and image files. By default, MicroStructPy outputs texts files containing data on the seeds, polygon mesh, and triangular mesh as well as the corresponding image files, saved in PNG format.

Note: The `<directory>` field can be an absolute or relative filepath. If it is relative, outputs are written relative to the **input file**, not the current working directory.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.7 - Fig. 3.9.

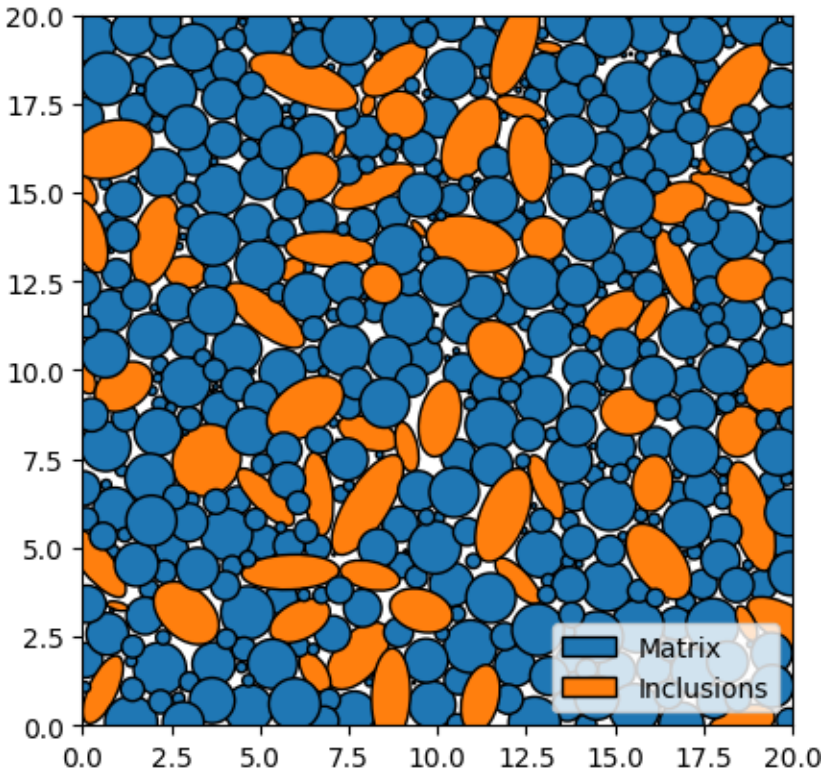

Fig. 3.7: Introduction 3 - seed geometries.

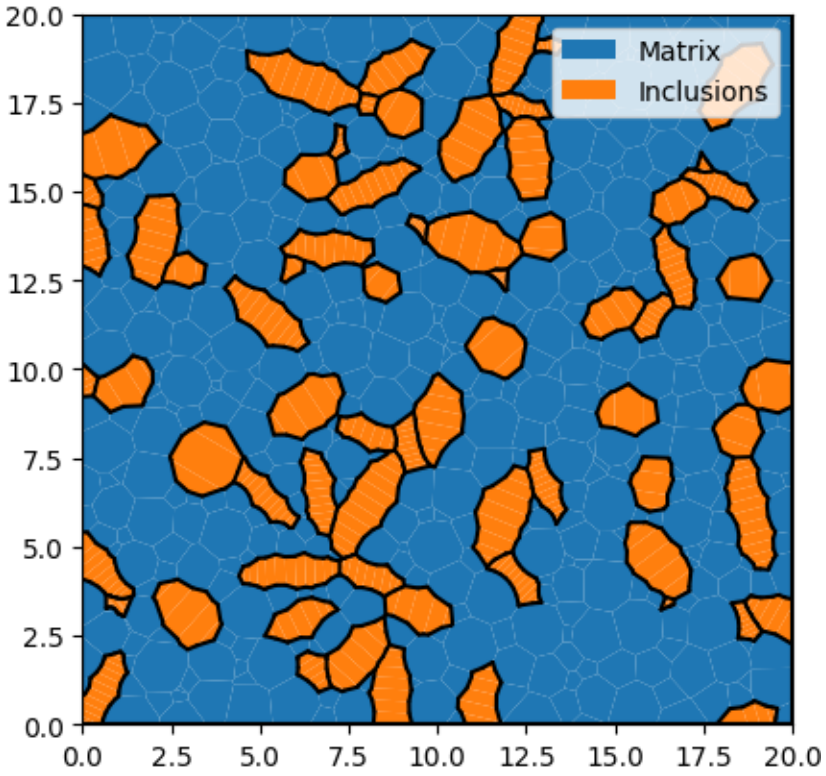

Fig. 3.8: Introduction 3 - polygonal mesh.

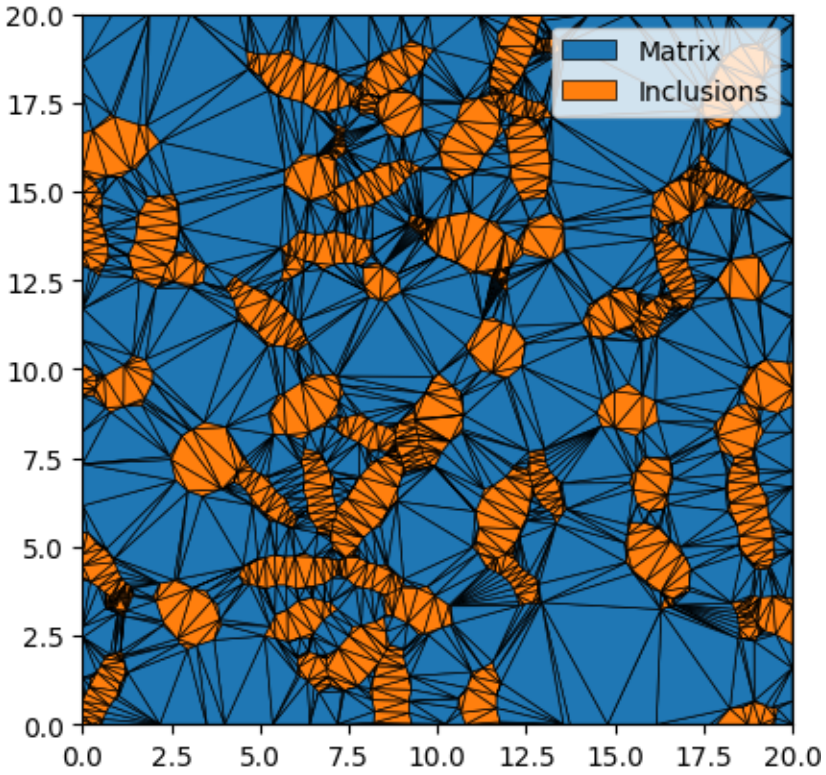

Fig. 3.9: Introduction 3 - triangular mesh.

3.1.4 Oriented Grains

XML Input File

The basename for this file is `intro_4_oriented.xml`. The file can be run using this command:

```
microstructpy --demo=intro_4_oriented.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Matrix </name>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>

        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
    </material>

    <material>
        <name> Inclusions </name>
        <fraction> 1 </fraction>
        <shape> ellipse </shape>
        <size>
            <dist_type> triang </dist_type>
            <loc> 0 </loc>
            <scale> 2 </scale>
            <c> 1 </c>
        </size>
        <aspect_ratio>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 2 </scale>
        </aspect_ratio>
        <angle_deg>
            <dist_type> uniform </dist_type>
            <loc> -10 </loc>
            <scale> 20 </scale>
        </angle_deg>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>
```

(continues on next page)

(continued from previous page)

```
    <settings>
        <directory> intro_4_oriented </directory>
        <verbose> True </verbose>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a matrix, which is represented with small circles.

The second material consists of elliptical inclusions with size ranging from 0 to 2 and aspect ratio ranging from 1 to 3. Note that the size is defined as the diameter of a circle with equivalent area. The orientation angle of the inclusions are uniformly distributed between -10 and +10 degrees, relative to the +x axis.

Domain Geometry

These two materials fill a square domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The side length is 20, which is 10x the size of the inclusions to ensure that the microstructure is statistically representative.

Settings

Many settings have been left to their defaults, with the exceptions being the verbose mode and output directory.

By default, MicroStructPy does not print status updates to the command line. Switching the verbose mode on will regularly print the status of the code.

The output directory is a filepath for writing text and image files. By default, MicroStructPy outputs texts files containing data on the seeds, polygon mesh, and triangular mesh as well as the corresponding image files, saved in PNG format.

Note: The `<directory>` field can be an absolute or relative filepath. If it is relative, outputs are written relative to the **input file**, not the current working directory.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.10 - Fig. 3.12.

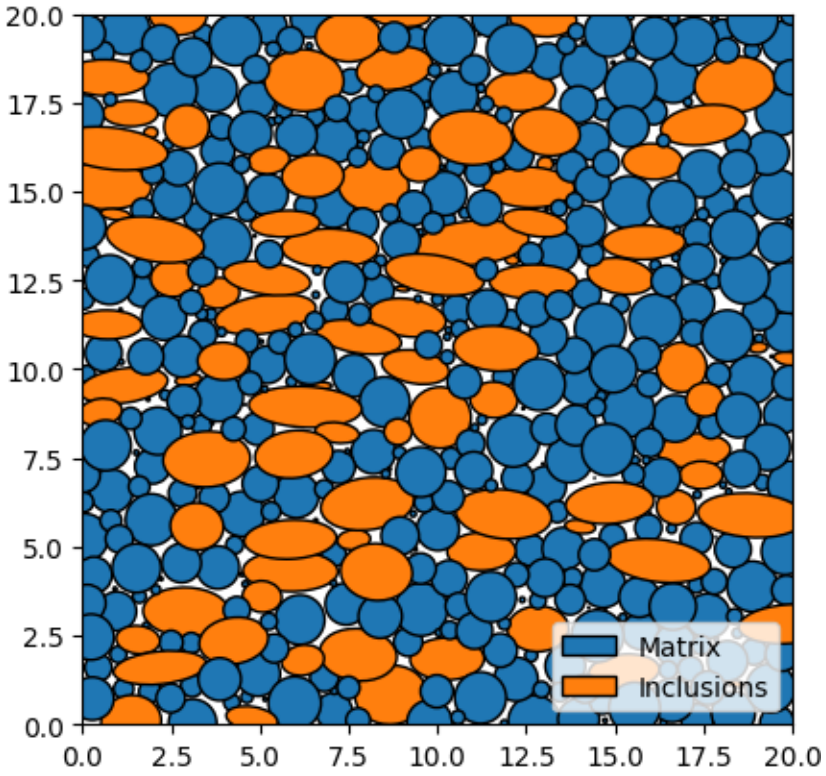

Fig. 3.10: Introduction 4 - seed geometries.

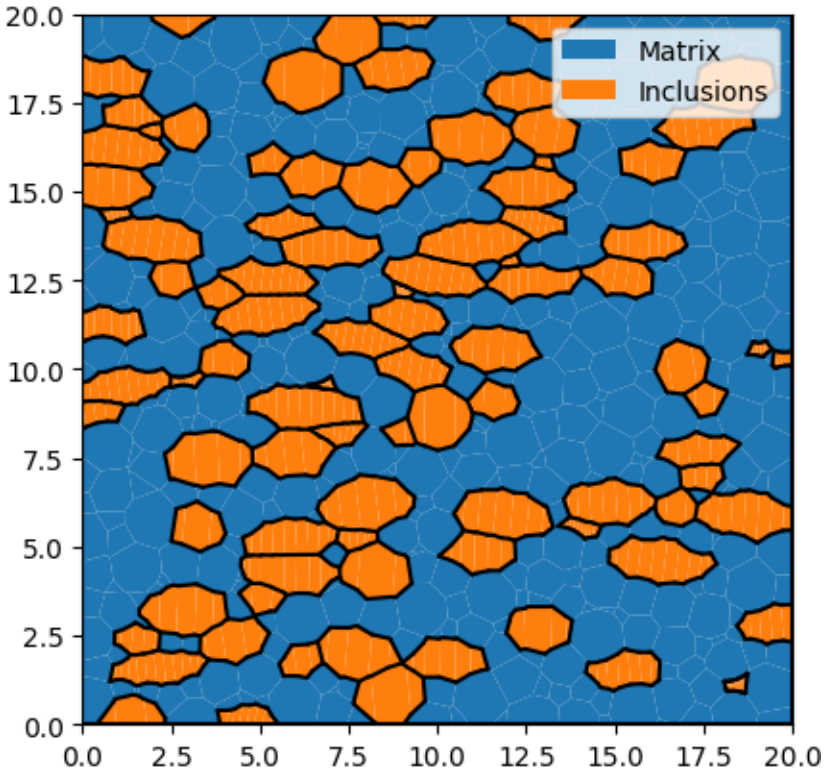

Fig. 3.11: Introduction 4 - polygonal mesh.

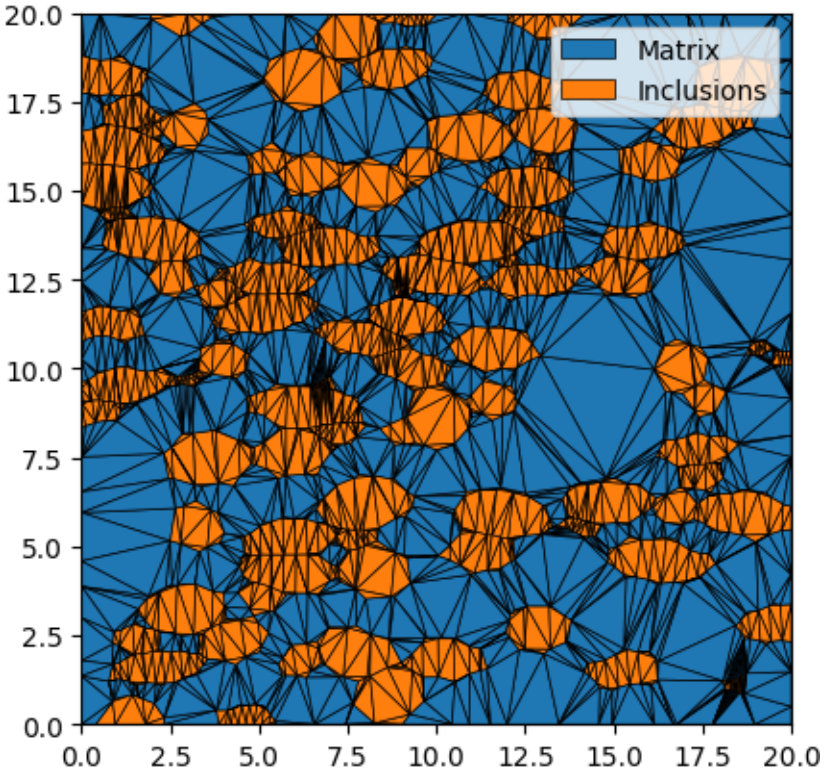

Fig. 3.12: Introduction 4 - triangular mesh.

3.1.5 Plot Controls

XML Input File

The basename for this file is `intro_5_plotting.xml`. The file can be run using this command:

```
microstructpy --demo=intro_5_plotting.xml
```

The full text of the file is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>

        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
        <color> pink </color>
    </material>

    <material>
```

(continues on next page)

(continued from previous page)

```
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <diameter> 2 </diameter>
        <color> lime </color>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>

    <settings>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png, pdf </poly_plot>
            <tri_plot> png </tri_plot>
            <tri_plot> eps </tri_plot>
            <tri_plot> pdf </tri_plot>
        </filetypes>

        <directory> intro_5_plotting </directory>
        <verbose> True </verbose>

        <seeds_kwargs>
            <alpha> 0.5 </alpha>
            <edgecolors> none </edgecolors>
        </seeds_kwargs>

        <poly_kwargs>
            <linewidth> 3 </linewidth>
            <edgecolors> #A4058F </edgecolors>
        </poly_kwargs>

        <tri_kwargs>
            <linewidth> 0.2 </linewidth>
            <edgecolor> navy </edgecolor>
        </tri_kwargs>

        <plot_axes> False </plot_axes>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a pink matrix, which is represented with small circles.

The second material consists of lime green circular inclusions with diameter 2.

Domain Geometry

These two materials fill a square domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The side length is 20, which is 10x the size of the inclusions.

Settings

PNG files of each step in the process will be output, as well as the intermediate text files. They are saved in a folder named `intro_5_plotting`, in the current directory (i.e `./intro_5_plotting`). PDF files of the poly and tri mesh are also generated, plus an EPS file for the tri mesh.

The seeds are plotted with transparency to show the overlap between them. The poly mesh is plotted with thick purple edges and the tri mesh is plotted with thin navy edges.

In all of the plots, the axes are toggles off, creating image files with minimal borders.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.13 - Fig. 3.15.

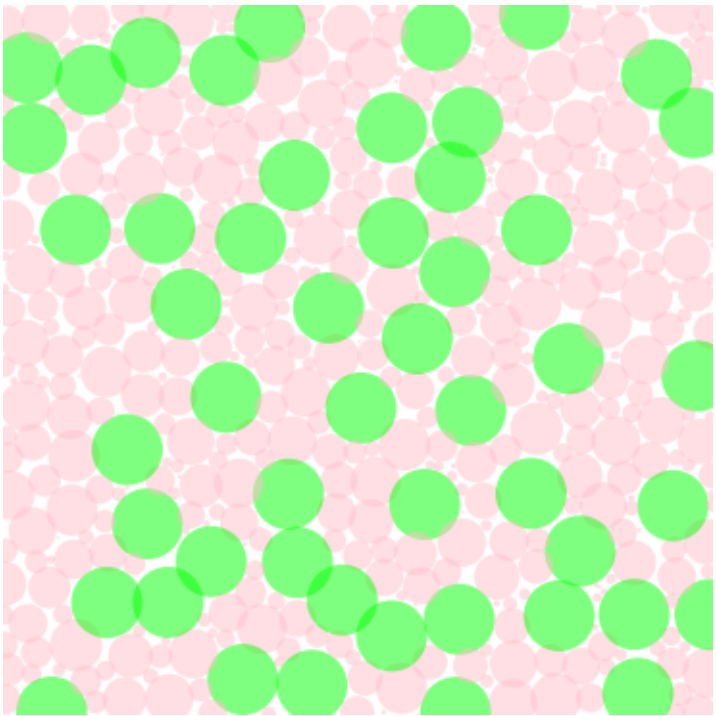

Fig. 3.13: Introduction 5 - seed geometries.

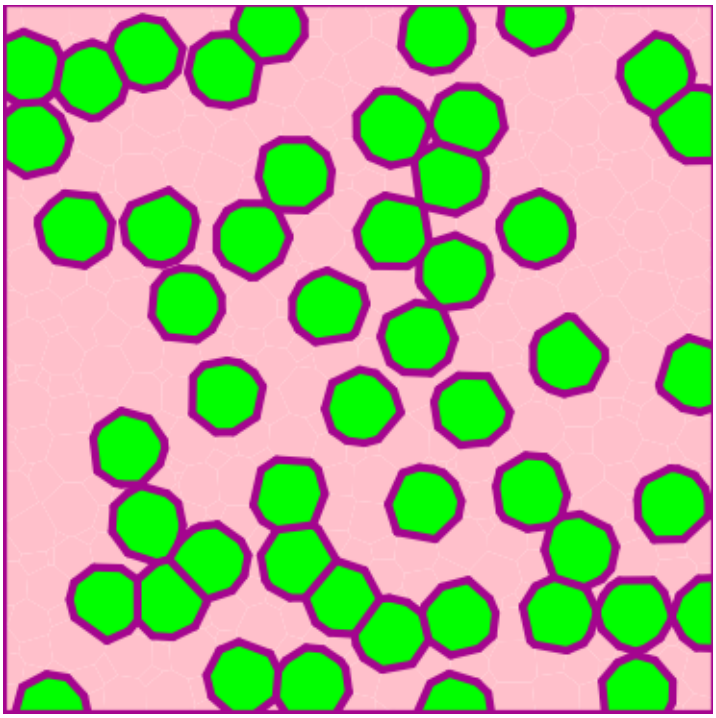

Fig. 3.14: Introduction 5 - polygonal mesh.

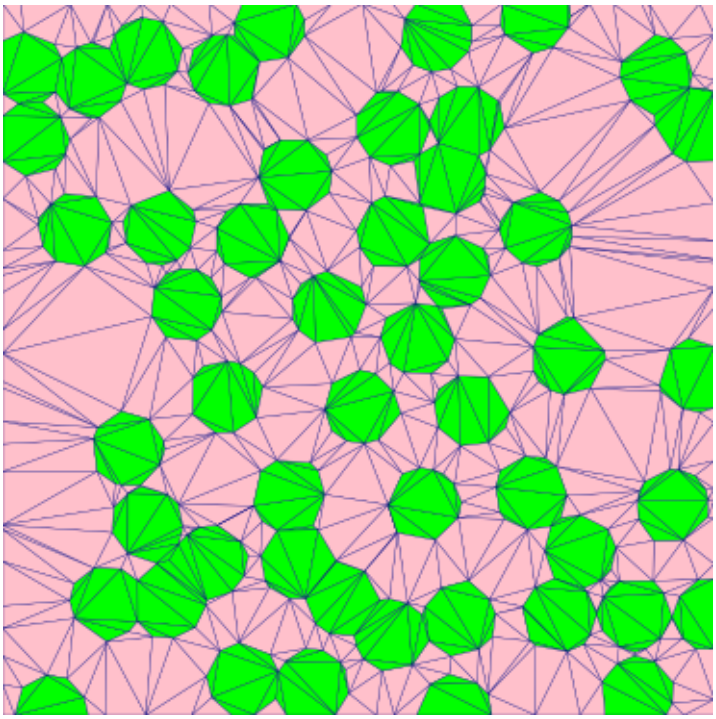

Fig. 3.15: Introduction 5 - triangular mesh.

3.1.6 Culmination

XML Input File

The basename for this file is `intro_6_culmination.xml`. The file can be run using this command:

```
microstructpy --demo=intro_6_culmination.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>

        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
        <color> pink </color>
    </material>

    <material>
        <fraction> 1 </fraction>
        <shape> ellipse </shape>
        <size>
            <dist_type> triang </dist_type>
            <loc> 0 </loc>
            <scale> 2 </scale>
            <c> 1 </c>
        </size>
        <aspect_ratio>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 2 </scale>
        </aspect_ratio>
        <angle_deg>
            <dist_type> uniform </dist_type>
            <loc> -10 </loc>
            <scale> 20 </scale>
        </angle_deg>
        <color> lime </color>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>
```

(continues on next page)

(continued from previous page)

```
    <settings>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png, pdf </poly_plot>
            <tri_plot> png </tri_plot>
            <tri_plot> eps </tri_plot>
            <tri_plot> pdf </tri_plot>
        </filetypes>

        <directory> intro_6_culmination </directory>
        <verbose> True </verbose>

        <mesh_min_angle> 25 </mesh_min_angle>
        <mesh_max_volume> 1 </mesh_max_volume>
        <mesh_max_edge_length> 0.1 </mesh_max_edge_length>

        <seeds_kwargs>
            <alpha> 0.5 </alpha>
            <edgecolors> none </edgecolors>
        </seeds_kwargs>

        <poly_kwargs>
            <linewidth> 3 </linewidth>
            <edgecolors> #A4058F </edgecolors>
        </poly_kwargs>

        <tri_kwargs>
            <linewidth> 0.2 </linewidth>
            <edgecolor> navy </edgecolor>
        </tri_kwargs>

        <plot_axes> False </plot_axes>
    </settings>
</input>
```

Materials

There are two materials, in a 2:1 ratio based on volume. The first is a pink matrix, which is represented with small circles.

The second material consists of lime green elliptical inclusions with size ranging from 0 to 2 and aspect ratio ranging from 1 to 3. Note that the size is defined as the diameter of a circle with equivalent area. The orientation angle of the inclusions are uniformly distributed between -10 and +10 degrees, relative to the +x axis.

Domain Geometry

These two materials fill a square domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The side length is 20, which is 10x the size of the inclusions.

Settings

PNG files of each step in the process will be output, as well as the intermediate text files. They are saved in a folder named `intro_5_plotting`, in the current directory (i.e `./intro_5_plotting`). PDF files of the poly and tri mesh are also generated, plus an EPS file for the tri mesh.

The seeds are plotted with transparency to show the overlap between them. The poly mesh is plotted with thick purple edges and the tri mesh is plotted with thin navy edges.

In all of the plots, the axes are toggles off, creating image files with minimal borders.

The minimum interior angle of the elements is 25 degrees, ensuring lower aspect ratios compared to the first example. The maximum area of the elements is also limited to 1, which populates the matrix with smaller elements. Finally, The maximum edge length of elements at interfaces is set to 0.1, which increasing the mesh density surrounding the inclusions and at the boundary of the domain.

Note that the edge length control is currently unavailable in 3D.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.16 - Fig. 3.18.

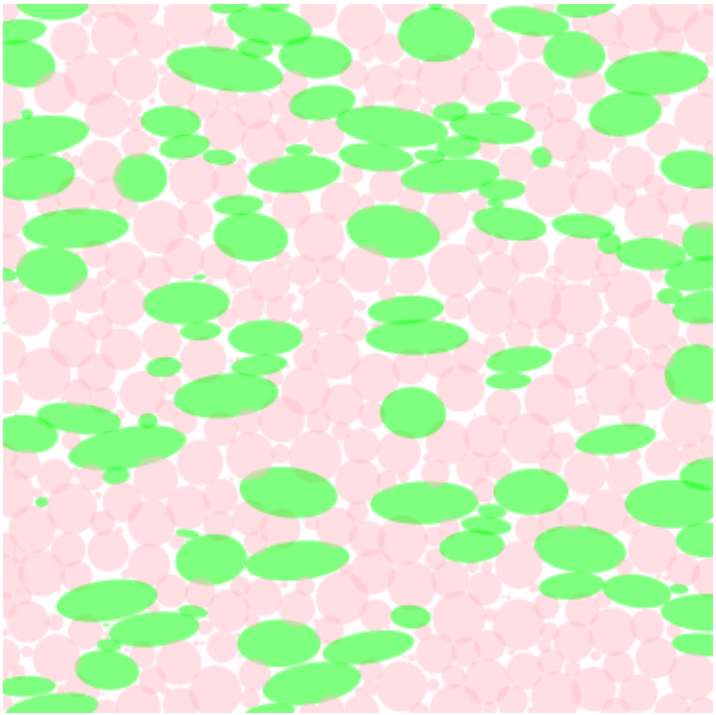

Fig. 3.16: Introduction 6 - seed geometries.

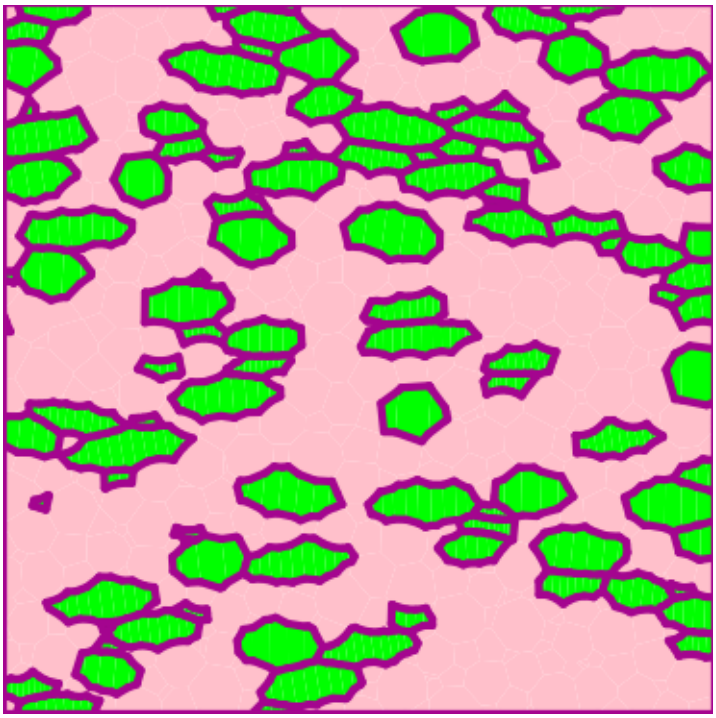

Fig. 3.17: Introduction 6 - polygonal mesh.

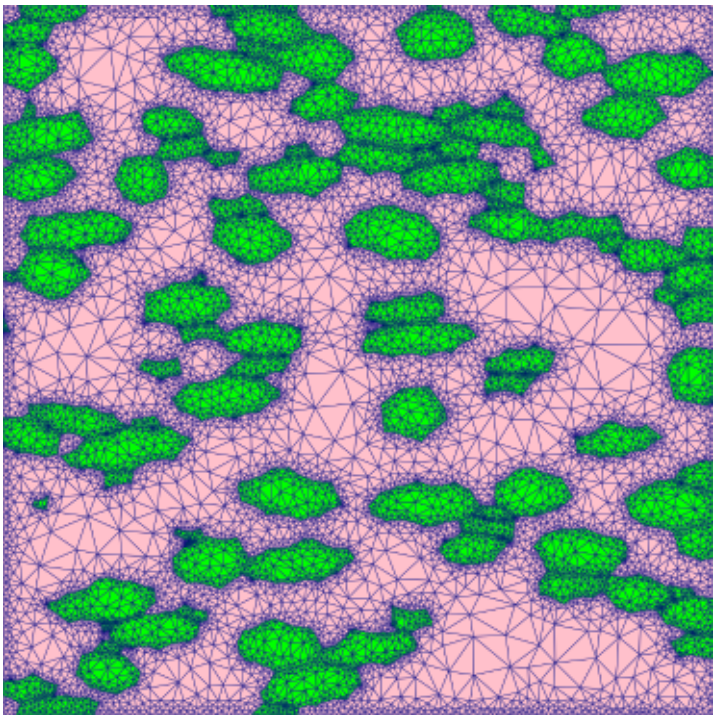

Fig. 3.18: Introduction 6 - triangular mesh.

3.2 CLI Examples

3.2.1 Elliptical Grains

XML Input File

The basename for this file is `elliptical_grains.xml`. The file can be run using this command:

```
microstructpy --demo=elliptical_grains.xml
```

The full text of the file is:

```
<?xml version="0.5" encoding="UTF-8"?>
<input>
    <material>
        <fraction> 2 </fraction>
        <shape> ellipse </shape>
        <a>
            <dist_type> uniform </dist_type>
            <loc> 0.2  </loc>
            <scale> 0.35 </scale>
        </a>
        <b> 0.05 </b>
        <angle_deg>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 20 </scale>
        </angle_deg>
        <color> orange </color>
    </material>

    <material>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <area>
            <dist_type> lognorm </dist_type>
            <scale> 0.004 </scale>
            <s> 1.0 </s>
        </area>
        <color> plum </color>
    </material>

    <material>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <area>
            <dist_type> lognorm </dist_type>
            <scale> 0.004 </scale>
            <s> 1.0 </s>
        </area>
        <color> lightblue </color>
    </material>
```

(continues on next page)

(continued from previous page)

```
    <material>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <area>
            <dist_type> lognorm </dist_type>
            <scale> 0.004 </scale>
            <s> 1.0 </s>
        </area>
        <color> lightgreen </color>
    </material>

    <material>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <area>
            <dist_type> lognorm </dist_type>
            <scale> 0.004 </scale>
            <s> 1.0 </s>
        </area>
        <color> khaki </color>
    </material>

    <domain>
        <shape> rectangle </shape>
        <side_lengths> (2.4, 1.2) </side_lengths>
    </domain>

    <settings>
        <verbose> True </verbose>
        <mesh_min_angle> 20 </mesh_min_angle>
        <mesh_max_edge_length> 0.01 </mesh_max_edge_length>
        <mesh_max_volume> 0.004 </mesh_max_volume>

        <directory> elliptical_grains </directory>
        <plot_axes> False </plot_axes>

        <seeds_kwargs>
            <linewidth> 1.0 </linewidth>
        </seeds_kwargs>

        <poly_kwargs>
            <linewidth> 1.0 </linewidth>
        </poly_kwargs>

        <tri_kwargs>
            <linewidth> 0.1 </linewidth>
        </tri_kwargs>
    </settings>
</input>
```

Materials

There are five materials, represented in equal proportions. The first material consists of ellipses and the semi-major axes are uniformly distributed, $A \sim U(0.20, 0.75)$. The semi-minor axes are fixed at 0.05, meaning the aspect ratio of these seeds are 4-15. The orientation angles of the ellipses are uniform random in distribution from 0 to 20 degrees counterclockwise from the +x axis.

The remaining four materials are all the same, with lognormal grain area distributions. The only difference among these materials is the color, which was done for visual effect.

Domain Geometry

The domain of the microstructure is a rectangle with side lengths 2.4 in the x-direction and 1.2 in the y-direction. The domain is centered on the origin, though the position of the domain is not relevant considering that the plot axes are switched off.

Settings

The aspect ratio of elements in the triangular mesh is controlled by setting the minimum interior angle for the elements at 20 degrees, the maximum element volume to 0.001, and the maximum edge length at grain boundaries to 0.01.

The function will output only plots of the microstructure process (no text files), and those plots are saved as PNGs. They are saved in a folder named `elliptical_grains`, in the current directory (i.e `./elliptical_grains`).

The axes are turned off in these plots, creating PNG files with minimal whitespace.

Finally, the linewiths in the seeds plot, polygonal mesh plot, and the triangular mesh plot are 0.5, 0.5, 0.1 respectively.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.19 - Fig. 3.21.

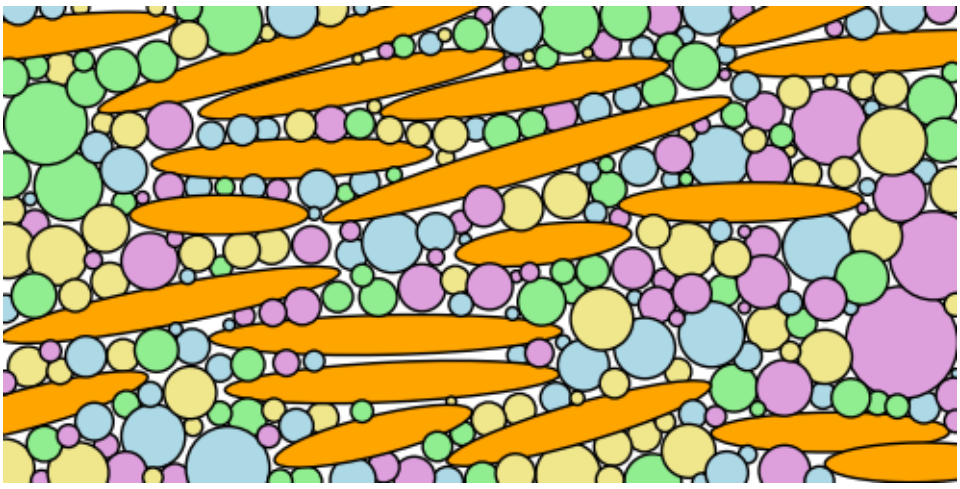

Fig. 3.19: Elliptical grain example - seed geometries.

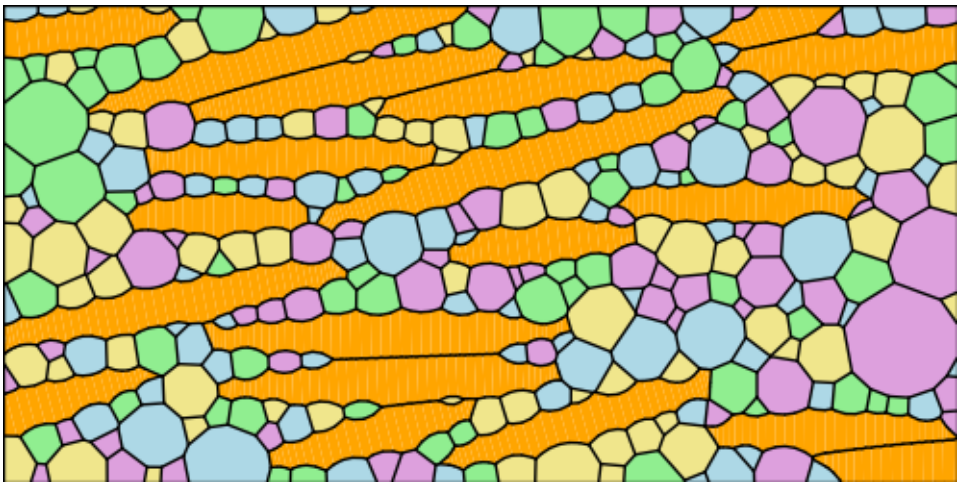

Fig. 3.20: Elliptical grain example - polygonal mesh.

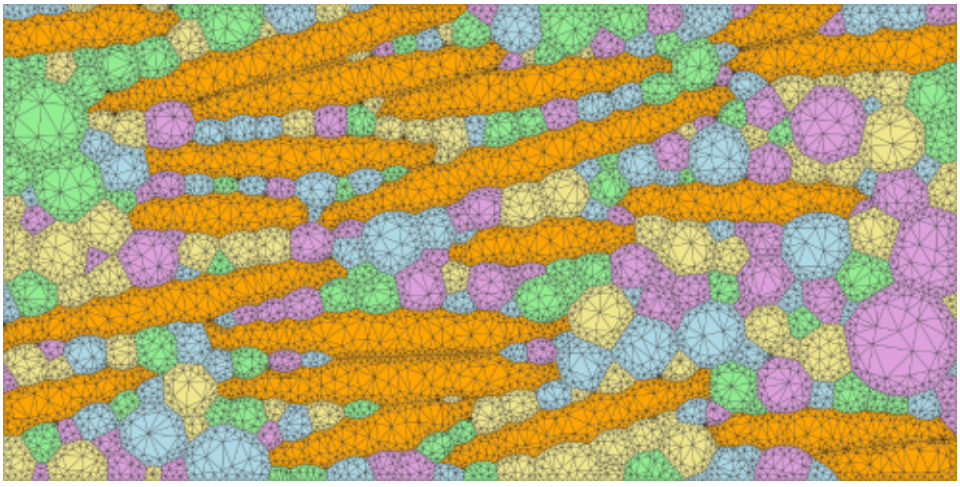

Fig. 3.21: Elliptical grain example - triangular mesh.

3.2.2 Minimal Example

XML Input File

The basename for this file is `minimal_paired.xml`. The file can be run using this command:

```
microstructpy --demo=minimal_paired.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 0.09 </size>
    </material>

    <domain>
        <shape> square </shape>
    </domain>

    <settings>
        <directory> minimal </directory>
        <plot_axes> False </plot_axes>
        <color_by> seed number </color_by>
        <colormap> Paired </colormap>
        <mesher> gmsh </mesher>
        <mesh_size> 0.03 </mesh_size>
    </settings>
</input>
```

Material

There is only one material, with a constant size of 0.09.

Domain Geometry

The material fills a square domain. The default side length is 1, meaning the domain is greater than 10x larger than the grains.

Settings

The function will output plots of the microstructure process and those plots are saved as PNGs. They are saved in a folder named `minimal`, in the current directory (i.e `./minimal`).

The axes are turned off in these plots, creating PNG files with minimal whitespace.

This example also demonstrates how to use gmsh to generate a mesh, using the `<mesher>` and `<mesh_size>` tags in the input file.

Finally, the seeds and grains are colored by their seed number, not by material.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.22 - Fig. 3.24.

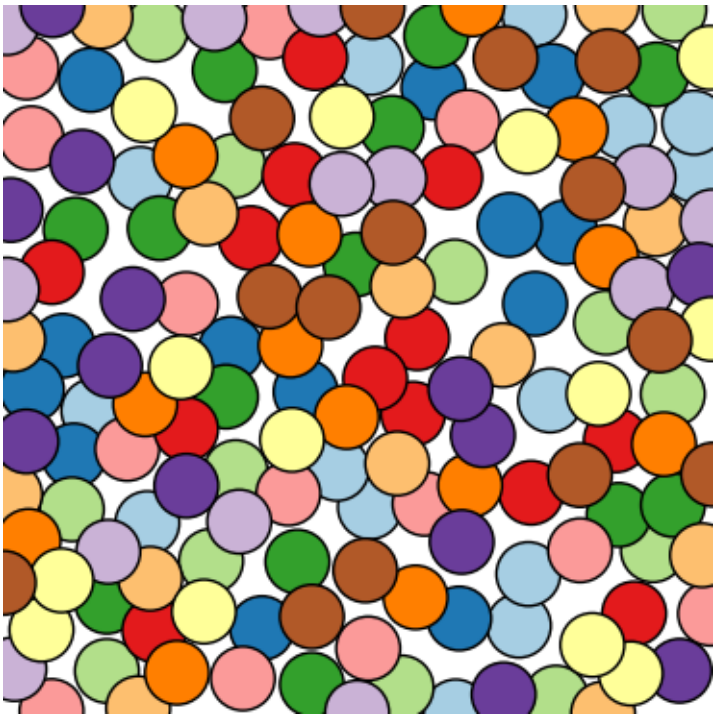

Fig. 3.22: Minimal example - seed geometries.

3.2.3 Picritic Basalt

XML Input File

The basename for this file is `basalt_circle.xml`. The file can be run using this command:

```
microstructpy --demo=basalt_circle.xml
```

The full text of the file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Plagioclase </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 45.2 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>
```

(continues on next page)

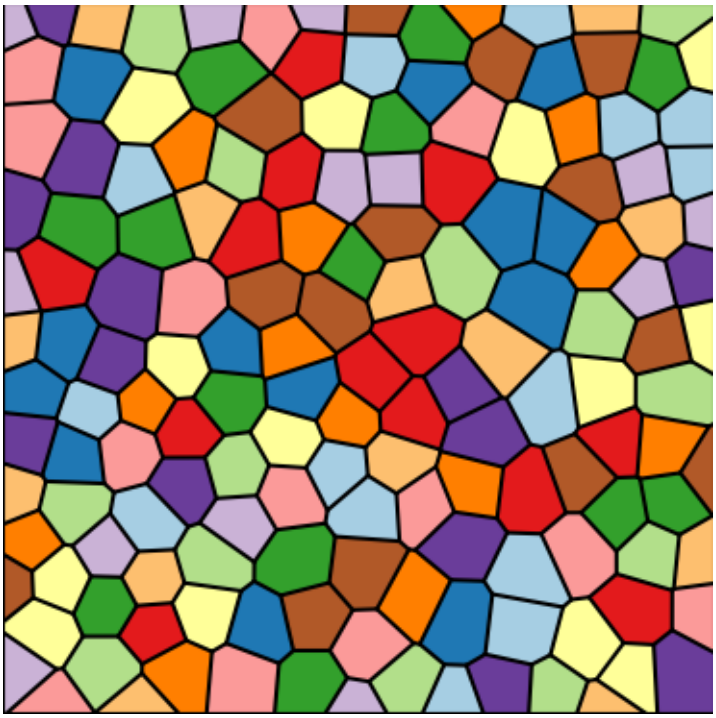

Fig. 3.23: Minimal example - polygonal mesh.

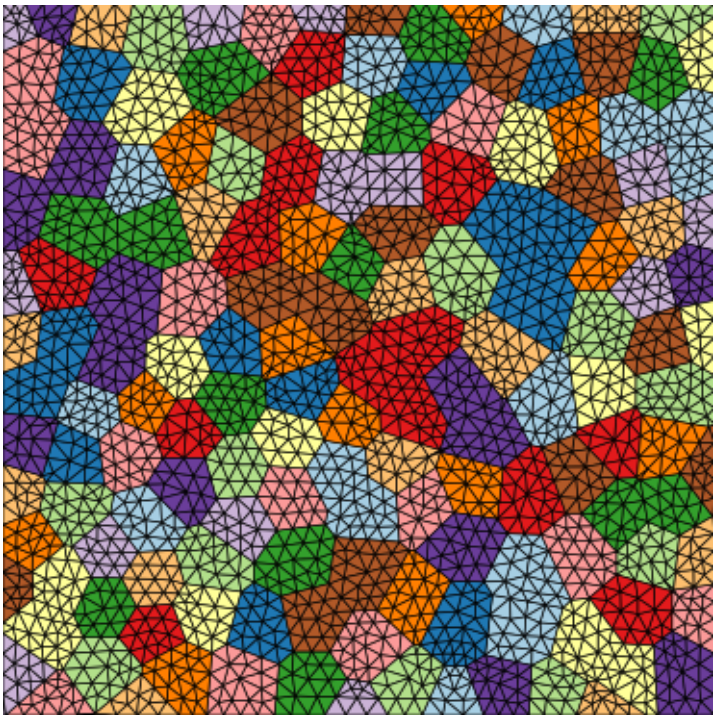

Fig. 3.24: Minimal example - triangular mesh.

(continued from previous page)

```
        </size>
        <color> #BDBDBD </color>
    </material>

    <material>
        <name> Olivine </name>
        <shape> ellipse </shape>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 19.1 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> olivine_cdf.csv </filename>
        </size>
        <aspect_ratio>
            <dist_type> uniform </dist_type>
            <loc> 1.0 </loc>
            <scale> 2.0 </scale>
        </aspect_ratio>
        <angle_deg>
            <dist_type> uniform </dist_type>
            <loc> -90 </loc>
            <scale> 180 </scale>
        </angle_deg>
        <color> #99BA73 </color>
    </material>

    <material>
        <name> Diopside </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 13.2 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>
        </size>
        <color> #709642 </color>
    </material>

    <material>
        <name> Hypersthene </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 16.6 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
```

(continues on next page)

(continued from previous page)

```
            <filename> aphanitic_cdf.csv </filename>
        </size>
        <color> #876E59 </color>
    </material>

    <material>
        <name> Magnetite </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 3.35 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>
        </size>
        <color> #6E6E6E </color>
    </material>

    <material>
        <name> Chromite </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 0.65 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>\
        </size>
        <color> #545454 </color>
    </material>

    <material>
        <name> Ilmenite </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 0.65 </loc>
            <scale> 0.2 </scale>
        </fraction>
        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>
        </size>
        <color> #6B6B6B </color>
    </material>

    <material>
        <name> Apatite </name>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 3.35 </loc>
```

(continues on next page)

(continued from previous page)

```
            <scale> 0.2 </scale>
        </fraction>

        <size>
            <dist_type> cdf </dist_type>
            <filename> aphanitic_cdf.csv </filename>
        </size>
        <color> #ABA687 </color>
    </material>

    <domain>
        <shape> circle </shape>
        <diameter> 10 </diameter>
    </domain>

    <settings>
        <directory> basalt_circle </directory>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
            <verify_plot> png </verify_plot>
        </filetypes>

        <plot_axes> False </plot_axes>
        <verbose> True </verbose>
        <verify> True </verify>

        <mesh_max_edge_length> 0.01 </mesh_max_edge_length>
        <mesh_min_angle> 20 </mesh_min_angle>
        <mesh_max_volume> 0.05 </mesh_max_volume>

        <seeds_kwargs>
            <linewidths> 0.2 </linewidths>
        </seeds_kwargs>
        <poly_kwargs>
            <linewidths> 0.2 </linewidths>
        </poly_kwargs>
        <tri_kwargs>
            <linewidths> 0.1 </linewidths>
        </tri_kwargs>
    </settings>
</input>
```

Material 1 - Plagioclase

Plagioclase composes approximately 45% of this picritic basalt sample. It is an *aphanitic* component, meaning fine-grained, and follows a custom size distribution.

Material 2 - Olivine

Olivine composes approximately 19% of this picritic basalt sample. There are large *phenocrysts* of olivine in picritic basalt, so the crystals are generally larger than the other components and have a non-circular shape. The orientation of the phenocrysts is uniform random, with the aspect ratio varying from 1 to 3 uniformly.

Materials 3-8

Diopside, hypersthene, magnetite, chromite, ilmenite, and apatie compose approximately 36% of this picritic basalt sample. They are *aphanitic* components, meaning fine-grained, and follow a custom size distribution.

Domain Geometry

These materials fill a circular domain with a diameter of 30 mm.

Settings

The function will output plots of the microstructure process and those plots are saved as PNGs. They are saved in a folder named `basalt_circle`, in the current directory (i.e `./basalt_circle`).

The axes are turned off in these plots, creating PNG files with minimal whitespace.

Mesh controls are introducted to increase grid resolution, particularly at the grain boundaries.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.25 - Fig. 3.27.

With the <`verification`> flag set to `True`, verification plots are generated by MicroStructPy. The grain size distribution comparison is given in Fig. 3.28.

Comparing the input and output distributions for olivine, it is clear that this microstructure is not statistically representative. A larger diameter for the domain would contain more grains of olivine, which would add more fidelity to the size CDF curve.

3.2.4 Two Phase 3D Example

XML Input File

The basename for this file is `two_phase_3D.xml`. The file can be run using this command:

```
microstructpy --demo=two_phase_3D.xml
```

The full text of the file is:

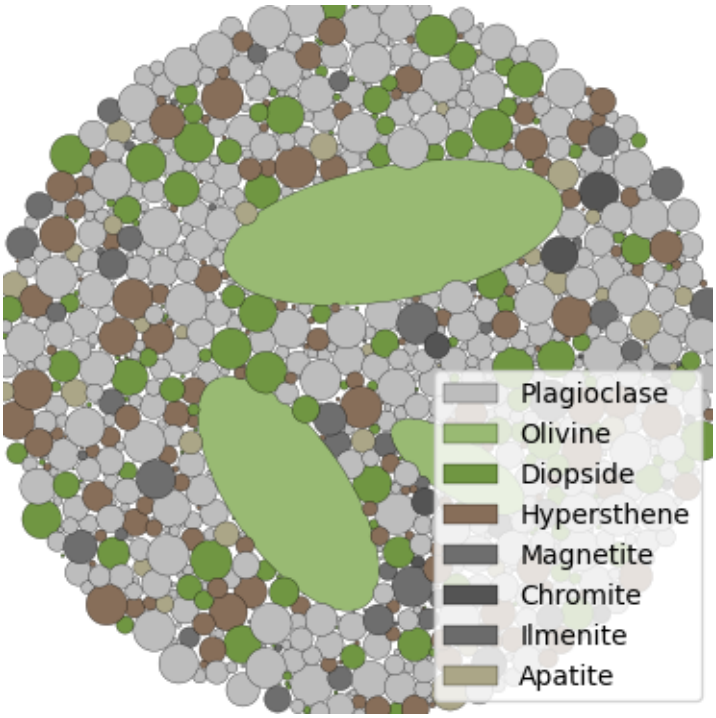

Fig. 3.25: Picritic basalt example - seed geometries

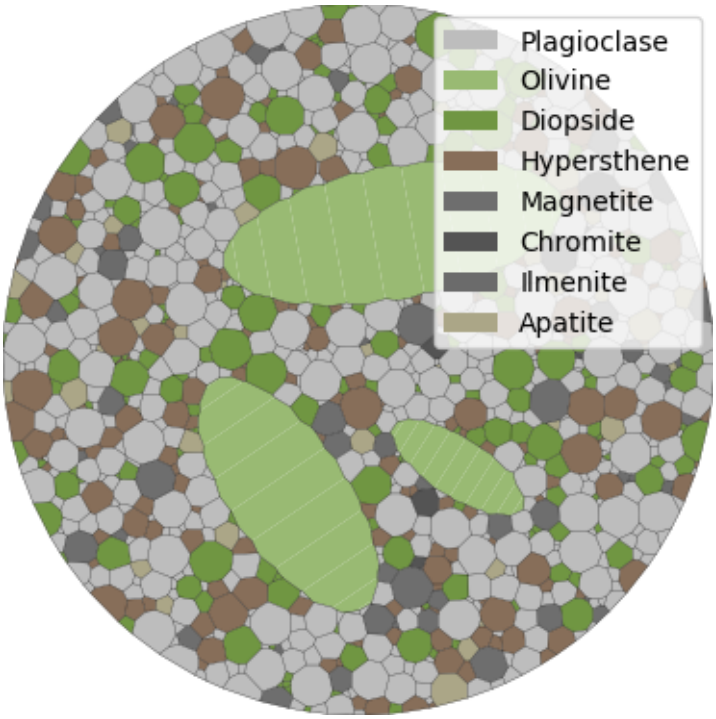

Fig. 3.26: Picritic basalt example - polygonal mesh

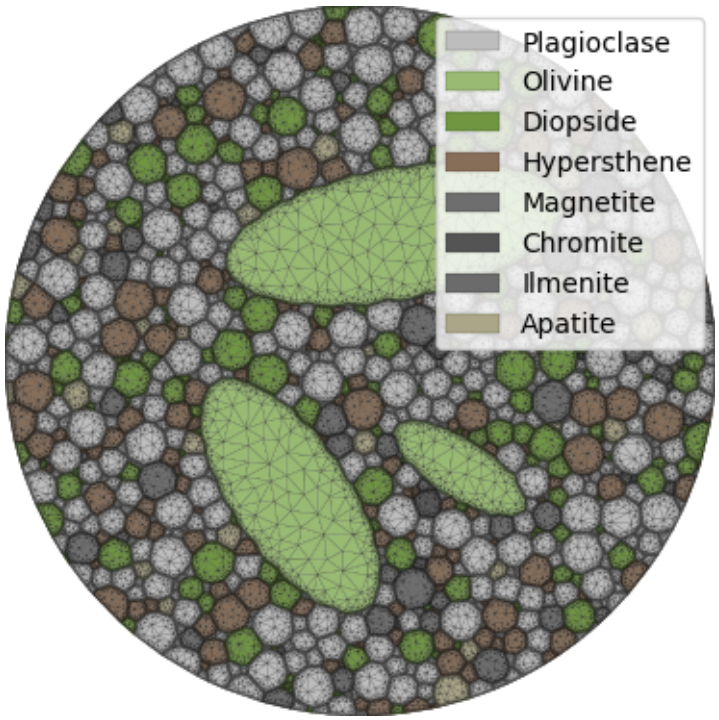

Fig. 3.27: Picritic basalt example - triangular mesh

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <fraction> 1 </fraction>
        <volume>
            <dist_type> lognorm </dist_type>
            <scale> 1 </scale>
            <s> 0.95 </s>
        </volume>
        <shape> sphere </shape>
        <name> Phase 1 </name>
    </material>

    <material>
        <fraction> 3 </fraction>
        <volume>
            <dist_type> lognorm </dist_type>
            <scale> 0.5 </scale>
            <s> 1.01 </s>
        </volume>
        <name> Phase 2 </name>
    </material>

    <domain>
        <shape> cube </shape>
        <side_length> 7 </side_length>
        <corner> (0, 0, 0) </corner>
```

(continues on next page)

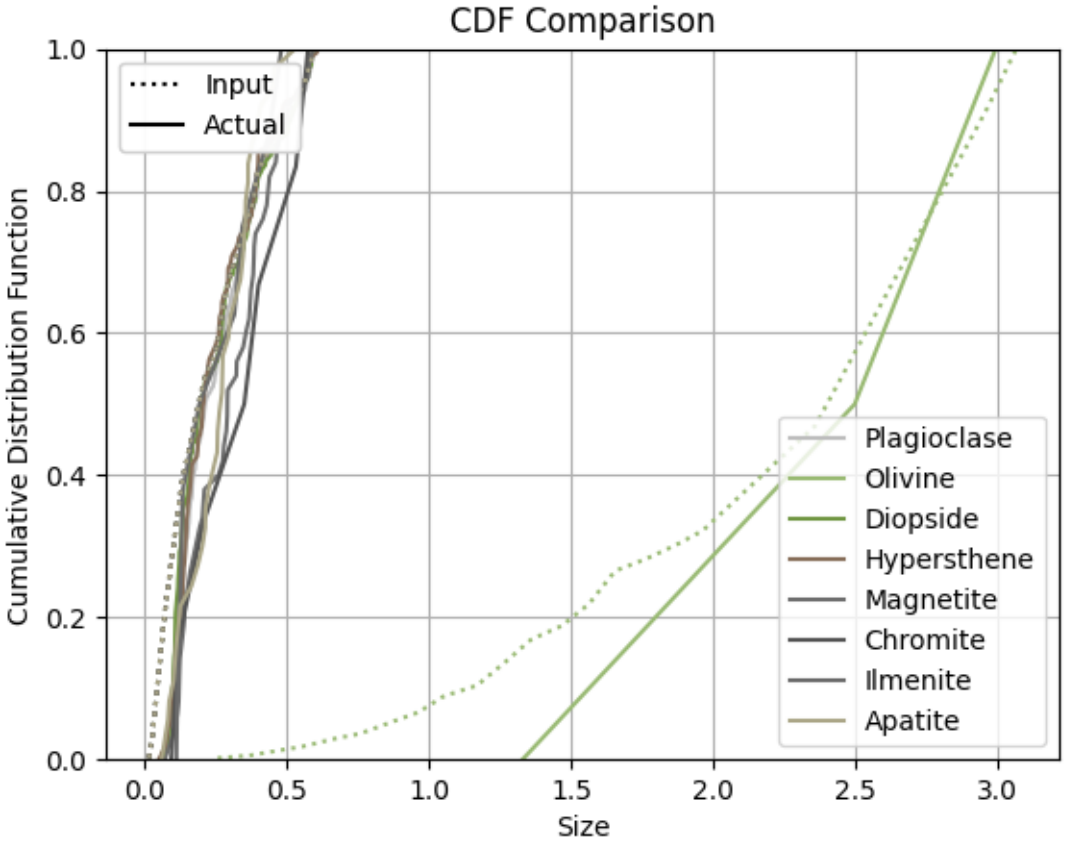

Fig. 3.28: Picritic basalt example - input and output crystal size distributions (CSDs).

(continued from previous page)

```
    </domain>

    <settings>
        <directory> two_phase_3D </directory>
        <verbose> True </verbose>

        <seeds_kwargs>
            <linewidths> 0.2 </linewidths>
        </seeds_kwargs>
        <poly_kwargs>
            <linewidths> 0.2 </linewidths>
        </poly_kwargs>
        <tri_kwargs>
            <linewidths> 0.2 </linewidths>
        </tri_kwargs>
    </settings>
</input>
```

Materials

The first material makes up 25% of the volume, with a lognormal grain volume distribution.

The second material makes up 75% of the volume, with an independent grain volume distribution.

Domain Geometry

These two materials fill a cube domain of side length 7.

Settings

The function will output plots of the microstructure process and those plots are saved as PNGs. They are saved in a folder named `two_phase_3D`, in the current directory (i.e `./two_phase_3D`).

The line width of the output plots is reduced to 0.2, to make them more visible.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.29 - Fig. 3.31.

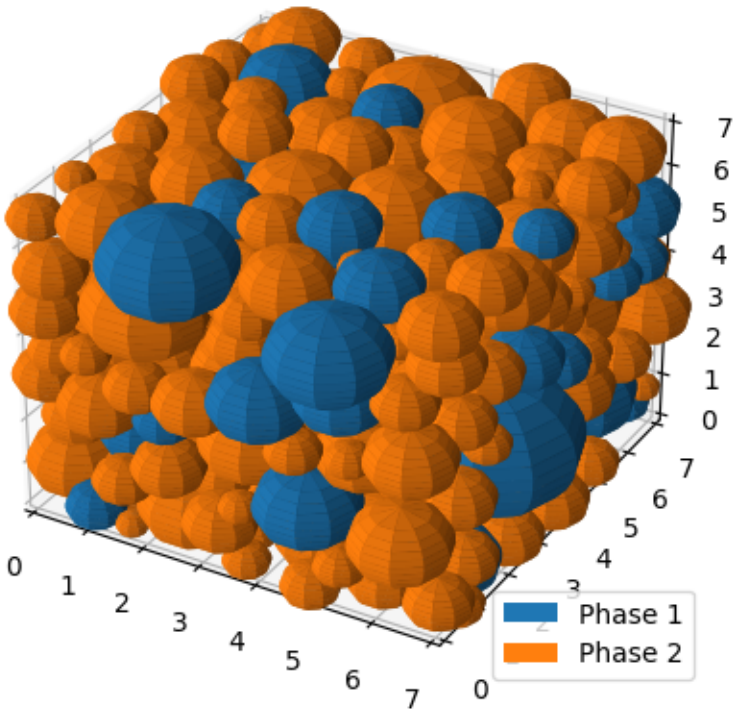

Fig. 3.29: Two phase 3D example - seed geometries.

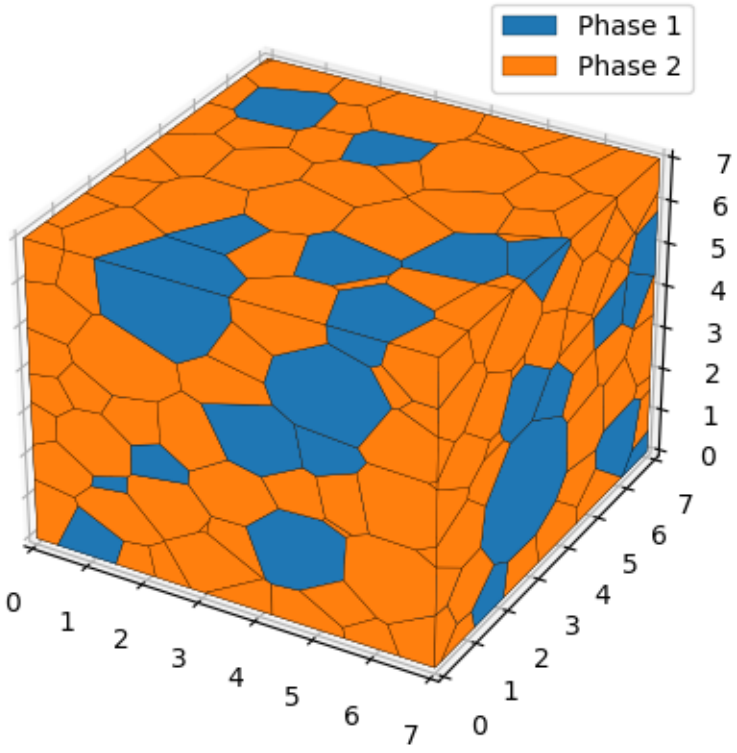

Fig. 3.30: Two phase 3D example - polygonal mesh.

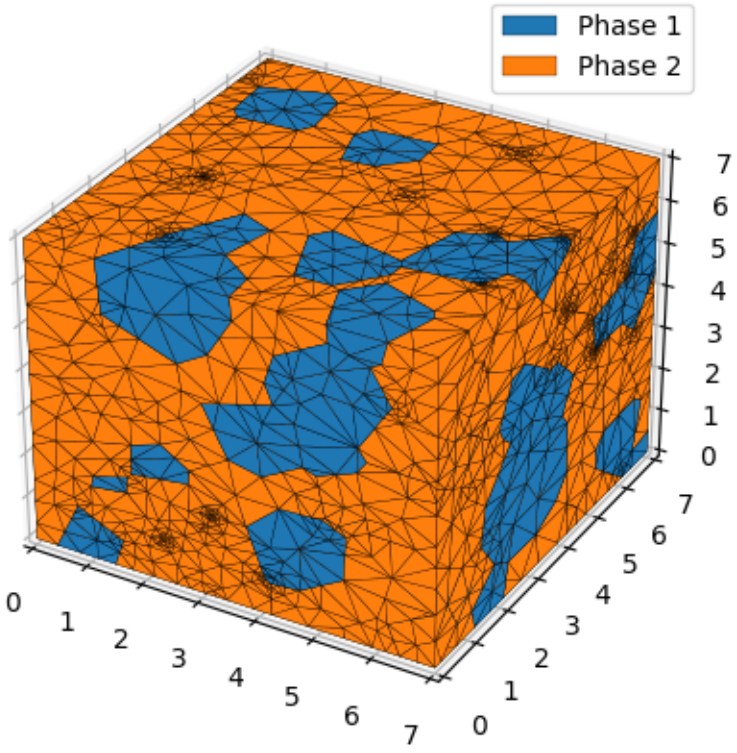

Fig. 3.31: Two phase 3D example - triangular mesh.

3.2.5 Colormap

In this example, a 3D microstructure is generated with grains colored by their seed number, rather than the material.

XML Input File

The basename for this input file is `colormap.xml`. The file can be run using this command:

```
microstructpy --demo=colormap.xml
```

The full text of the script is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> sphere </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 2 </scale>
        </size>
    </material>

    <domain>
        <shape> cube </shape>
        <side_length> 15 </side_length>
    </domain>

    <settings>
        <directory> colormap </directory>
        <mesh_min_angle> 15 </mesh_min_angle>
        <color_by> seed number </color_by>
        <colormap> RdYlBu </colormap>
    </settings>
</input>
```

Materials

There is a single material with grain sizes ranging from 1 to 3 on a uniform distribution.

Domain

The domain of the microstructure is a *Cube*. The cube's center is at the origin and its side length is 15.

Settings

The output directory is `./colormap`, which contains the text files and PNG plots of the microstructure. The minimum dihedral angle for the mesh elements is set to 15 degrees to ensure mesh quality.

The `<color_by>` option indicates how to color seeds in the output plots. There are three values available for this option: "material", "seed number", and "material number". The "material" value will use colors specified in each `<material>` field. The "seed number" value will use the seed numbers as values for a colormap. Similarly, "material number" will use the material number in a colormap.

The `<colormap>` option indicates which colormap should be used in the output plot. The default colormap is "viridis", which is also the default for matplotlib. A complete listing of available colormaps is available on the matplotlib Choosing Colormaps in Matplotlib webpage.

Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. These three plots are shown in Fig. 3.32 - Fig. 3.34.

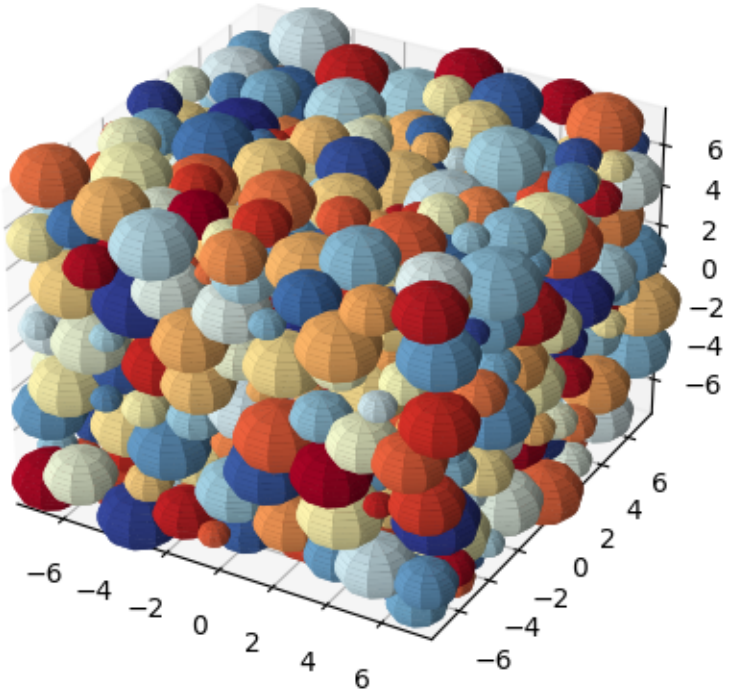

Fig. 3.32: Colormap example - seed geometries.

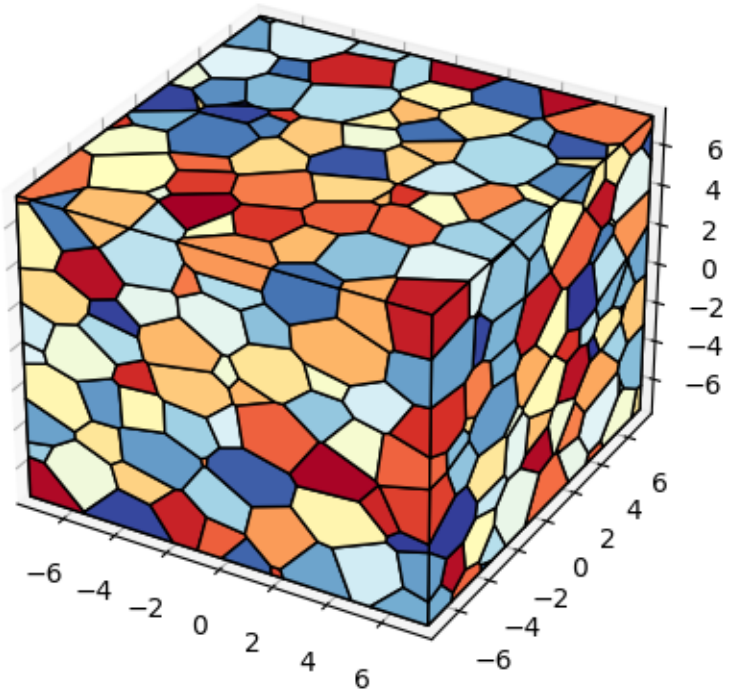

Fig. 3.33: Colormap example - polygonal mesh.

3.3 Python Package Examples

3.3.1 Standard Voronoi Diagram

Python Script

The basename for this file is `standard_voronoi.py`. The file can be run using this command:

```
microstructpy --demo=standard_voronoi.py
```

The full text of the script is:

```
import os

from matplotlib import pyplot as plt

import microstructpy as msp

# Create domain
domain = msp.geometry.Square()

# Create list of seed points
factory = msp.seeding.Seed.factory
n = 50
```

(continues on next page)

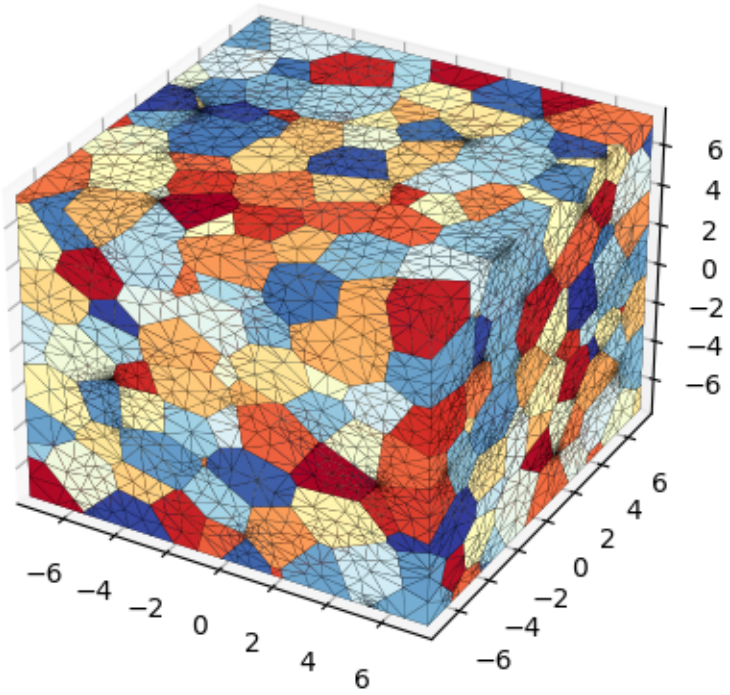

Fig. 3.34: Colormap example - triangular mesh.

(continued from previous page)

```
seeds = msp.seeding.SeedList([factory('circle', r=0.01) for i in range(n)])
seeds.position(domain)

# Create Voronoi diagram
pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

# Plot Voronoi diagram and seed points
pmesh.plot(edgecolors='k', facecolors='gray')
seeds.plot(edgecolors='k', facecolors='none')

plt.axis('square')
plt.xlim(domain.limits[0])
plt.ylim(domain.limits[1])

file_dir = os.path.dirname(os.path.realpath(__file__))
filename = os.path.join(file_dir, 'standard_voronoi/voronoi_diagram.png')
dirs = os.path.dirname(filename)
if not os.path.exists(dirs):
    os.makedirs(dirs)
plt.savefig(filename, bbox_inches='tight', pad_inches=0)
```

Domain

The domain of the microstructure is a *Square*. Without arguments, the square's center is (0, 0) and side length is 1.

Seeds

A set of 50 seed circles with small radius is initially created. Calling the *position()* method positions the points according to random uniform distributions in the domain.

Polygon Mesh

A polygon mesh is created from the list of seed points using the *from_seeds()* class method. The mesh is plotted and saved into a PNG file in the remaining lines of the script.

Plotting

The output Voronoi diagram is plotted in Fig. 3.35.

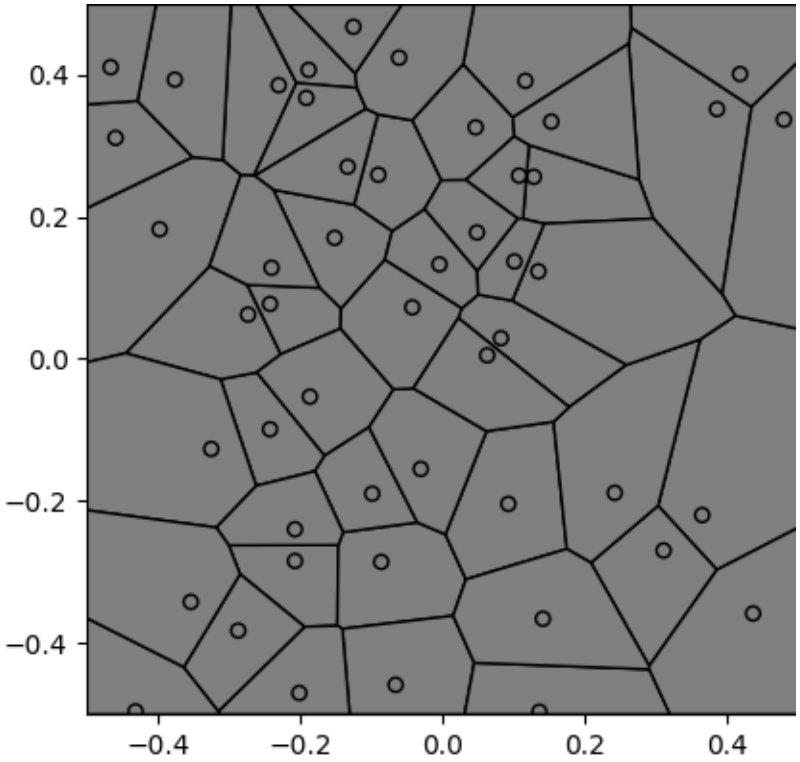

Fig. 3.35: Standard Voronoi diagram.

3.3.2 Uniform Seeding Voronoi Diagram

Python Script

The basename for this file is `uniform_seeding.py`. The file can be run using this command:

```
microstructpy --demo=uniform_seeding.py
```

The full text of the script is:

```
from __future__ import division

import os

import matplotlib as mpl
import numpy as np
from matplotlib import pyplot as plt
from scipy.spatial import distance

import microstructpy as msp

# Create domain
domain = msp.geometry.Square(corner=(0, 0))

# Create list of seed points
factory = msp.seeding.Seed.factory
n = 200
seeds = msp.seeding.SeedList([factory('circle', r=0.007) for i in range(n)])

# Position seeds according to Mitchell's Best Candidate Algorithm
np.random.seed(0)

lims = np.array(domain.limits) * (1 - 1e-5)
centers = np.zeros((n, 2))

for i in range(n):
    f = np.random.rand(i + 1, 2)
    pts = f * lims[:, 0] + (1 - f) * lims[:, 1]
    try:
        min_dists = distance.cdist(pts, centers[:i]).min(axis=1)
        i_max = np.argmax(min_dists)
    except ValueError:  # this is the case when i=0
        i_max = 0
    centers[i] = pts[i_max]
    seeds[i].position = centers[i]

# Create Voronoi diagram
pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

# Set colors based on area
areas = pmesh.volumes
std_area = domain.area / n
min_area = min(areas)
```

(continues on next page)

(continued from previous page)

```
max_area = max(areas)
cell_colors = np.zeros((n, 3))
for i in range(n):
    if areas[i] < std_area:
        f_red = (areas[i] - min_area) / (std_area - min_area)
        f_green = (areas[i] - min_area) / (std_area - min_area)
        f_blue = 1
    else:
        f_red = 1
        f_green = (max_area - areas[i]) / (max_area - std_area)
        f_blue = (max_area - areas[i]) / (max_area - std_area)
    cell_colors[i] = (f_red, f_green, f_blue)

# Create colorbar
vs = (std_area - min_area) / (max_area - min_area)
colors = [(0, (0, 0, 1)), (vs, (1, 1, 1)), (1, (1, 0, 0))]
cmap = mpl.colors.LinearSegmentedColormap.from_list('area_cmap', colors)
norm = mpl.colors.Normalize(vmin=min_area, vmax=max_area)
sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])
cb = plt.colorbar(sm, ticks=[min_area, std_area, max_area],
                  orientation='horizontal', fraction=0.046, pad=0.08)
cb.set_label('Cell Area')

# Plot Voronoi diagram and seed points
pmesh.plot(edgecolors='k', facecolors=cell_colors)
seeds.plot(edgecolors='k', facecolors='none')

plt.axis('square')
plt.xlim(domain.limits[0])
plt.ylim(domain.limits[1])

# Save diagram
file_dir = os.path.dirname(os.path.realpath(__file__))
filename = os.path.join(file_dir, 'uniform_seeding/voronoi_diagram.png')
dirs = os.path.dirname(filename)
if not os.path.exists(dirs):
    os.makedirs(dirs)
plt.savefig(filename, bbox_inches='tight', pad_inches=0)
```

Domain

The domain of the microstructure is a *Square*. Without arguments, the square's center is (0, 0) and side length is 1.

Seeds

A set of 200 seed circles with small radius is initially created. The positions of the seeds are set with Mitchell's Best Candidate Algorithm[1]. This algorithm positions seed i by sampling $i + 1$ random points and picking the one that is furthest from its nearest neighbor.

Polygon Mesh

A polygon mesh is created from the list of seed points using the `from_seeds()` class method.

Plotting

The facecolor of each polygon is determined by its area. If it is below the standard area (domain area / number of cells), then it is shaded blue. If it is above the standard area, it is shaded red. A custom colorbar is added to the figure and it is saved as a PNG, shown in Fig. 3.36.

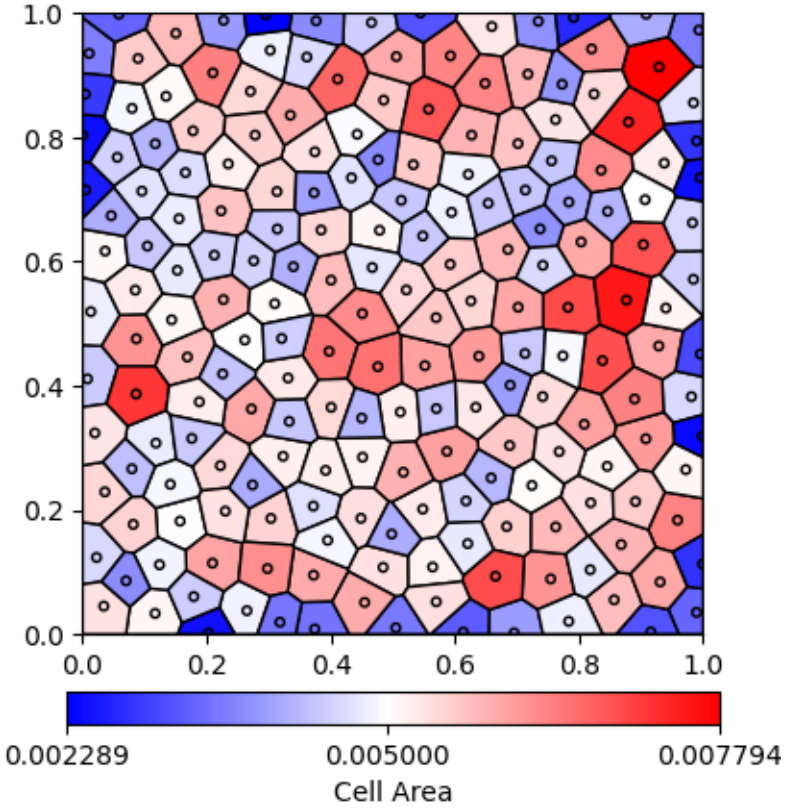

Fig. 3.36: Uniformly seeded Voronoi diagram with cells colored by area.

[1] Mitchell, T.J., "An Algorithm for the Construction of "D-Optimal" Experimental Designs," Technometrics, Vol. 16, No. 2, May 1974, pp. 203-210. (https://www.jstor.org/stable/1267940)

3.3.3 Foam

In this example, a foam microstructure is generated by first tesselating the voids, then adding foam material to the edges between the voids.

Python Script

The basename for this file is `foam.py`. The file can be run using this command:

```
microstructpy --demo=foam.py
```

The full text of the script is:

```
import os

import numpy as np
import scipy.stats
from matplotlib import pyplot as plt

import microstructpy as msp


def main():
    # Create Directory
    dirname = os.path.join(os.path.dirname(__file__), 'foam')
    if not os.path.exists(dirname):
        os.makedirs(dirname)

    # Define Domain
    domain = msp.geometry.Square(side_length=8)

    # Create Void Tessellation
    void_mat = {'material_type': 'void',
                'shape': 'circle',
                'size': scipy.stats.lognorm(scale=1, s=0.2)
                }

    void_a = 0.7 * domain.area
    void_seeds = msp.seeding.SeedList.from_info(void_mat, void_a)
    void_seeds.position(domain, rtol=0.03, verbose=True)
    void_tess = msp.meshing.PolyMesh.from_seeds(void_seeds, domain)

    # Add Foam
    foam_mat = {'material_type': 'amorphous',
                'shape': 'circle',
                'size': scipy.stats.lognorm(scale=0.15, s=0.1)
                }

    foam_a = 0.15 * domain.area
    foam_seeds = msp.seeding.SeedList.from_info(foam_mat, foam_a)
    inds = np.flip(np.argsort([s.volume for s in foam_seeds]))
    foam_seeds = foam_seeds[inds]
```

(continues on next page)

(continued from previous page)

```
    bkdwns = np.array([s.breakdown[0] for s in foam_seeds])
    np.random.seed(0)
    for i, seed in enumerate(foam_seeds):
        if i == 0:
            trial_pt = trial_position(void_tess)
        else:
            r = seed.geometry.r
            check_bkdwns = bkdwns[:i]
            good_pt = False
            while not good_pt:
                trial_pt = trial_position(void_tess)
                good_pt = check_pt(trial_pt, r, check_bkdwns)

        seed.position = trial_pt
        bkdwns[i] = seed.breakdown
        seed.phase = 1

    # Combine Results
    materials = [void_mat, foam_mat]
    seeds = void_seeds + foam_seeds
    pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

    # Triangular Mesh
    tmesh = msp.meshing.TriMesh.from_polymesh(pmesh,
                                              materials,
                                              min_angle=20,
                                              max_edge_length=0.1)

    # Plot
    tmesh.plot(facecolor='aquamarine',
               edgecolor='k',
               linewidth=0.2)

    plt.gca().set_position([0, 0, 1, 1])
    plt.axis('image')
    plt.gca().set_axis_off()
    plt.gca().get_xaxis().set_visible(False)
    plt.gca().get_yaxis().set_visible(False)

    xlim, ylim = domain.limits
    plt.axis([xlim[0], xlim[1], ylim[0], ylim[1]])

    for ext in ['png', 'pdf']:
        fname = os.path.join(dirname, 'trimesh.' + ext)
        plt.savefig(fname, bbox_inches='tight', pad_inches=0)


def pick_edge(void_tess):
    f_neighs = void_tess.facet_neighbors
    i = -1
    neighs = [-1, -1]
    while any([n < 0 for n in neighs]):
```

(continues on next page)

(continued from previous page)

```
        i = np.random.randint(len(f_neighs))
        neighs = f_neighs[i]
    facet = void_tess.facets[i]
    j = np.random.randint(len(facet))
    kp1 = facet[j]
    kp2 = facet[j - 1]
    return kp1, kp2


def trial_position(void_tess):
    kp1, kp2 = pick_edge(void_tess)
    pt1 = void_tess.points[kp1]
    pt2 = void_tess.points[kp2]

    f = np.random.rand()
    return [f * x1 + (1 - f) * x2 for x1, x2 in zip(pt1, pt2)]


def check_pt(point, r, breakdowns):
    pts = breakdowns[:, :-1]
    rads = breakdowns[:, -1]

    rel_pos = pts - point
    dist = np.sqrt(np.sum(rel_pos * rel_pos, axis=1))
    min_dist = rads + r - 0.3 * np.minimum(rads, r)
    return np.all(dist > min_dist)


if __name__ == '__main__':
    main()
```

Domain

The domain of the microstructure is a *Square*. Without arguments, the square's center is (0, 0) and side length is 15.

Seeds

Initially, the seed list is entirely voids following a lognormal size distribution. These are then tessellated to determine the boundaries between the voids. These voids are generated to fill 70% of the domain and are positioned with a custom `rtol` value of 3%. This ensures that most of the voids do not connect with each other and that the foam seeds positioned along the edges do not become consumed by the void cells.

Next, foam seeds are added to the edges between voids. These seeds are given a size distribution, however there are no foam grains since the material is amorphous. Once the foam seeds are positioned in the domain, the lists of void and foam seeds are combined into a single seed list.

Polygon Mesh

A polygon mesh is created from the list of seed points using the *from_seeds()* class method.

Triangular Mesh

A triangular mesh is created from the polygonal mesh using the *from_polymesh()* class method. The optional `phases` parameter is used in this case since the mesh contains non-crystalline materials. Additionally, the minimum interior angle of the mesh elements is set to 20 to ensure good mesh quality and the maximum edge length is set to increase mesh resolution near the voids.

Plotting

The triangular mesh in this example is plotted in aquamarine, one of several named colors in matplotlib. Next, the axes are turned off and the limits are set to equal the bounds of the domain. Finally, the triangular mesh is saved as a PNG and as a PDF, with the resulting plot shown in Fig. 3.37.

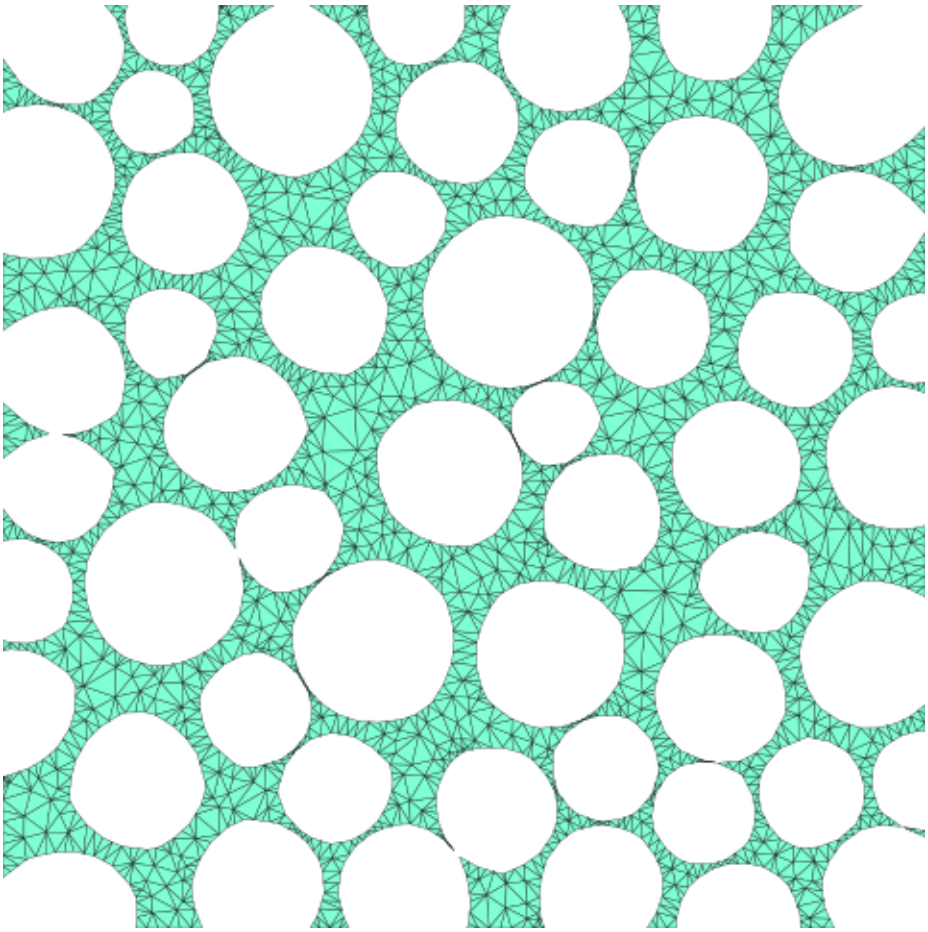

Fig. 3.37: Foam example - triangular mesh.

3.3.4 MicroStructPy Logo

Python Script

The basename for this file is `logo.py`. The file can be run using this command:

```
microstructpy --demo=logo.py
```

The full text of the script is:

```
from __future__ import division

import os

import numpy as np
from matplotlib import collections
from matplotlib import pyplot as plt
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.offsetbox import AnnotationBbox
from matplotlib.offsetbox import OffsetImage

import microstructpy as msp


def main(n_seeds, size_rng, pos_rng, k_lw):
    bkgrnd_color = 'black'
    line_color = (1, 1, 1, 1)  # white

    dpi = 300
    init_size = 2000
    logo_size = 1500
    favicon_size = 48

    logo_basename = 'logo.svg'
    favicon_basename = 'favicon.ico'
    social_basename = 'social.png'
    file_dir = os.path.dirname(os.path.realpath(__file__))
    path = os.path.join(file_dir, 'logo')
    if not os.path.exists(path):
        os.makedirs(path)
    logo_filename = os.path.join(path, logo_basename)
    pad_filename = os.path.join(path, 'pad_' + logo_basename)
    favicon_filename = os.path.join(path, favicon_basename)
    social_filename = os.path.join(path, social_basename)

    # Set Domain
    domain = msp.geometry.Circle()

    # Set Seed List
    np.random.seed(size_rng)
    rs = 0.3 * np.random.rand(n_seeds)

    factory = msp.seeding.Seed.factory
```

(continues on next page)

(continued from previous page)

```
    seeds = msp.seeding.SeedList([factory('circle', r=r) for r in rs])
    seeds.position(domain, rng_seed=pos_rng)

    # Create the Poly Mesh
    pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

    # Create and Format the Figure
    plt.clf()
    plt.close('all')
    fig = plt.figure(figsize=(init_size / dpi, init_size / dpi), dpi=dpi)
    ax = plt.Axes(fig, [0., 0., 1., 1.])
    ax.set_axis_off()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    fig.add_axes(ax)

    # Plot the Domain
    domain.plot(ec='none', fc=bkgrnd_color)

    # Plot the Facets
    facet_colors = []
    for neigh_pair in pmesh.facet_neighbors:
        if min(neigh_pair) < 0:
            facet_colors.append((0, 0, 0, 0))
        else:
            facet_colors.append(line_color)

    lw = k_lw * init_size / 100
    pmesh.plot_facets(index_by='facet', colors=facet_colors,
                      linewidth=lw, capstyle='round')

    pts = np.array(pmesh.points)
    rs = np.sqrt(np.sum(pts * pts, axis=1))
    mask = np.isclose(rs, 1)

    edges = []
    for facet in pmesh.facets:
        if np.sum(mask[facet]) != 1:
            continue

        edge = np.copy(pts[facet])
        if mask[facet[0]]:
            u = edge[0] - edge[1]
            u *= 1.1
            edge[0] = edge[1] + u
        else:
            u = edge[1] - edge[0]
            u *= 1.1
            edge[1] = edge[0] + u
        edges.append(edge)

    pc = collections.LineCollection(edges, color=line_color, linewidth=lw,
```

(continues on next page)

(continued from previous page)

```
                                    capstyle='round')
    ax.add_collection(pc)

    # Format the Plot and Convert to Image Array
    plt.axis('square')
    plt.axis(1.01 * np.array([-1, 1, -1, 1]))
    canvas = FigureCanvasAgg(fig)
    canvas.draw()

    plt_im = np.array(canvas.buffer_rgba())
    mask = plt_im[:, :, 0] > 0.5 * 255

    # Create the Logo
    logo_im = np.copy(plt_im)

    xx, yy = np.meshgrid(*[np.arange(n) for n in logo_im.shape[:2]])
    zz = - 0.2 * xx + 0.9 * yy
    ss = (zz - zz.min()) / (zz.max() - zz.min())

    c1 = [67, 206, 162]
    c2 = [24, 90, 157]

    logo_im[mask, -1] = 0  # transparent background

    # gradient
    for i in range(logo_im.shape[-1] - 1):
        logo_im[~mask, i] = (1 - ss[~mask]) * c1[i] + ss[~mask] * c2[i]

    inds = np.linspace(0, logo_im.shape[0] - 1, logo_size).astype('int')
    logo_im = logo_im[inds]
    logo_im = logo_im[:, inds]

    pad_w = logo_im.shape[0]
    pad_h = 0.5 * logo_im.shape[1]
    pad_shape = np.array([pad_w, pad_h, logo_im.shape[2]]).astype('int')
    logo_pad = np.zeros(pad_shape, dtype=logo_im.dtype)
    pad_im = np.concatenate((logo_pad, logo_im, logo_pad), axis=1)
    doc_im = np.concatenate((logo_pad, pad_im, logo_pad), axis=1)

    plt.imsave(logo_filename, logo_im, dpi=dpi)
    plt.imsave(logo_filename.replace('.svg', '.png'), np.ascontiguousarray(logo_im),␣
↪dpi=dpi)
    plt.imsave(pad_filename, pad_im, dpi=dpi)
    plt.imsave(pad_filename.replace('.svg', '.png'), np.ascontiguousarray(doc_im),␣
↪dpi=dpi)

    # Create the Favicon
    fav_im = np.copy(logo_im)
    inds = np.linspace(0, fav_im.shape[0] - 1, favicon_size).astype('int')
    fav_im = fav_im[inds]
    fav_im = fav_im[:, inds]
```

(continues on next page)

(continued from previous page)

```
    plt.imsave(favicon_filename, np.ascontiguousarray(fav_im), dpi=dpi, format='png')

    # Create the Social Banner
    fig_social, ax_social = plt.subplots()

    ax_social.set_xlim(0, 2)
    ax_social.set_ylim(0, 1)
    ax_social.set_aspect('equal')

    ax_social.set_axis_off()
    ax_social.get_xaxis().set_visible(False)
    ax_social.get_yaxis().set_visible(False)

    imagebox = OffsetImage(logo_im, zoom=0.05)
    ab = AnnotationBbox(imagebox, (1, 0.7), frameon=False)
    ax_social.add_artist(ab)
    ax_social.text(1, 0.35, 'MicroStructPy',
                   fontsize=20,
                   weight='bold',
                   horizontalalignment='center',
                   verticalalignment='center')
    ax_social.text(1, 0.23, 'Microstructure Mesh Generation in Python',
                   fontsize=10,
                   horizontalalignment='center',
                   verticalalignment='center')
    plt.draw()
    plt.savefig(social_filename, bbox_inches='tight')
    plt.close('all')


if __name__ == '__main__':
    n_seeds = 14
    size_rng = 4
    pos_rng = 7
    k_lw = 1.1

    main(n_seeds, size_rng, pos_rng, k_lw)
```

Domain

The domain of the microstructure is a *Circle*. Without arguments, the circle's center is (0, 0) and side length is 1.

Seeds

The seeds are 14 circles with radii uniformly distributed from 0 to 0.3. Calling the *position()* method positions the points according to random uniform distributions in the domain.

Polygon Mesh

A polygon mesh is created from the list of seed points using the *from_seeds()* class method. The mesh is plotted and saved into a PNG file in the remaining lines of the script.

Plot Logo

The edges in the polygonal mesh are plotted white on a black background. This image is converted into a mask and the white pixels are converted into transparent pixels. The remaining pixels are colored with a linear gradient between two colors. This image is saved in padded and tight versions, as well as a favicon for the HTML documentation. The logo that results is shown in Fig. 3.38.

Fig. 3.38: MicroStructPy logo.

3.3.5 Grain Neighborhoods

Python Script

The basename for this file is `grain_neighborhoods.py`. The file can be run using this command:

```
microstructpy --demo=grain_neighborhoods.py
```

The full text of the script is:

```
from __future__ import division

import os

import numpy as np
import scipy.integrate
import scipy.stats
from matplotlib import pyplot as plt

import microstructpy as msp

# Define the domain
domain = msp.geometry.Square(corner=(0, 0), side_length=10)

# Define the material phases
a_dist = scipy.stats.lognorm(s=1, scale=0.1)
matrix_phase = {'fraction': 1,
                'material_type': 'matrix',
                'shape': 'circle',
                'area': a_dist}

neighborhood_phase = {'fraction': 1,
                      'material_type': 'solid',
                      'shape': 'ellipse',
                      'a': 1.5,
                      'b': 0.6,
                      'angle_deg': scipy.stats.uniform(0, 360)}

phases = [matrix_phase, neighborhood_phase]

# Create the seed list
seeds = msp.seeding.SeedList.from_info(phases, domain.area)
seeds.position(domain)

# Replace the neighborhood phase with materials
a = neighborhood_phase['a']
b = neighborhood_phase['b']
r = b / 3
n = 16

t_perim = np.linspace(0, 2 * np.pi, 201)
x_perim = (a - r) * np.cos(t_perim)
y_perim = (b - r) * np.sin(t_perim)
```

(continues on next page)

(continued from previous page)

```
dx = np.insert(np.diff(x_perim), 0, 0)
dy = np.insert(np.diff(y_perim), 0, 0)
ds = np.sqrt(dx * dx + dy * dy)
arc_len = scipy.integrate.cumtrapz(ds, x=t_perim, initial=0)
eq_spaced = arc_len[-1] * np.arange(n) / n
x_pts = np.interp(eq_spaced, arc_len, x_perim)
y_pts = np.interp(eq_spaced, arc_len, y_perim)

repl_seeds = msp.seeding.SeedList()
geom = {'a': a - 2 * r, 'b': b - 2 * r}
for sn, seed in enumerate(seeds):
    if seed.phase == 0:
        repl_seeds.append(seed)
    else:
        center = seed.position
        theta = seed.geometry.angle_rad

        geom['angle_rad'] = theta
        geom['center'] = center
        core_seed = msp.seeding.Seed.factory('ellipse', phase=3,
                                             position=seed.position, **geom)
        repl_seeds.append(core_seed)

        x_ring = center[0] + x_pts * np.cos(theta) - y_pts * np.sin(theta)
        y_ring = center[1] + x_pts * np.sin(theta) + y_pts * np.cos(theta)
        for i in range(n):
            phase = 1 + (i % 2)
            center = (x_ring[i], y_ring[i])
            ring_geom = {'center': center, 'r': r}
            ring_seed = msp.seeding.Seed.factory('circle', position=center,
                                                 phase=phase, **ring_geom)
            if domain.within(center):
                repl_seeds.append(ring_seed)

# Create polygon and triangle meshes
pmesh = msp.meshing.PolyMesh.from_seeds(repl_seeds, domain)
phases = [{'material_type': 'solid'} for i in range(4)]
phases[0]['material_type'] = 'matrix'
tmesh = msp.meshing.TriMesh.from_polymesh(pmesh, phases, min_angle=20,
                                          max_volume=0.1)

# Plot triangle mesh
colors = ['C' + str(repl_seeds[att].phase) for att in tmesh.element_attributes]
tmesh.plot(facecolors=colors, edgecolors='k', linewidth=0.2)

plt.axis('square')
plt.xlim(domain.limits[0])
plt.ylim(domain.limits[1])

file_dir = os.path.dirname(os.path.realpath(__file__))
filename = os.path.join(file_dir, 'grain_neighborhoods/trimesh.png')
dirs = os.path.dirname(filename)
```

(continues on next page)

(continued from previous page)

```
if not os.path.exists(dirs):
    os.makedirs(dirs)
plt.savefig(filename, bbox_inches='tight', pad_inches=0)
```

Domain

The domain of the microstructure is a *microstructpy.geometry.Rectangle*, with the bottom left corner at the origin and side lengths of 8 and 12.

Phases

There are initially two phases: a matrix phase and a neighborhood phase. The neighborhood phase will be broken down into materials later. The matrix phase occupies two thirds of the domain, while the neighborhoods occupy one third.

Seeds

The seeds are generated to fill 1.1x the area of the domain, to account for overlap with the boundaries. They are positioned according to random uniform distributions.

Neighborhood Replacement

The neighborhood seeds are replaced by a set of three different materials. One material occupies the center of the neighborhood, while the other two alternate in a ring around the center.

Polygon and Triangle Meshing

The seeds are converted into a triangular mesh by first constructing a polygon mesh. Each material is solid, except for the first which is designated as a matrix phase. Mesh quality controls are specified to prevent high aspect ratio triangles.

Plotting

The triangular mesh is plotted and saved to a file. Each triangle is colored based on its material phase, using the standard matplotlib colors: C0, C1, C2, etc. The output PNG file is shown in Fig. 3.39.

3.3.6 Microstructure from Image

Note: The open source and freely available software package OOF is better equiped to create unstructured meshes from images.

https://www.ctcms.nist.gov/oof/

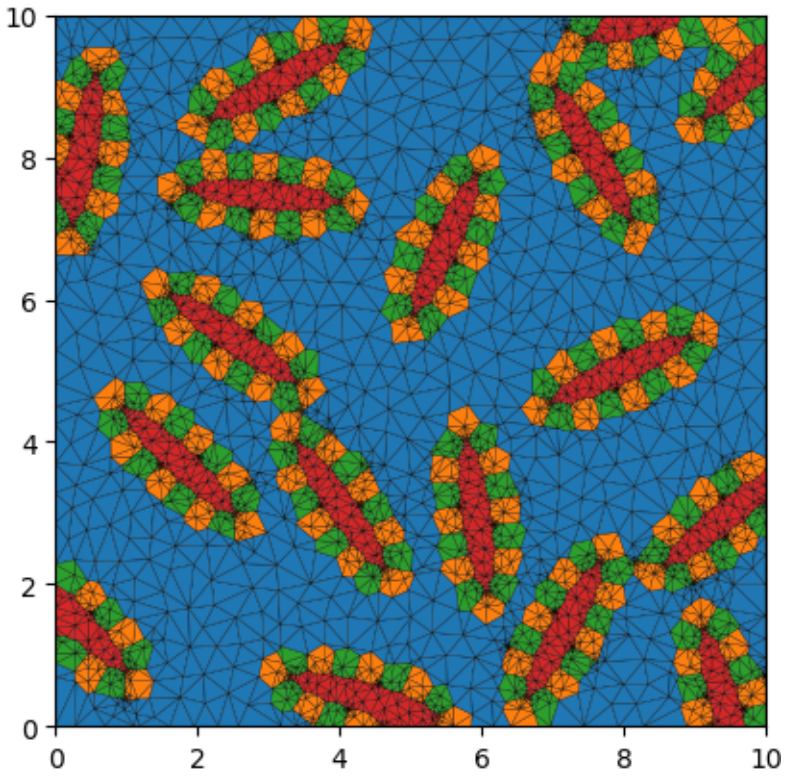

Fig. 3.39: Triangular mesh of microstructure with seed neighborhoods.

Python Script

The basename for this file is `from_image.py`. The file can be run using this command:

```
microstructpy --demo=from_image.py
```

The full text of the script is:

```
import os
import shutil

import numpy as np
from matplotlib import image as mpim
from matplotlib import pyplot as plt

import microstructpy as msp

# Read in image
image_basename = 'aluminum_micro.png'
image_path = os.path.dirname(__file__)
image_filename = os.path.join(image_path, image_basename)
image = mpim.imread(image_filename)
im_brightness = image[:, :, 0]

# Bin the pixels
br_bins = [0.00, 0.50, 1.00]
```

(continues on next page)

(continued from previous page)

```
bin_nums = np.zeros_like(im_brightness, dtype='int')
for i in range(len(br_bins) - 1):
    lb = br_bins[i]
    ub = br_bins[i + 1]
    mask = np.logical_and(im_brightness >= lb, im_brightness <= ub)
    bin_nums[mask] = i

# Define the phases
phases = [{'color': c, 'material_type': 'amorphous'} for c in ('C0', 'C1')]

# Create the polygon mesh
m, n = bin_nums.shape
x = np.arange(n + 1).astype('float')
y = m + 1 - np.arange(m + 1).astype('float')
xx, yy = np.meshgrid(x, y)
pts = np.array([xx.flatten(), yy.flatten()]).T
kps = np.arange(len(pts)).reshape(xx.shape)

n_facets = 2 * (m + m * n + n)
n_regions = m * n
facets = np.full((n_facets, 2), -1)
regions = np.full((n_regions, 4), 0)
region_phases = np.full(n_regions, 0)

facet_top = np.full((m, n), -1, dtype='int')
facet_bottom = np.full((m, n), -1, dtype='int')
facet_left = np.full((m, n), -1, dtype='int')
facet_right = np.full((m, n), -1, dtype='int')

k_facets = 0
k_regions = 0
for i in range(m):
    for j in range(n):
        kp_top_left = kps[i, j]
        kp_bottom_left = kps[i + 1, j]
        kp_top_right = kps[i, j + 1]
        kp_bottom_right = kps[i + 1, j + 1]

        # left facet
        if facet_left[i, j] < 0:
            fnum_left = k_facets
            facets[fnum_left] = (kp_top_left, kp_bottom_left)
            k_facets += 1

            if j > 0:
                facet_right[i, j - 1] = fnum_left
        else:
            fnum_left = facet_left[i, j]

        # right facet
        if facet_right[i, j] < 0:
```

(continues on next page)

(continued from previous page)

```
            fnum_right = k_facets
            facets[fnum_right] = (kp_top_right, kp_bottom_right)
            k_facets += 1

            if j + 1 < n:
                facet_left[i, j + 1] = fnum_right
        else:
            fnum_right = facet_right[i, j]

        # top facet
        if facet_top[i, j] < 0:
            fnum_top = k_facets
            facets[fnum_top] = (kp_top_left, kp_top_right)
            k_facets += 1

            if i > 0:
                facet_bottom[i - 1, j] = fnum_top
        else:
            fnum_top = facet_top[i, j]

        # bottom facet
        if facet_bottom[i, j] < 0:
            fnum_bottom = k_facets
            facets[fnum_bottom] = (kp_bottom_left, kp_bottom_right)
            k_facets += 1

            if i + 1 < m:
                facet_top[i + 1, j] = fnum_bottom
        else:
            fnum_bottom = facet_bottom[i, j]

        # region
        region = (fnum_top, fnum_left, fnum_bottom, fnum_right)
        regions[k_regions] = region
        region_phases[k_regions] = bin_nums[i, j]
        k_regions += 1


pmesh = msp.meshing.PolyMesh(pts, facets, regions,
                             seed_numbers=range(n_regions),
                             phase_numbers=region_phases)

# Create the triangle mesh
tmesh = msp.meshing.TriMesh.from_polymesh(pmesh, phases=phases, min_angle=20)

# Plot triangle mesh
fig = plt.figure()
ax = plt.Axes(fig, [0., 0., 1., 1.])
ax.set_axis_off()
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
fig.add_axes(ax)
```

(continues on next page)

(continued from previous page)

```
fcs = [phases[region_phases[r]]['color'] for r in tmesh.element_attributes]
tmesh.plot(facecolors=fcs, edgecolors='k', lw=0.2)


plt.axis('square')
plt.xlim(x.min(), x.max())
plt.ylim(y.min(), y.max())
plt.axis('off')

# Save plot and copy input file
plot_basename = 'from_image/trimesh.png'
file_dir = os.path.dirname(os.path.realpath(__file__))
filename = os.path.join(file_dir, plot_basename)
dirs = os.path.dirname(filename)
if not os.path.exists(dirs):
    os.makedirs(dirs)
plt.savefig(filename, bbox_inches='tight', pad_inches=0)

shutil.copy(image_filename, dirs)
```

Read Image

The first section of the script reads the image using matplotlib. The brightness of the image is taken as the red channel, since the RGB values are equal. That image is shown in Fig. 3.40.

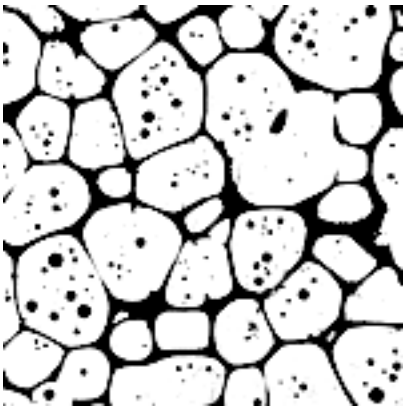

Fig. 3.40: Micrograph of aluminum.

Bin Pixels

The pixel values are binned based on whether the brightness is above or below 0.5.

Phases

The two phases are considered amorphous, to prevent pixilation in the triangle mesh.

Create the Polygon Mesh

The polygon mesh is a reproduction of the pixel grid in the image. The facets are edges between pixels, and the polygons are all squares.

Create the Triangle Mesh

The triangle mesh is created from the polygon mesh and uses the amorphous specifications for the phases. The minimum interior angle of the triangles is set to 20 degrees to control the aspect ratio of triangles.

Plot Triangle Mesh

The axes of the plot are switched off, then the triangle mesh is plotted. The color of each triangle is set by the phase.

Save Plot and Copy Input File

The final plot is saved to a file, then the input image is copied to the same directory for comparison. The output PNG file of this script is shown in Fig. 3.41.

3.3.7 Microstructure Mesh Process

Python Script

The basename for this file is `docs_banner.py`. The file can be run using this command:

```
microstructpy --demo=docs_banner.py
```

The full text of the file is:

```
from __future__ import division

import os

import numpy as np
import scipy.stats
from matplotlib import pyplot as plt

import microstructpy as msp


def main():
```

(continues on next page)

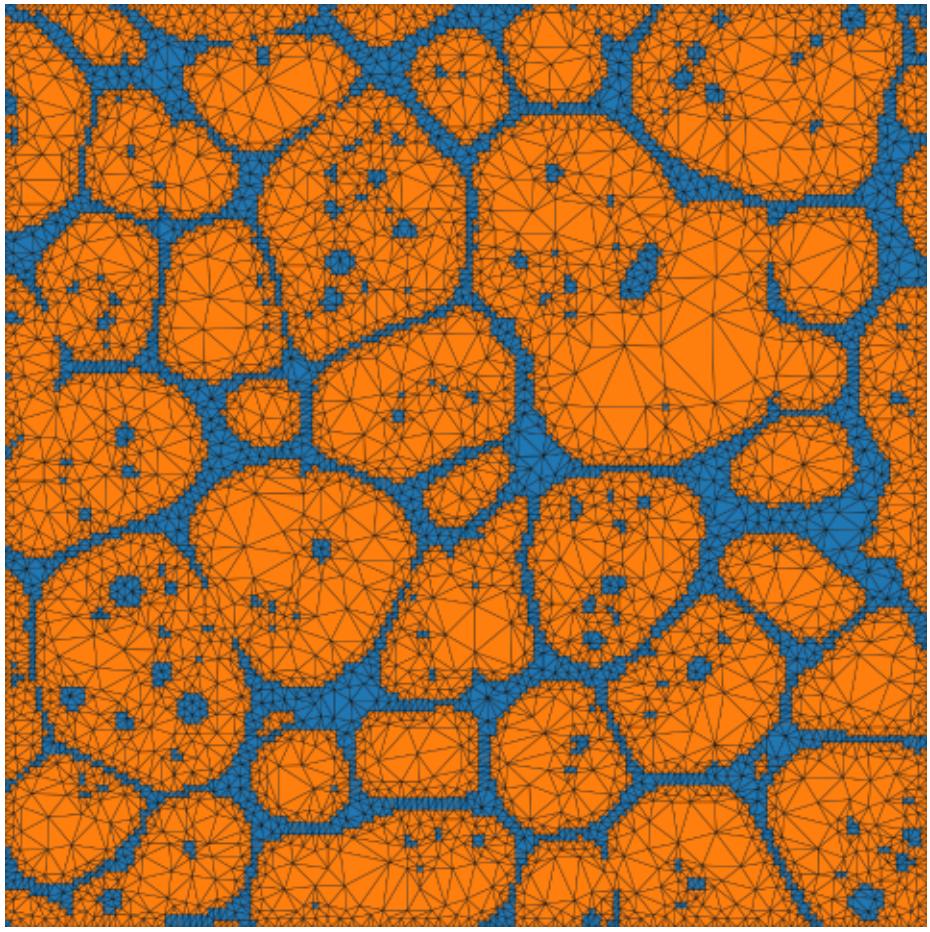

Fig. 3.41: Triangular mesh of aluminum microstructure.

(continued from previous page)

```
    # Colors
    c1 = '#12C2E9'
    c2 = '#C471ED'
    c3 = '#F64F59'

    # Offset
    off = 1

    # Create Directory
    dirname = os.path.join(os.path.dirname(__file__), 'docs_banner')
    if not os.path.exists(dirname):
        os.makedirs(dirname)

    # Create Domain
    domain = msp.geometry.Rectangle(width=10, length=20)

    # Create Unpositioned Seeds
    phase2 = {'color': c1}
    ell_geom = msp.geometry.Ellipse(a=8, b=3)
    ell_seed = msp.seeding.Seed(ell_geom, phase=2)

    mu = 1
    bnd = 0.5
    d_dist = scipy.stats.uniform(loc=mu-bnd, scale=2*bnd)
    phase0 = {'color': c2, 'shape': 'circle', 'd': d_dist}
    phase1 = {'color': c3, 'shape': 'circle', 'd': d_dist}
    circle_area = domain.area - ell_geom.area
    seeds = msp.seeding.SeedList.from_info([phase0, phase1], circle_area)

    seeds.append(ell_seed)
    hold = [False for seed in seeds]
    hold[-1] = True
    phases = [phase0, phase1, phase2]

    # Create Positioned Seeds
    seeds.position(domain, hold=hold, verbose=True)

    # Create Polygonal Mesh
    pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

    # Create Triangular Mesh
    tmesh = msp.meshing.TriMesh.from_polymesh(pmesh,
                                              min_angle=12,
                                              max_edge_length=0.2,
                                              max_volume=0.4)

    # Create Figure
    k = 0.12
    len_x = 3 * domain.length + 4 * off
    len_y = domain.width + 2 * off
    plt.figure(figsize=(k * len_x, k * len_y))
```

(continues on next page)

(continued from previous page)

```
    # Plot Seeds
    seed_colors = [phases[s.phase]['color'] for s in seeds]
    seeds.plot(color=seed_colors, alpha=0.8, edgecolor='k', linewidth=0.3)
    domain.plot(facecolor='none', edgecolor='k', linewidth=0.3)

    # Plot Polygonal Mesh
    pmesh.points = np.array(pmesh.points)
    pmesh.points[:, 0] += domain.length + off
    for region, phase_num in zip(pmesh.regions, pmesh.phase_numbers):
        if phase_num == 2:
            continue
        color = phases[phase_num]['color']

        facets = [pmesh.facets[f] for f in region]
        kps = ordered_kps(facets)
        x, y = zip(*[pmesh.points[kp] for kp in kps])
        plt.fill(x, y, color=color, alpha=0.8, edgecolor='none')

    ellipse_regions = set()
    for region_num, phase_num in enumerate(pmesh.phase_numbers):
        if phase_num == 2:
            ellipse_regions.add(region_num)

    ellipse_facets = []
    for facet, neighbors in zip(pmesh.facets, pmesh.facet_neighbors):
        common_regions = ellipse_regions & set(neighbors)
        if len(common_regions) == 1:
            ellipse_facets.append(facet)
    ellipse_kps = ordered_kps(ellipse_facets)
    x, y = zip(*[pmesh.points[kp] for kp in ellipse_kps])
    plt.fill(x, y, color=phases[-1]['color'], alpha=0.8, edgecolor='none')

    for facet, neighbors in zip(pmesh.facets, pmesh.facet_neighbors):
        common_regions = ellipse_regions & set(neighbors)
        if len(common_regions) < 2:
            x, y = zip(*[pmesh.points[kp] for kp in facet])
            plt.plot(x, y, color='k', linewidth=0.3)

    # Plot Triangular Mesh
    tmesh.points = np.array(tmesh.points)
    tmesh.points[:, 0] += 2 * off + 2 * domain.length
    tri_colors = [seed_colors[n] for n in tmesh.element_attributes]
    tmesh.plot(color=tri_colors, alpha=0.8, edgecolor='k', linewidth=0.2)

    # Set Up Axes
    plt.gca().set_position([0, 0, 1, 1])
    plt.axis('image')
    plt.gca().set_axis_off()
    plt.gca().get_xaxis().set_visible(False)
    plt.gca().get_yaxis().set_visible(False)

    xlim, ylim = domain.limits
```

(continues on next page)

(continued from previous page)

```
    xlim[0] -= off
    xlim[1] += 3 * off + 2 * domain.length

    ylim[0] -= off
    ylim[1] += off

    plt.axis(list(xlim) + list(ylim))

    fname = os.path.join(dirname, 'banner.png')
    plt.savefig(fname, bbox='tight', pad_inches=0)
    plt.savefig(fname.replace('.png', '.pdf'), bbox='tight', pad_inches=0)


def ordered_kps(pairs):
    t_pairs = [tuple(p) for p in pairs]
    kps = list(t_pairs.pop())
    while t_pairs:
        for i, pair in enumerate(t_pairs):
            if kps[-1] in pair:
                break
        assert kps[-1] in pair, pairs
        kps += [kp for kp in t_pairs.pop(i) if kp != kps[-1]]
    return kps[:-1]


if __name__ == '__main__':
    main()
```

Domain Geometry

The materials fill a rectangular domain with side lengths 20 and 10. The center of the rectangle defaults to the origin.

Seeds

The first material is phase 2, which contains a single elliptical seed with semi-axes 8 and 3. Next, phases 0 and 1 are created with identical size distributions and different colors. The size distributions are uniform random from 0.5 to 1.5. Seeds of phase 0 and phase 1 are generated to fill the area between the rectangular domain and the elliptical seed from phase 2.

Next, the phase 2 seed is appended to the list of phase 0 and 1 seeds. A hold list is then created to indicate to *position()* which seeds should have their positions (centers) held. The default position of a seed is the origin, so by setting the hold flag to `True` for the elliptical seed, it will be fixed to the center of the domain while the remaining seeds will be randomly positioned around it.

Polygonal and Triangular Meshing

Once the seeds are positioned in the domain, a polygonal mesh is created using *from_seeds()*. The triangular mesh is created using *from_polymesh()*, with the quality control settings `min_angle`, `max_edge_length`, and `max_volume`.

Plot Figure

The figure contains three plots: the seeds, the polygonal mesh, and the triangular/unstructured mesh. First, the seeds plot is generated using SeedList *plot()* and Rectangle *plot()* to show the boundary of the domain. The seeds are plotted with some transparency to show overlap.

Next, the polygonal mesh is translated to the right and plotted in such a way that avoids the internal geometry of the elliptical seed. This internal geometry is created by the multi-circle approximation used in polygonal meshing, then removed during the triangular meshing process. In the interest of clarity, these two steps are combined and the elliptical grain is plotted without internal geomtry.

Finally, the triangular mesh is translated to the right of the polygonal mesh and plotted using TriMesh *plot()*.

Once all three plots have been added to the figure, the axes and aspect ratio are adjusted. This figure is shown in Fig. 3.42. The PNG and PDF versions of this plot are saved in a folder named `docs_banner`, in the current directory (i.e `./docs_banner`).

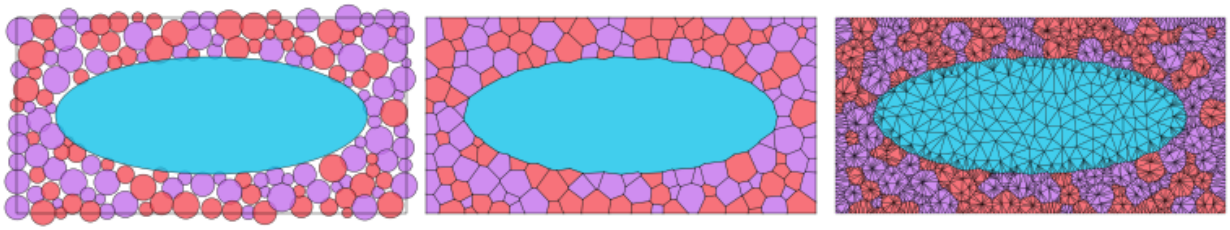

Fig. 3.42: Microstructure meshing process.

The three major steps are: 1) seed the domain with particles, 2) create a Voronoi power diagram, and 3) convert the diagram into an unstructured mesh.

CHAPTER

FOUR

COMMAND LINE GUIDE

4.1 Introduction

4.1.1 Using the Command Line Interface

The command line interface (CLI) for this package is `microstructpy`. This command accepts the names of user-generated files and demonstration files. Multiple filenames can be specified.

To run demos, you can specify a particular demo file or to run all of them:

```
microstructpy --demo=minimal.xml
microstructpy --demo=all
```

Demo files are copied to the current working directory and then executed. Running all of the demonstration files may take several minutes.

User-generated input files can be run in a number of ways:

```
microstructpy /path/to/my/input_file.xml
microstructpy input_1.xml input_2.xml input_3.xml
microstructpy input_*.xml
```

Both relative and absolute filepaths are acceptable.

The following pages describe in detail the various uses and options for the material, domain, and settings fields of a MicroStructPy input file.

4.1.2 Command Line Procedure

The following tasks are performed by the CLI:

1. Make the output directory, if necessary
2. **Create a list of unpositioned seeds**
3. **Position the seeds in the domain**
4. Save the seeds in a text file
5. Save a plot of the seeds to an image file
6. **Create a polygon mesh from the seeds**
7. Save the mesh to the output directory
8. Save a plot of the mesh to the output directory

9. **Create an unstructured (triangular or tetrahedral) mesh**
10. Save the unstructured mesh
11. Save a plot of the unstructured mesh
12. (optional) Verify the output mesh against the input file.

Intermediate results are saved in steps 4, 7, and 10 to give the option of restarting the procedure. The format of the output files can be specified in the input file (e.g. PNG and/or PDF plots).

4.1.3 Example Input File

Input files for MicroStructPy must be in XML format. The three fields of the input file that MicroStructPy looks for are: <material>, <domain>, and <settings> (optional). For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 0.09 </size>
    </material>

    <domain>
        <shape> square </shape>
    </domain>

    <settings>
        <directory> minimal </directory>
        <plot_axes> False </plot_axes>
        <color_by> seed number </color_by>
        <colormap> Paired </colormap>
        <mesher> gmsh </mesher>
        <mesh_size> 0.03 </mesh_size>
    </settings>
</input>
```

This will create a microstructure with approximately circular grains that fill a domain that is 11x larger and color them according to the colormap Paired.

Note: XML fields that are not recognized by MicroStructPy will be ignored by the software. For example, material properties or notes can be included in the input file without affecting program execution.

Note: The order of fields in the XML input file is not strictly important, since the file is converted into a Python dictionary. When fields are repeated, such as including multiple materials, the order is preserved.

4.1.4 Including References to Other Input Files

The input file can optionally *include* references to other input files. For example if the file `materials.xml` contains:

```
<input>
    <material>
        <shape> circle </shape>
        <size> 0.1 </size>
    </material>
</input>
```

and another file, `domain_1.xml`, contains:

```
<input>
    <include> materials.xml </include>
    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

then MicroStructPy will read the contents of `materials.xml` when `microstructpy domain_1.xml` is called. This functionality can allows multiple input files to reference the same material properties. For example, a mesh convergence study could keep the materials and domain definitions in a single file, then the input files for each mesh size would contain the run settings and a reference to the definitions file.

This way, if a parameter such as the grain size distribution needs to be updated, it only needs to be changed in a single file.

Advanced Usage

The <include> tag can be included at any heirarchical level of the input file. It can also be nested, with <include> tags in the file being included. For example, if the file `fine_grained.xml` contains:

```
<material>
    <shape> circle </shape>
    <size> 0.1 </size>
</material>
```

and the file `materials.xml` contains:

```
<input>
    <material>
        <name> Fine 1 </name>
        <include> fine_grained.xml </include>
    </material>

    <material>
        <name> Fine 2 </name>
        <include> fine_grained.xml </include>
    </material>

    <material>
        <name> Coarse </name>
```

(continues on next page)

(continued from previous page)

```
        <shape> circle </shape>
        <size> 0.3 </size>
    </material>
</input>
```

and the file `input.xml` contains:

```
<input>
    <include> materials.xml </include>
    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
    </domain>
</input>
```

then running `microstructpy input.xml` would be equivalent to running this file:

```
<input>
    <material>
        <name> Fine 1 </name>
        <shape> circle </shape>
        <size> 0.1 </size>
    </material>

    <material>
        <name> Fine 2 </name>
        <shape> circle </shape>
        <size> 0.1 </size>
    </material>

    <material>
        <name> Coarse </name>
        <shape> circle </shape>
        <size> 0.3 </size>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
    </domain>
</input>
```

The <include> tag can reduce file sizes and the amount of copy/paste for microstructures with multiple materials of the same size distribution, or multiple runs with the same material.

4.2 <material> - Material Phases

4.2.1 Single Material

MicroStructPy supports an arbitrary number of materials, including just one. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 0.15 </size>
    </material>

    <domain>
        <shape> square </shape>
    </domain>
</input>
```

This input file will produce a microstructure with nearly circular grains of size 0.15 (within a domain with side length 1).

4.2.2 Multiple Materials

MicroStructPy supports an arbitrary number of materials within a microstructure. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 1 </size>
        <fraction> 0.2 </fraction>
    </material>

    <material>
        <shape> circle </shape>
        <size> 0.5 </size>
        <fraction> 0.3 </fraction>
    </material>

    <material>
        <shape> circle </shape>
        <size> 1.5 </size>
        <fraction> 0.5 </fraction>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

Here there are three phases: the first has grain size 1 and makes up 20% of the area, the second has grain size 0.5 and makes up 30% of the area, and the third has grain size 1.5 and makes up 50% of the area. If the fractions are not

specified, MicroStructPy assumes the phases have equal volume fractions. The fractions can also be given as ratios (e.g. 2, 3, and 5) and MicroStructPy will normalize them to fractions.

Volume fractions can also be distributed quantities, rather than fixed values. This is useful if measured volume fractions have some uncertainty. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 1 </size>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 0.7 </loc>
            <scale> 0.03 </scale>
        </fraction>
    </material>

    <material>
        <shape> circle </shape>
        <size> 0.5 </size>
        <fraction>
            <dist_type> norm </dist_type>
            <loc> 0.3 </loc>
            <scale> 0.03 </scale>
        </fraction>
    </material>
</input>
```

Here the standard deviation on the volume fractions is 0.03, meaning that volume fractions are accurate to within 6 percentage points at 95% confidence.

Note: If the volume fraction distribution has negative numbers in the support, MicroStructPy will re-sample the distribution until a non-negative volume fraction is sampled.

4.2.3 Grain Size Distributions

Distributed grain sizes, rather than constant sizes, can be specified as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 1 </scale>
        </size>
    </material>

    <material>
        <shape> circle </shape>
```

(continues on next page)

(continued from previous page)

```
        <size>
            <dist_type> lognorm </dist_type>
            <scale> 0.5 </scale>
            <s> 0.1 </s>
        </size>
    </material>

    <material>
        <shape> circle </shape>
        <size>
            <dist_type> cdf </dist_type>
            <filename> my_empirical_dist.csv </filename>
        </size>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

In all three materials, the `size` field contains a `dist_type`. This type can match the name of a statistical distribution in the SciPy `scipy.stats` module, or be either "pdf" or "cdf". If it is a SciPy distribution name, then the remaining parameters must match the inputs for that function. The first material has size distribution $S \sim U(1, 2)$ and the second has distribution $S \sim 0.5e^{N(0,0.1)}$. Refer to the SciPy website for the complete list of available distributions and their input parameters.

In the case that the distribution type is "pdf" then the only other field should be `filename`. For a PDF, the file should contain two lines: the first has the (n) bin heights and the second has the (n+1) bin locations. A PDF file could contain, for example:

```
0.5, 1
1, 2, 2.5
```

For a CDF, the file should have two columns: the first being the size and the second being the CDF value. The equivalent CDF file would contain:

```
1, 0
2, 0.5
2.5, 1
```

Both PDF and CDF files should be in CSV format.

Warning: Do not use distributions that are equivalent to a deterministic value, such as $S \sim N(1, 0)$. The infinite PDF value causes numerical issues for SciPy. Instead, replace the distribution with the deterministic value or use a small, non-zero variance.

4.2.4 Grain Geometries

MicroStructPy supports several grain geometries and each can be specified in multiple ways. For example, the ellipse can be specified in terms of its area and aspect ratio, or by its semi-major and semi-minor axes. The 'size' of a grain is defined as the diameter of a circle or sphere with equivalent area (so for a general ellipse, this would be $2\sqrt{ab}$). The parameters available for each geometry are described in the lists below.

Circle

Class: *microstructpy.geometry.Circle*

Parameters

- `area` - the area of the circle
- `d` - alias for `diameter`
- `diameter` - the diameter of the circle
- `r` - alias for `radius`
- `radius` - the radius of the circle
- `size` - same as `diameter`

Only one of the above is necessary to define the circle geometry. If no parameters are specified, the default is a unit circle.

Ellipse

Class: *microstructpy.geometry.Ellipse*

Parameters

- `a` - the semi-major axis of the ellipse
- `angle` - alias for `angle_deg`
- `angle_deg` - the counterclockwise positive angle between the semi-major axis and the +x axis, measured in degrees
- `angle_rad` - the counterclockwise positive angle between the semi-major axis and the +x axis, measured in radians
- `aspect_ratio` - the ratio a/b
- `axes` - semi-axes of ellipse, equivalent to [a, b]
- `b` - the semi-minor axis of the ellipse
- `matrix` - orientation matrix for the ellipse
- `orientation` - alias for `matrix`
- `size` - the diameter of a circle with the same area as the ellipse

Two shape parameters and one orientation parameter are necessary to fully define the ellipse geometry. If less than two shape parameters are given, the default is a unit circle. If an orientation parameter is not given, the default is aligned with the coordinate axes.

Note: The default orientation of an ellipse is aligned with the coordinate axes. Uniform random orientation can be achieved by setting `<orientation> random </orientation>` in the input file.

Ellipsoid

Class: *microstructpy.geometry.Ellipsoid*

Parameters

- `a` - first semi-axis of the ellipsoid
- `axes` - semi-axes of the ellipsoids, equivalent to [a, b, c]
- `b` - second semi-axis of the ellipsoid
- `c` - third semi-axis of the ellipsoid
- `matrix` - orientation matrix for the ellipsoid
- `orientation` - alias for `matrix`
- `ratio_ab` - the ratio a/b
- `ratio_ac` - the ratio a/c
- `ratio_ba` - the ratio b/a
- `ratio_bc` - the ratio b/c
- `ratio_ca` - the ratio c/a
- `ratio_cb` - the ratio c/b
- `rot_seq` - alias for `rot_set_seq`
- `rot_seq_deg` - a rotation sequence, with angles in degrees, to define the orientation of the ellipsoid. See below for details.
- `rot_seq_rad` - a rotation sequence, with angles in radians, to define the orientation of the ellipsoid. See below for details.
- `size` - the diameter of a sphere with the same volume as the ellipsoid

Three shape parameters and one orientation parameter are necessary to fully define the ellipsoid geometry. If the length of a semi-axis cannot be determined from the input parameters, it defaults to unit length. If an orientation parameter is not given, the default is aligned with the coordinate axes.

A rotation sequence is a list of axes and angles to rotate the ellipsoid. The order of the rotations is as such: supposing after a z-rotation that the new x and y axes are x' and y', a subsequent y-rotation would be about the y' axis. An example rotation sequence is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> ellipsoid </shape>
        <axes> 5, 3, 1 </axes>

        <rot_seq_deg>
            <axis> z </axis>
            <angle> 10 </angle>
```

(continues on next page)

(continued from previous page)

```
        </rot_seq_deg>

        <rot_seq_deg>
            <axis> x </axis>
            <angle>
                <dist_type> uniform </dist_type>
                <loc> 30 </loc>
                <scale> 30 </scale>
            </angle>
        </rot_seq_deg>

        <rot_seq_deg>
            <axis> y </axis>
            <angle>
                <dist_type> norm </dist_type>
                <loc> 0 </loc>
                <scale> 30 </scale>
            </angle>
        </rot_seq_deg>
    </material>
</input>
```

This represents first a z-rotation of 10 degrees, then an x-rotation of 30-60 degrees, then finally a y-rotation of $N(0, 30)$ degrees.

Note: Ellipsoids with uniform random distribution will be generated using `<orientation> random </orientation>`. Positions on the unit 4-sphere are generated with a uniform random distribution, then converted into a quaternion and finally into a rotation matrix.

Rectangle

Class: *microstructpy.geometry.Ellipsoid*

Parameters

- `angle`- alias for `angle_deg`
- `angle_deg` - rotation angle, in degrees, measured counterclockwise from the +x axis
- `angle_rad` - rotation angle, in radians, measured counterclockwise from the +x axis
- `length` - the x-direction side length of the rectangle
- `matrix` - the orientation matrix of the rectangle
- `side_lengths` - equivalent to [length, width]
- `width` - the y-direction side length of the rectangle

Both the length and the width of the rectangle must be specified. If either is not specified, the default rectangle is a sqaure with unit side length. If an orientation is not specified, the default is aligned with the coordinate axes.

Sphere

Class: *microstructpy.geometry.Sphere*

Parameters

- `d` - alias for `diameter`
- `diameter` - the diameter of the sphere
- `r` - alias for `radius`
- `radius` - the radius of the sphere
- `size` - alias for `diameter`
- `volume` - volume of the sphere

Only one of the above is necessary to define the sphere geometry. If no parameters are specified, the default is a unit sphere.

Square

Class: *microstructpy.geometry.Square*

Parameters

- `angle` - alias for `angle_deg`
- `angle_deg` - the rotation angle, in degrees, of the square measured counterclockwise from the +x axis
- `angle_rad` - the rotation angle, in radians, of the square measured counterclockwise from the +x axis
- `matrix` - the orientation matrix of the square
- `side_length` - the side lnegth of the square

If the side length of the square is not specified, the default is 1. If an orientation parameter is not specified, the default orientation is aligned with the coordinate axes.

Note: Over-parameterizing grain geometries will cause unexpected behavior.

For parameters such as "side_lengths" and "axes", the input is expected to be a list, e.g. `<axes> 1, 2 </axes>` or `<axes> (1, 2) </axes>`. For matrices, such as "orientation", the input is expected to be a list of lists, e.g. `<orientation> [[0, -1], [1, 0]] </orientation>`.

Each of the scalar arguments can be either a constant value or a distribution. For uniform random distribution of ellipse and ellipsoid axes, used the parameter `<orientation> random </orientation>`. The default orientation is axes-aligned.

Here is an example input file with non-circular grains:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> ellipse </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 1 </scale>
```

(continues on next page)

(continued from previous page)

```
        </size>
        <aspect_ratio> 3 </aspect_ratio>
        <orientation> random </orientation>
    </material>

    <material>
        <shape> square </shape>
        <side_length>
            <dist_type> lognorm </dist_type>
            <scale> 0.5 </scale>
            <s> 0.1 </s>
        </side_length>
    </material>

    <material>
        <shape> rectangle </shape>
        <length>
            <dist_type> cdf </dist_type>
            <filename> my_empirical_dist.csv </filename>
        </length>
        <width> 0.2 </width>
        <angle_deg>
            <dist_type> uniform <dist_type>
            <loc> -30 </loc>
            <scale> 60 </scale>
        </angle_deg>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

4.2.5 Material Type

There are three types of materials supported by MicroStructPy: crystalline, amorphous, and void. For crystalline phases, facets between cells of the same grain are removed before unstructured meshing. For amorphous phases, facets between cells of the same phase are removed before meshing. Finall, void phases produce empty spaces in the unstructured mesh. There are several synonyms for these material types, including:

- **crystalline**: granular, solid
- **amorphous**: glass, matrix
- **void**: crack, hole

The default material type is crystalline. An example input file with material types is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
```

(continues on next page)

(continued from previous page)

```
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1 </scale>
        </size>
        <material_type> matrix </material_type>
    </material>

    <material>
        <shape> square </shape>
        <side_length> 0.5 </side_length>
        <material_type> void </material_type>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

Here, the first phase is an amorphous (matrix) phase and the second phase contains square voids of constant size. A material can contain multiple amorphous and void phases.

Note: Void phases may cause parts of the mesh to become disconnected. MicroStructPy does not check for or remove disconnected regions from the mesh.

4.2.6 Grain Position Distribution

The default position distribution for grains is random uniform throughout the domain. Grains can be non-uniformly distributed by adding a position distribution. The x, y, and z can be independently distributed or coupled. The coupled distributions can be any of the multivariate distributions listed in the SciPy `scipy.stats` module.

In the example below, the first material has independently distributed coordinates while the second has a coupled distribution.

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1 </scale>
        </size>
        <position>  <!-- x -->
            <dist_type> binom </dist_type>
            <loc> 0.5 </loc>
            <n> 9 </n>
            <p> 0.5 </p>
        </position>
```

(continues on next page)

(continued from previous page)

```
        <position> <!-- y -->
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 10 </scale>
        </position>
    </material>

    <material>
        <shape> square </shape>
        <side_length> 0.5 </side_length>
        <position>
            <dist_type> multivariate_normal </dist_type>
            <mean> [2, 3] </mean>
            <cov> [[4, -1], [-1, 3]] </cov>
        </position>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
        <corner> 0, 0 </corner>
    </domain>
</input>
```

Position distributions should be used with care, as seeds may not fill the entire domain.

4.2.7 Other Material Settings

Name The name of each material can be specified by adding a "name" field. The default name is "Material N" where N is the order of the material in the XML file, starting from 0.

Color The color of each material in output plots can be specified by adding a "color" field. The default color is "CN" where N is the order of the material in the XML file, starting from 0. For more information about color specification, visit the Matplotlib Specifying Colors webpage.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Aluminum </name>
        <color> silver </color>
        <shape> circle </shape>
        <size> 1 </size>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
    </domain>
</input>
```

4.3 <domain> - Microstructure Domain

MicroStructPy supports the following domain geometries:

- **2D**: circle, ellipse, rectangle, square
- **3D**: box, cube

Each geometry can be defined several ways, such as a center and edge lengths for the rectangle or two bounding points. Note that over-parameterizing the domain geometry will cause unexpected behavior.

4.3.1 Box

Class: *microstructpy.geometry.Box*

Parameters

- `bounds` - alias for `limits`
- `center` - the center of the box
- `corner` - the bottom-most corner of the box (i.e. $(x, y, z)_{min}$)
- `limits` - the x, y, z upper and lower bounds of the box (i.e. $[[x_{min}, x_{max}], [y_{min}, y_{max}], [z_{min}, z_{max}]]$)
- `side_lengths` - the x, y, and z side lengths of the box

Below are some example box domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example box domains -->
<input>
    <domain>
        <shape> box </shape>
        <!-- default side length is 1 -->
        <!-- default center is the origin -->
    </domain>

    <domain>
        <shape> box </shape>
        <side_lengths> 2, 1, 6 </side_lengths>
        <corner> 0, 0, 0 </corner>
    </domain>

    <domain>
        <shape> BOX </shape>
        <limits> 0, 2 </limits>   <!-- x -->
        <limits> -2, 1 </limits>  <!-- y -->
        <limits> -3, 0 </limits>  <!-- z -->
    </domain>

    <domain>
        <shape> boX </shape>  <!-- case insensitive -->
        <bounds> [[0, 2], [-2, 1], [-3, 0]] </bounds>
    </domain>
</input>
```

4.3.2 Circle

Class: *microstructpy.geometry.Circle*

Parameters

- `area` - the area of the circle
- `center` - the center of the circle
- `d` - alias for `diameter`
- `diameter` - the diameter of the circle
- `r` - alias for `radius`
- `radius` - the radius of the circle
- `size` - same as `diameter`

The default radius of a circle is 1, while the default center is (0, 0).

Below are some example circle domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example circle domains -->
<input>
    <domain>
        <shape> circle </shape>
        <!-- default radius is 1 -->
        <!-- default center is the origin -->
    </domain>

    <domain>
        <shape> circle </shape>
        <diameter> 3 </diameter>
    </domain>

    <domain>
        <shape> circle </shape>
        <radius> 10 </radius>
        <center> 0, 10 <center>
    </domain>
</input>
```

4.3.3 Cube

Class: *microstructpy.geometry.Cube*

Parameters

- `center` - the center of the cube
- `corner` - the bottom-most corner of the cube (i.e. $(x, y, z)_{min}$)
- `side_length` - the side length of the cube

The defaultt side length of the cube is 1, while the default center is (0, 0).

Below are some example cube domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example cube domains -->
<input>
    <domain>
        <shape> cube </shape>
        <!-- default side length is 1 -->
        <!-- default center is the origin -->
    </domain>

    <domain>
        <shape> cube </shape>
        <side_length> 10 </side_length>
        <corner> (0, 0, 0) </corner>
    </domain>

    <domain>
        <shape> cube </shape>
        <corner> 0, 0, 0 </corner>
    </domain>
</input>
```

4.3.4 Ellipse

Class: *microstructpy.geometry.Ellipse*

Parameters

- `a` - the semi-major axis of the ellipse
- `angle` - alias for `angle_deg`
- `angle_deg` - the counterclockwise positive angle between the semi-major axis and the +x axis, measured in degrees
- `angle_rad` - the counterclockwise positive angle between the semi-major axis and the +x axis, measured in radians
- `aspect_ratio` - the ratio a/b
- `axes` - semi-axes of ellipse, equivalent to [a, b]
- `b` - the semi-minor axis of the ellipse
- `center` - the center of the ellipse
- `matrix` - orientation matrix for the ellipse
- `orientation` - alias for `matrix`
- `size` - the diameter of a circle with the same area as the ellipse

The default value for the semi-axes of the ellipse is 1. The default orientation of the ellipse is aligned with the coordinate axes. Finally, the default position of the ellipse is centered at (0, 0).

Below are some example ellipse domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example ellipse domains -->
```

(continues on next page)

(continued from previous page)

```
<input>
    <domain>
        <shape> ellipse </shape>
        <!-- default is a unit circle centered at the origin -->
    </domain>

    <domain>
        <shape> ellipse </shape>
        <a> 10 </a>
        <b>  4 </b>
        <angle> 30 </angle>
        <center> 2, -1 </center>
    </domain>

    <domain>
        <shape> ellipse </shape>
        <axes> 5, 3 </axes>
    </domain>

    <domain>
        <shape> ellipse </shape>
        <size> 10 </size>
        <aspect_ratio> 5 </aspect_ratio>
        <angle_deg> -45 </angle_deg>
    </domain>
</input>
```

4.3.5 Rectangle

Class: *microstructpy.geometry.Rectangle*

Parameters

- `bounds` - alias for `limits`
- `center` - the center of the rectangle
- `corner` - the bottom-most corner of the rectangle (i.e. $(x, y)_{min}$)
- `length` - the x-direction side length of the rectangle
- `limits` - the x and y upper and lower bounds of the rectangle (i.e. $[[x_{min}, x_{max}], [y_{min}, y_{max}]]$)
- `side_lengths` - equivalent to [length, width]
- `width` - the y-direction side length of the rectangle

The default side lengths of the rectangle are 1, while the default position is centered at the origin.

Below are some example rectangle domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example rectangle domains -->
<input>
    <domain>
        <shape> rectangle </shape>
```

(continues on next page)

(continued from previous page)

```
        <!-- default side length is 1 -->
        <!-- default center is the origin -->
    </domain>

    <domain>
        <shape> rectangle </shape>
        <side_lengths> 2, 1 </side_lengths>
        <corner> 0, 0 </corner>
    </domain>

    <domain>
        <shape> rectangle </shape>
        <limits> 0, 2 </limits>   <!-- x -->
        <limits> -2, 1 </limits>  <!-- y -->
    </domain>

    <domain>
        <shape> rectangle </shape>
        <bounds> [[0, 2], [-2, 1]] </bounds>
    </domain>
</input>
```

4.3.6 Square

Class: *microstructpy.geometry.Square*

Parameters

- `side_length` - the side length of the square
- `center` - the position of the center of the square
- `corner` - the bottom-most corner of the square (i.e. $(x, y)_{min}$)

The default side length of a square is 1, while the default center position is (0, 0).

Below are some example square domain definitions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example square domains -->
<input>
    <domain>
        <shape> square </shape>
        <!-- default side length is 1 -->
        <!-- default center is the origin -->
    </domain>

    <domain>
        <shape> square </shape>
        <side_length> 2 </side_length>
        <corner> 0, 0 </corner>
    </domain>

    <domain>
```

(continues on next page)

(continued from previous page)

```
        <shape> square </shape>
        <corner> 0, 0 </corner>
    </domain>

    <domain>
        <shape> square </shape>
        <side_length> 10 </side_length>
        <center> 5, 0 </center>
    </domain>
</input>
```

4.4 <settings> - Settings

4.4.1 Defaults

Settings can be added to the input file to specify file outputs and mesh quality, among other things. The default settings are:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Default settings -->
<input>
    <settings>
        <!-- File and Console I/O -->
        <verbose> False </verbose>
        <directory> . </directory>

        <filetypes>
            <seeds> txt </seeds>
            <poly> txt </poly>
            <tri> txt </tri>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
            <verify_plot> png </verify_plot>
        </filetypes>

        <!-- Run Settings -->
        <restart> True </restart>

        <rng_seeds>
            <position> 0 </position>
            <!-- RNG can be set for grain shape distributions as well. -->
            <!-- For example, <size> 2 </size> seeds the RNG for       -->
            <!-- sampling size distributions with 2.                   -->
        </rng_seeds>

        <rtol> fit </rtol>

        <edge_opt> False </edge_opt>
        <edge_opt_n_iter> 100 </edge_opt_n_iter>
```

(continues on next page)

(continued from previous page)

```
        <mesher> Triangle/TetGen </mesher>

        <mesh_size> inf </mesh_size> <!-- used with gmsh -->
        <mesh_max_volume> inf </mesh_max_volume> <!-- used with Triangle and TetGen -->
        <mesh_min_angle> 0 </mesh_min_angle> <!-- used with Triangle and TetGen -->
        <mesh_max_edge_length> inf </mesh_max_edge_length> <!-- used with Triangle -->

        <verify> False </verify>

        <!-- Plot Controls -->
        <plot_axes> True </plot_axes>

        <color_by> material </color_by>
        <colormap> viridis </colormap>

        <seeds_kwargs> </seeds_kwargs>
        <poly_kwargs> </poly_kwargs>
        <tri_kwargs> </tri_kwargs>
    </settings>
</input>
```

4.4.2 File and Console I/O

verbose

The verbose flag toggles text updates to the console as MicroStructPy runs. Setting `<verbose> True </verbose>` will print updates, while False turns them off.

directory

The directory field is for the path to the output files. It can be an absolute file path, or relative to the input file. For example, if the file is in `aa/bb/cc/input.xml` and the directory field is `<directory> ../output </directory>`, then MicroStructPy will write output files to `aa/bb/output/`. If the output directory does not exist, MicroStructPy will create it.

filetypes

This field is for specifying output filetypes. The possible subfields are seeds, seeds_plot, poly, poly_plot, tri, tri_plot, and verify_plot. Below is an outline of the possible filetypes for each subfield.

- seeds

  txt, **vtk**

  Currently the only options are to output the seed geometries as a cache txt file or as a VTK legacy file. The VTK file can be opened in ParaView, with the seeds shown as glyphs.

- seeds_plot

  ps, **eps**, **pdf**, **pgf**, **png**, **raw**, **rgba**, **svg**, **svgz**, **jpg**, **jpeg**, **tif**, **tiff**

  These are the standard matplotlib output filetypes.

- poly

  txt, **poly** (2D only), **ply**, **vtk**

  A poly file contains a planar straight line graph (PSLG) and cane be read by Triangle. More details on poly files can be found on the .poly files page of the Triangle website. The ply file contains the surfaces between grains and the boundary of the domain. VTK legacy files also contain the polygonal grains in 2D and polyhedral grains in 3D.

- poly_plot

  ps, **eps**, **pdf**, **pgf**, **png**, **raw**, **rgba**, **svg**, **svgz**, **jpg**, **jpeg**, **tif**, **tiff**

  These are the standard matplotlib output filetypes.

- tri

  txt, **abaqus**, **tet/tri**, **vtk** (3D only)

  The abaqus option will create a part for each grain and assembly the parts. The tet/tri option will create .node and .elem files in the same format as the output of Triangle or TetGen. VTK files are suitable for viewing the mesh interactively in a program such as Paraview.

- tri_plot

  ps, **eps**, **pdf**, **pgf**, **png**, **raw**, **rgba**, **svg**, **svgz**, **jpg**, **jpeg**, **tif**, **tiff**

  These are the standard matplotlib output filetypes.

- verify_plot

  ps, **eps**, **pdf**, **pgf**, **png**, **raw**, **rgba**, **svg**, **svgz**, **jpg**, **jpeg**, **tif**, **tiff**

  These are the standard matplotlib output filetypes.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <settings>
        <filetypes>
            <seeds> txt </seeds>
            <seeds_plot> png, pdf </seeds_plot>
            <poly> txt, ply </poly>
            <poly_plot> svg </poly_plot>
            <tri> txt </tri>
            <tri_plot> pdf </tri_plot>
            <verify_plot> pdf </verify_plot>
        </filetypes>
    </settings>
</input>
```

If a subfield is not specified, that output is not saved to any file. The exception is, if `<restart> True </restart>`, then the seeds, poly mesh, and tri mesh will all be output to txt files.

4.4.3 Run Settings

restart

The restart flag will read the intermediate txt output files, if they exist, instead of duplicating previous work. Setting `<restart> True </restart>` will read the txt files, while False will ignore the existing txt files.

rng_seeds

The random number generator (RNG) seeds can be included to create multiple, repeatable realizations of a microstructure. By default, RNG seeds are all set to 0. An RNG seed can be specified for any of the distributed parameters in grain geometry. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <radius>
            <dist_type> uniform </dist_type>
            <loc> 1 </loc>
            <scale> 2 </scale>
        </radius>
    </material>

    <material>
        <shape> ellipse </shape>
        <axes> 1, 2 </axes>
        <angle_deg>
            <dist_type> norm </dist_type>
            <loc> 0 <loc>
            <scale> 15 </scale>
        </angle_deg>
    </material>

    <settings>
        <rng_seeds>
            <radius> 1 </radius>
            <angle_deg> 0 </angle_deg>
            <position> 3 </position>
        </rng_seeds>
    </settings>
</input>
```

In this case, if the position RNG were changed from 3 to 4 and the rest of the RNG seeds remained the same, MicroStructPy would generate the same set of seed geometries and arrange them differently in the domain.

rtol

The rtol field is for the relative overlap tolerance between seed geometries. The overlap is relative to the radius of the smaller circle or sphere. Overlap is acceptable if

$$\frac{r_1 + r_2 - ||x_1 - x_2||}{min(r_1, r_2)} < rtol$$

The default value is `<rtol> fit </rtol>`, which uses a fit curve to determine an appropriate value of rtol. This curve considers the coefficient of variation in grain volume and estimates an rtol value that maximizes the fit between input and output distributions.

Acceptable values of rtol are 0 to 1 inclusive, though rtol below 0.2 will likely result in long runtimes.

edge_opt

The edge_opt field provides the option to maximize the shortest edge in the polygonal/polyhedral mesh. The default is `<edge_opt> False </edge_opt>`, which skips the optimization process. This optimization is performed by making small adjustments to the positions of seeds surrounding the shortest edge, assessing if the change created an improvement, then either a) attempting a different change for the same edge if there was not improvement or b) moving on to the new shortest edge. The optimization algorithm exits when `edge_opt_n_iter` iterations have been performed on the same edge.

This flag is useful if the polygonal/polyhedral or triangular/tetrahedral are used in numerical simulations, such as finite element analysis. A high ratio of longest edge to shortest edge leads to a high ratio in maximum to minimum eigenvalue in FEA stiffness matrices, which can create problems for the FEA solver. Setting `edge_opt` to `True` will reduce short edges in the polygonal mesh, which translates into reduced short edges in the triangular mesh. This optimization process, however, will increase the time to generate a polygonal mesh. To track the progress of the optimizer, set `verbose` to `True`.

edge_opt_n_iter

This field specifies how many times the optimizer should attempt to increase the length of the shortest edge in the polygonal mesh. The default is `<edge_opt_n_iter> 100 </edge_opt_n_iter>`, which limits the optimizer to 100 attempts per edge. This field is ignored if `edge_opt` is set to `False`.

mesher

This field specifies how to mesh the PolyMesh. If set to `raster`, the output mesh will contain pixels/voxels of the PolyMesh. Other options include `Triangle/Tetgen` and `gmsh`.

mesh_size

This field specifies a target element size if using the `gmsh` mesher.

mesh_max_volume

This field defines the maximum volume (or area, in 2D) of any element in the triangular (unstructured) mesh. The default is <mesh_max_volume> inf </mesh_max_volume>, which turns off the volume control. In this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <area> 0.01 </area>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 1 </side_length>
    </domain>

    <settings>
        <mesh_max_volume> 0.001 </mesh_max_volume>
    </settings>
</input>
```

the unstructured mesh will have at least 10 elements per grain and at least 1000 elements overall.

mesh_min_angle

This field defines the minimum interior angle, measured in degrees, of any element in the triangular mesh. For 3D meshes, this is the minimum *dihedral* angle, which is between faces of the tetrahedron. This setting controls the aspect ratio of the elements, with angles between 15 and 30 degrees producing good quality meshes. The default is <mesh_min_angle> 0 </mesh_min_angle>, which effectively turns off the angle quality control.

mesh_max_edge_length

This field defines the maximum edge length along a grain boundary in a 2D triangular mesh. A small maximum edge length will increase resolution of the mesh at grain boundaries. Currently this feature has no equivalent in 3D. The default value is <mesh_max_edge_length> inf </mesh_max_edge_length>, which effectively turns off the edge length quality control.

verify

The verify flag will perform mesh verification on the triangular mesh and report error metrics. To include mesh verification, include <verify> True </verify> in the settings. The default behavior is to not perform mesh verification.

4.4.4 Plot Controls

plot_axes

The plot_axes flag toggles the axes on or off in the output plots. Setting it to False turns the axes off, producing images with miniminal borders. The default setting is `<plot_axes> True </plot_axes>`, which includes the coordinate axes in output plots.

color_by

The color_by field defines how the seeds and grains should be colored in the output plots. There are three options for this field: "material", "seed number", and "material number". The default setting is `<color_by> material </color_by>`. Using "material", the output plots will color each seed/grain with the color of its material. Using "seed number", the seeds/grains are colored by their seed number, which is converted into a color using the `colormap`. The "material number" option behaves in the same was as "seed number", except that the material numbers are used instead of seed numbers.

colormap

The colormap field is used when `color_by` is set to either "seed number" or "material number". This gives the name of the colormap to be used in coloring the seeds/grains. For a complete list of available colormaps, visit the Choosing Colormaps in Matplotlib webpage.

seeds_kwargs

This field contains optional keyword arguments passed to matplotlib when plotting the seeds. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <settings>
        <seeds_kwargs>
            <edgecolor> none </edgecolor>
            <alpha> 0.5 </alpha>
        </seeds_kwargs>
    </settings>
</input>
```

will plot the seeds with some transparency and no borders.

poly_kwargs

This field contains optional keyword arguments passed to matplotlib when plotting the polygonal mesh. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <settings>
        <poly_kwargs>
            <linewidth> 0.5 </linewidth>
            <edgecolors> blue </edgecolors>
        </poly_kwargs>
```

(continues on next page)

(continued from previous page)

```
    </settings>
</input>
```

will plot the mesh with thin, blue lines between the grains.

tri_kwargs

This field contains optional keyword arguments passed to matplotlib when plotting the triangular mesh. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <settings>
        <tri_kwargs>
            <linewidth> 0.5 </linewidth>
            <edgecolors> white </edgecolors>
        </tri_kwargs>
    </settings>
</input>
```

will plot the mesh with thin, white lines between the elements.

CHAPTER

FIVE

PYTHON PACKAGE GUIDE

The Python package for MicroStructPy includes the following:

```
microstructpy
├─ cli
├─ geometry
│   ├─ Box
│   ├─ Cube
│   ├─ Circle
│   ├─ Ellipse
│   ├─ Ellipsoid
│   ├─ Rectangle
│   ├─ Square
│   └─ Sphere
├─ seeding
│   ├─ Seed
│   └─ SeedList
├─ meshing
│   ├─ PolyMesh
│   └─ TriMesh
└─ verification
```

The cli module contains the functions related to the command line interface (CLI), including converting XML input files into dictionaries. The geometry module contains classes for seed and domain geometries. In the seeding package, there is the single Seed class and the SeedList class, which functions like a Python list but includes some additional methods such as positioning and plotting the seeds. Next, the PolyMesh and TriMesh classes are contained in the meshing module. A PolyMesh can be created from a SeedList and a TriMesh can be created from a PolyMesh. Finally, the verification module contains functions to compare the output PolyMesh and TriMesh with desired microstructural properties.

This guide explains how to use the MicroStructPy Python package. It starts with a script that executes an abbreviated version of the standard workflow. The checks, restarts, etc are excluded to show how the principal classes are used in a workflow. The following sections describe the meshing methods, the file I/O and plotting functions, and the format of a material phase dictionary.

5.1 The Standard Workflow

Below is an input file similar to the *Input File Introduction*. The script that follows will produce the same results as running this script from the command line interface.

XML Input File

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <name> Matrix </name>
        <material_type> matrix </material_type>
        <fraction> 2 </fraction>
        <shape> circle </shape>
        <size>
            <dist_type> uniform </dist_type>
            <loc> 0 </loc>
            <scale> 1.5 </scale>
        </size>
    </material>

    <material>
        <name> Inclusions </name>
        <fraction> 1 </fraction>
        <shape> circle </shape>
        <diameter> 2 </diameter>
    </material>

    <domain>
        <shape> square </shape>
        <side_length> 20 </side_length>
        <corner> (0, 0) </corner>
    </domain>

    <settings>
        <rng_seeds>
            <size> 1 </size>
        </rng_seeds>

        <mesh_min_angle> 25 </mesh_min_angle>
    </settings>
</input>
```

Equivalent Python Script

```
import matplotlib.pyplot as plt
import microstructpy as msp
import scipy.stats

# Create Materials
material_1 = {
    'name': 'Matrix',
    'material_type': 'matrix',
```

(continues on next page)

(continued from previous page)

```
    'fraction': 2,
    'shape': 'circle',
    'size': scipy.stats.uniform(loc=0, scale=1.5)
}

material_2 = {
    'name': 'Inclusions',
    'fraction': 1,
    'shape': 'circle',
    'diameter': 2
}

materials = [material_1, material_2]

# Create Domain
domain = msp.geometry.Square(side_length=15, corner=(0, 0))

# Create List of Un-Positioned Seeds
seed_area = domain.area
rng_seeds = {'size': 1}
seeds = msp.seeding.SeedList.from_info(materials,
                                       seed_area,
                                       rng_seeds)

# Position Seeds in Domain
seeds.position(domain)

# Create Polygonal Mesh
pmesh = msp.meshing.PolyMesh.from_seeds(seeds, domain)

# Create Triangular Mesh
min_angle = 25
tmesh = msp.meshing.TriMesh.from_polymesh(pmesh,
                                          materials,
                                          min_angle)

# Save txt files
seeds.write('seeds.txt')
pmesh.write('polymesh.txt')
tmesh.write('trimesh.txt')

# Plot outputs
seed_colors = ['C' + str(s.phase) for s in seeds]
seeds.plot(facecolors=seed_colors, edgecolor='k')
plt.axis('image')
plt.savefig('seeds.png')
plt.clf()

poly_colors = [seed_colors[n] for n in pmesh.seed_numbers]
pmesh.plot(facecolors=poly_colors, edgecolor='k')
plt.axis('image')
plt.savefig('polymesh.png')
```

(continues on next page)

(continued from previous page)

```
plt.clf()

tri_colors = [seed_colors[n] for n in tmesh.element_attributes]
tmesh.plot(facecolors=tri_colors, edgecolor='k')
plt.axis('image')
plt.savefig('trimesh.png')
plt.clf()
```

Highlighted are the four principal methods used in generating a microstructure: *SeedList.from_info()*, *SeedList.position()*, *PolyMesh.from_seeds()*, *TriMesh.from_polymesh()*.

5.2 Meshing Methods

5.2.1 Laguerre-Voronoi Tessellation

Polygonal/polyhedral meshes are generated in MicroStructPy using a Laguerre-Voronoi tessellation, also known as a Power Diagram. It is conceptually similar to a Voronoi diagram, the difference being that seed points are weighted rather than unweighted. In the *PolyMesh.from_seeds()* method, the center of a seed is consider a Voronoi seed point and the radius is its weight.

Non-circular seeds are replaced by their breakdown, resulting in multiple Voronoi cells representing a single grain. To retrieve all of the cells that represent a single grain, mask the `seed_numbers` property of a *PolyMesh*.

The Laguerre-Voronoi diagram is created by Voro++, which is accessed using pyvoro.

5.2.2 Unstructured Meshing

The triangular/tetrahedral meshes are generated in MicroStructPy using the MeshPy and pygmsh packages. MeshPy links with Triangle to create 2D triangular meshes and with TetGen to create 3D tetrahedral meshes. Pygmsh links with gmsh to produce both 2D and 3D meshes.

A polygonal mesh, *PolyMesh*, can be converted into an unstructured mesh using the *TriMesh.from_polymesh()* method. Cells of the same seed number are merged before meshing to prevent unnecessary internal geometry. Similarly, if the `material_type` of a phase is set to `amorphous`, then cells of the same phase number are also merged. Cells with the `material_type` set to `void` are treated as holes in MeshPy, resulting in voids in the output mesh.

5.3 File I/O & Plot Methods

There are file read and write functions associated with each of the classes listed above.

The read methods are:

- *SeedList.from_file()*
- *PolyMesh.from_file()*
- *TriMesh.from_file()*

The write methods are:

- *SeedList.write()*
- *PolyMesh.write()*

- *TriMesh.write()*

The read functions currently only support reading cache text files. The SeedList only writes to cache text files, while PolyMesh and TriMesh can output to several file formats.

The SeedList, PolyMesh, and TriMesh classes have the following plotting methods:

- *SeedList.plot()*
- *SeedList.plot_breakdown()*
- *PolyMesh.plot()*
- *PolyMesh.plot_facets()*
- *TriMesh.plot()*

These functions operate like the matplotlib `plt.plot` function in that they just plot to the current figure. You still need to add `plt.axis('equal')`, `plt.show()`, etc to format and view the plots.

5.4 Phase Dictionaries

Functions with phase information input require a list of dictionaries, one for each material phase. The dictionaries should be organized in a manner similar to the example below.

```
phase = {
        'name': 'Example Phase',
        'color': 'blue',
        'material_type': 'crystalline',
        'fraction': 0.5,
        'max_volume': 0.1,
        'shape': 'ellipse',
        'size': 1.2,
        'aspect_ratio': 2
}
```

The dictionary contains both data about the phase as a whole, such as its volume fraction and material type, and about the individual grains. The keywords `size` and `aspect_ratio` are keyword arguments for defining an *Ellipse*, so those are passed through to the Ellipse class when creating the seeds. For a non-uniform size (or aspect ratio) distribution, replace the constant value with a distribution from the SciPy `scipy.stats` module. For example:

```
import scipy.stats
size_dist = scipy.stats.uniform(loc=1, scale=0.4)
phase['size'] = size_dist
```

The `max_volume` option allows for maximum element volume controls to be phase-specific.

CHAPTER

SIX

OUTPUT FILE FORMATS

MicroStructPy creates output files for the seed geometries, polygonal meshes, the unstructured/triangular meshes, and verification data. Some of these outputs can be written in standard file formats, such as VTK. Output files with a `.txt` extension are custom and explained in the following sections.

6.1 List of Seeds

The *SeedList* class can write its contents to a file using the *SeedList.write()* method and be read from a file using the *SeedList.from_file()* method. The CLI reads from and writes to `seeds.txt` in a run's directory.

This file contains a printed list of all the seeds in the list. Specifically, the seeds are converted to strings and the strings are written to the file.

The file that results looks like:

```
Geometry: circle
Radius: 1
Center: (2, -1)
Phase: 0
Breakdown: ((2, -1, 1))
Position: (2, -1)
Geometry: ellipse
a: 3
b: 1.5
angle: -15
center: (-5 3)
phase: 1
breakdown: ((-5, 3, 1.5), (-4, 2.5, 1.3), (-6, 3.5, 1.3))
position: (-5, 3)
...
...
...
Geometry: <class name from microstructpy.geometry>
<param1>: <value1>
<param2>: <value2>
    ...
<paramN>: <valueN>
phase: <phase number>
breakdown: <circular/spherical breakdown of geometry>
position: <position of seed>
```

For more information on how each seed listing is converted back into an instance of the Seed class, see *Seed.from_str()*.

Note: For geomtries such as the circle and ellipse, it seems redundant to specify both the center and the position of the seed. The rationale is that some geometries may be specified by some other point instead of the center.

6.2 Polygonal Mesh

The polygonal mesh (or polyhedral mesh in 3D) can be written to and read from a `.txt` file. It can also be written to `.poly` files for 2D meshes, `.vtk` files for 2D and 3D meshes, and `.ply` files for any number of dimensions.

6.2.1 Text File

The text string output file is meant solely for saving the polygon/ polyhedron mesh as an intermediate step in the meshing process. The format for the text string file is:

```
Mesh Points: <numPoints>
    x1, y1(, z1)      <- optional tab at line start
    x2, y2(, z2)
    ...
    xn, yn(, zn)
Mesh Facets: <numFacets>
    f1_1, f1_2, f1_3, ...
    f2_1, f2_2, f2_3, ...
    ...
    fn_1, fn_2, fn_3, ...
Mesh Regions: <numRegions>
    r1_1, r1_2, r1_3, ...
    r2_1, r2_2, r2_3, ...
    ...
    rn_1, rn_2, rn_3, ...
Seed Numbers: <numRegions>
    s1
    s2
    ...
    sn
Phase Numbers: <numRegions>
    p1
    p2
    ...
    pn
```

For example:

```
Mesh Points: 4
    0.0, 0.0
    1.0, 0.0
    3.0, 2.0
    2.0, 2.0
Mesh Facets: 5
```

(continues on next page)

(continued from previous page)

```
    0, 1
    1, 2
    2, 3
    3, 0
    1, 3
Mesh Regions: 2
    0, 4, 3
    1, 2, 4
Seed Numbers: 2
    0
    1
Phase Numbers: 2
    0
    0
```

In this example, the polygon mesh contains a parallelogram that has been divided into two triangles. In general, the regions do not need to have the same number of facets. For 3D meshes, the mesh facets should be an ordered list of point indices that create the polygonal facet.

Note: Everything is indexed from 0 since this file is produced in Python.

6.2.2 Additional Formats

These additional output file formats are meant for processing and interpretation by other programs.

The `.poly` POLY file contains a planar straight line graph (PSLG) and can be read by the Triangle program from J. Shewchuk. See .poly files from the Triangle documentation for more details.

The `.vtk` VTK legacy file format supports POLYDATA datasets. The *facets* of a polyhedral mesh are written to the VTK file, but not the region data, seed numbers, or phase numbers. See File Formats for VTK Version 4.2 for a guide to the VTK legacy format.

The `.ply` polygon file format is intended for 3D scans but can also store the polygons and polyhedral facets of a polygonal mesh. See PLY - Polygon File Format for a description and examples of ply files.

6.3 Triangular Mesh

The triangular mesh (or tetrahedral mesh in 3D) can be written to and read from a `.txt` file. It can also be written to `.inp` Abaqus input files, `.vtk` files for 3D meshes, and `.node`/`.ele` files like Triangle and TetGen.

6.3.1 Text File

The organization of the triangular mesh text file is similar to the `meshpy.triangle.MeshInfo` and `meshpy.tet.MeshInfo` classes from MeshPy . The format for the text string file is:

```
Mesh Points: <numPoints>
    x1, y1(, z1)      <- optional tab at line start
    x2, y2(, z2)
    ...
    xn, yn(, zn)
Mesh Elements: <numElements>
    e1_1, e1_2, e1_3(, e1_4)
    e2_1, e2_2, e2_3(, e2_4)
    ...
    en_1, en_2, en_3(, en_4)
Element Attributes: <numElements>
    a1,
    a2,
    ...
    an
Facets: <numFacets>
    f1_1, f1_2(, f1_3)
    f2_1, f2_2(, f2_3)
    ...
    fn_1, fn_2(, fn_3)
Facet Attributes: <numFacets>
    a1,
    a2,
    ...
    an
```

In MicroStructPy, the element attribute is the seed number associated with the element. The facet attribute is the facet number from the polygonal mesh, so all of the triangular mesh facets with the same attribute make up a polygonal mesh facet.

Note: Everything is indexed from 0 since this file is produced in Python.

6.3.2 Additional Formats

Triangular and tetrahedral meshes can be output to additional file formats for processing and vizualization by other programs. These include Abaqus input files, TetGen/Triangle standard outputs, and the VTK legacy format.

The Abaqus input file option, `format='abaqus'` in *TriMesh.write()*, creates an input file for the mesh that defines each grain as its own part. It also creates surfaces between the grains and on the domain boundary for applying boundary conditions and loads.

The TetGen/Triangle file option, `format='tet/tri'`, creates `.node`, `.edge` (or `.face`), and `.ele` files. See Triangle and TetGen's File Formats for more details on these files and their format.

CHAPTER

SEVEN

API

7.1 microstructpy.cli

Command Line Interface.

This module contains the command line interface (CLI) for MicroStructPy. The CLI primarily reads XML input files and creates a microstructure according to those inputs. It can also run demo input files.

microstructpy.cli.**dict_convert**(*dictionary*, *filepath='.'*)

Convert dictionary from xmltodict

The xmltodict `parse` method creates dictionaries with values that are all strings, rather than strings, floats, ints, etc. This function recursively searches the dictionary for string values and attempts to convert the dictionary values.

If a dictionary contains the key `dist_type`, it is assumed that the corresponding name is a `scipy.stats` statistical distribution and the remaining keys are inputs for that distribution, with two exceptions. First, if the value of `dist_type` is `cdf`, then the remaining key should be `filename` and its value should be the path to a CSV file, where each row contains the (x, CDF) points along the CDF curve. Second, if the value of `dist_type` is `histogram`, then the remaining key should also be `filename` and its value should be the path to a CSV file. For the histogram, the first row of this CDF should be the *n* bin heights and the second row should be the *n+1* bin locations.

Additionally, if a key in the dictionary contains `filename` or `directory` and the value associated with that key is a relative path, then the filepath is converted from a relative to an absolute path using the `filepath` input as the reference point. This behavior can be switched off by setting `filepath=False`.

Parameters

- **dictionary** (*list*, *dict*, *or* *collections.OrderedDict*) – Dictionary or dictionaries to be converted.
- **filepath** (*str*) – *(optional)* Reference path to resolve relative paths.

Returns A copy of the input where the string values have been converted. If only one dict is passed into the function, then an instance of `collections.OrderedDict` is returned.

Return type list or collections.OrderedDict

microstructpy.cli.**input2dict**(*filename*, *root_tag='input'*)

Read input file into a dictionary

This function reads an input file and creates a dictionary of strings contained within the file.

Parameters **filename** – Name of the input file.

Returns Dictionary of input strings.

Return type collections.OrderedDict

microstructpy.cli.**main**()
CLI calling function

microstructpy.cli.**plot_poly**(*pmesh*, *phases*, *plot_files=['polymesh.png']*, *plot_axes=True*, *color_by='material'*, *colormap='viridis'*, ***edge_kwargs*)
Plot polygonal/polyhedral mesh

This function creates formatted plots of a *PolyMesh*.

Parameters

- **pmesh** (PolyMesh) – Polygonal mesh to plot.
- **phases** (*list*) – List of phase dictionaries. See *Phase Dictionaries* for more details.
- **plot_files** (*list*) – *(optional)* List of files to save the output plot. Defaults to saving the plot to `polymesh.png`.
- **plot_axes** (*bool*) – *(optional)* Flag to turn the axes on or off. True shows the axes, False removes them. Defaults to True.
- **color_by** (*str*) – *(optional)* {'material' | 'seed number' | 'material number'} Option to choose how the polygons/polyhedra are colored. Defaults to 'material'.
- **colormap** (*str*) – *(optional)* Name of the matplotlib colormap to color the seeds. Ignored if *color_by='material'*. Defaults to 'viridis', the standard matplotlib colormap. See Choosing Colormaps in Matplotlib for more details.
- ****edge_kwargs** – Additional keyword arguments that will be passed to *PolyMesh.plot_facets()* in 2D and *PolyMesh.plot()* in 3D.

microstructpy.cli.**plot_seeds**(*seeds*, *phases*, *domain*, *plot_files=[]*, *plot_axes=True*, *color_by='material'*, *colormap='viridis'*, ***edge_kwargs*)
Plot seeds

This function creates formatted plots of a *SeedList*.

Parameters

- **seeds** (SeedList) – Seed list to plot.
- **phases** (*list*) – List of phase dictionaries. See *Phase Dictionaries* for more details.
- **domain** (from *microstructpy.geometry*) – Domain geometry.
- **plot_files** (*list*) – *(optional)* List of files to save the output plot. Defaults to saving the plot to `seeds.png`.
- **plot_axes** (*bool*) – *(optional)* Flag to turn the axes on or off. True shows the axes, False removes them. Defaults to True.
- **color_by** (*str*) – *(optional)* {'material' | 'seed number' | 'material number'} Option to choose how the polygons/polyhedra are colored. Defaults to 'material'.
- **colormap** (*str*) – *(optional)* Name of the matplotlib colormap to color the seeds. Ignored if *color_by='material'*. Defaults to 'viridis', the standard matplotlib colormap. See Choosing Colormaps in Matplotlib for more details.
- ****edge_kwargs** – additional keyword arguments that will be passed to *SeedList.plot()*.

microstructpy.cli.**plot_tri**(*tmesh*, *phases*, *seeds*, *pmesh*, *plot_files=[]*, *plot_axes=True*, *color_by='material'*, *colormap='viridis'*, ***edge_kwargs*)
Plot seeds

This function creates formatted plots of a *TriMesh*.

Parameters

- **tmesh** (TriMesh) – Triangular mesh to plot.
- **phases** (*list*) – List of phase dictionaries. See *Phase Dictionaries* for more details.
- **seeds** (SeedList) – List of seed geometries.
- **pmesh** (PolyMesh) – Polygonal mesh from which `tmesh` was generated.
- **plot_files** (*list*) – *(optional)* List of files to save the output plot. Defaults to saving the plot to `trimesh.png`.
- **plot_axes** (*bool*) – *(optional)* Flag to turn the axes on or off. True shows the axes, False removes them. Defaults to True.
- **color_by** (*str*) – *(optional)* {'material' | 'seed number' | 'material number'} Option to choose how the polygons/polyhedra are colored. Defaults to 'material'.
- **colormap** (*str*) – *(optional)* Name of the matplotlib colormap to color the seeds. Ignored if *color_by='material'*. Defaults to 'viridis', the standard matplotlib colormap. See Choosing Colormaps in Matplotlib for more details.
- ****edge_kwargs** – Additional keyword arguments that will be passed to *TriMesh.plot()*.

microstructpy.cli.**read_input**(*filename*)

Convert input file to dictionary

This function reads an input file and parses it into a dictionary.

Parameters **filename** (*str*) – The name of an XML input file.

Returns Dictionary of run inputs.

Return type collections.OrderedDict

microstructpy.cli.**run**(*phases*, *domain*, *verbose=False*, *restart=True*, *directory='.'*, *filetypes={}*, *rng_seeds={}*, *plot_axes=True*, *rtol='fit'*, *edge_opt=False*, *edge_opt_n_iter=100*, *mesher='Triangle/TetGen'*, *mesh_max_volume=inf*, *mesh_min_angle=0*, *mesh_max_edge_length=inf*, *mesh_size=inf*, *verify=False*, *color_by='material'*, *colormap='viridis'*, *seeds_kwargs={}*, *poly_kwargs={}*, *tri_kwargs={}*)

Run MicroStructPy

This is the primary run function for the package. It performs these steps:

- Create a list of un-positioned seeds
- Position seeds in domain
- Create a polygon mesh from the seeds
- Create a triangle mesh from the polygon mesh
- (optional) Perform mesh verification

Parameters

- **phases** (*list or dict*) – Single phase dictionary or list of multiple phase dictionaries. See *Phase Dictionaries* for more details.
- **domain** (from *microstructpy.geometry*) – The geometry of the domain.
- **verbose** (*bool*) – *(optional)* Option to run in verbose mode. Prints status updates to the terminal. Defaults to False.

- **restart** (*bool*) – *(optional)* Option to run in restart mode. Saves caches at the end of each step and reads caches to restart the analysis. Defaults to True.
- **directory** (*str*) – *(optional)* File path where outputs will be saved. This path can either be relative to the current directory, or an absolute path. Defaults to the current working directory.
- **filetypes** (*dict*) – *(optional)* Filetypes for the output files. A dictionary containing many of the possible file types is:

```
filetypes = {'seeds': 'txt',
             'seeds_plot': ['eps',
                            'pdf',
                            'png',
                            'svg'],
             'poly': ['txt', 'ply', 'vtk'],
             'poly_plot': 'png',
             'tri': ['txt', 'abaqus', 'vtk'],
             'tri_plot': ['png', 'pdf'],
             'verify_plot': 'pdf'
             }
```

  If an entry is not included in the dictionary, then that output is not saved. Default is an empty dictionary. If *restart* is True, then 'txt' is added to the 'seeds', 'poly', and 'tri' fields.

- **rng_seeds** (*dict*) – *(optional)* The random number generator (RNG) seeds. The dictionary values should all be non-negative integers. Specifically, RNG seeds should be convertible to NumPy uint32. An example dictionary is:

```
rng_seeds = {'fraction': 0,
             'phase': 134092,
             'position': 1,
             'size': 95,
             'aspect_ratio': 2,
             'orienation': 2
             }
```

  If a seed is not specified, the default value is 0.

- **rtol** (*float or str*) – *(optional)* The relative overlap tolerance between seeds. This parameter should be between 0 and 1. The condition for two circles to overlap is:

$$||x_2 - x_1|| + \text{rtol}\,\min(r_1, r_2) < r_1 + r_2$$

  The default value is `'fit'`, which uses the mean and variance of the size distribution to estimate a value for rtol.

- **edge_opt** (*bool*) – *(optional)* This option will maximize the minimum edge length in the PolyMesh. The seeds associated with the shortest edge are displaced randomly to find improvement and this process iterates until *n_iter* attempts have been made for a given edge. Defaults to False.
- **edge_opt_n_iter** (*int*) – *(optional)* Maximum number of iterations per edge during optimization. Ignored if *edge_opt* set to False. Defaults to 100.
- **mesher** (*str*) – {'raster' | 'Triangle/TetGen' | 'Triangle' | 'TetGen' | 'gmsh'} specify the mesh generator. Default is 'Triangle/TetGen'.

- **mesh_max_volume** (*float*) – *(optional)* The maximum volume (area in 2D) of a mesh cell in the triangular mesh. Default is infinity, which turns off the maximum volume quality setting. Value should be strictly positive.
- **mesh_min_angle** (*float*) – *(optional)* The minimum interior angle, in degrees, of a cell in the triangular mesh. For 3D meshes, this is the dihedral angle between faces of the tetrahedron. Defaults to 0, which turns off the angle quality constraint. Value should be in the range 0-60.
- **mesh_max_edge_length** (*float*) – *(optional)* The maximum edge length of elements along grain boundaries. Currently only supported in 2D.
- **mesh_size** (*float*) – The target size of the mesh elements. This option is used with gmsh. Default is infinity, whihch turns off this control.
- **plot_axes** (*bool*) – *(optional)* Option to show the axes in output plots. When False, the plots are saved without axes and very tight borders. Defaults to True.
- **verify** (*bool*) – *(optional)* Option to verify the output mesh against the input phases. Defaults to False.
- **color_by** (*str*) – *(optional)* {'material' | 'seed number' | 'material number'} Option to choose how the polygons/polyhedra are colored. Defaults to 'material'.
- **colormap** (*str*) – *(optional)* Name of the matplotlib colormap to color the seeds. Ignored if *color_by='material'*. Defaults to 'viridis', the standard matplotlib colormap. See Choosing Colormaps in Matplotlib for more details.
- **seed_kwargs** (*dict*) – additional keyword arguments that will be passed to *SeedList.plot()*.
- **poly_kwargs** (*dict*) – Additional keyword arguments that will be passed to *PolyMesh.plot_facets()* in 2D and *PolyMesh.plot()* in 3D.
- **tri_kwargs** (*dict*) – Additional keyword arguments that will be passed to *TriMesh.plot()*.

microstructpy.cli.**run_file**(*filename*)

Run an input file

This function reads an input file and runs it through the standard workflow.

Parameters **filename** (*str*) – The name of an XML input file.

7.2 microstructpy.geometry

The geometry module contains classes for several 2D and 3D geometries. The module also contains some N-D geometries, which are inherited by the 2D and 3D geometries.

2D Geometries

- *microstructpy.geometry.Circle* † ‡
- *microstructpy.geometry.Ellipse* † ‡
- *microstructpy.geometry.Rectangle* † ‡
- *microstructpy.geometry.Square* † ‡

3D Geometries

- *microstructpy.geometry.Box* ‡

- *microstructpy.geometry.Cube* ‡
- *microstructpy.geometry.Ellipsoid* †
- *microstructpy.geometry.Sphere* †

ND Geometries

- *microstructpy.geometry.n_box.NBox*
- *microstructpy.geometry.n_sphere.NSphere*

†: These classes may be used to define seed particles.

‡: These classes may be used to define the microstructure domain.

To assist with creating geometries, a factory method is included in the module:

- *microstructpy.geometry.factory*

7.2.1 microstructpy.geometry.Box

class microstructpy.geometry.**Box**(***kwargs*)

Bases: *microstructpy.geometry.n_box.NBox*

This class contains a generic, 3D box. The position and dimensions of the box can be specified using any of the parameters below.

Without any parameters, this is a unit cube centered on the origin.

Parameters

- **side_lengths** (*list*) – *(optional)* Side lengths.
- **center** (*list*) – *(optional)* Center of box.
- **corner** (*list*) – *(optional)* Bottom-left corner.
- **limits** (*list*) – *(optional)* Bounds of box.
- **bounds** (*list*) – *(optional)* Alias for `limits`.

plot(***kwargs*)

Plot the box.

This function adds an `mpl_toolkits.mplot3d.art3d.Poly3DCollection` to the current axes. The keyword arguments are passed through to the Poly3DCollection.

Parameters ****kwargs** (*dict*) – Keyword arguments for Poly3DCollection.

within(*points*)

Test if points are within n-box.

This function tests whether a point or set of points are within the n-box. For the set of points, a list of booleans is returned to indicate which points are within the n-box.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

property bounds

(lower, upper) bounds of the box

Type list

property corner
bottom-left corner

Type list

property limits
(lower, upper) bounds of the box

Type list

property n_dim
number of dimensions, 3

Type int

property n_vol
area, volume of n-box

Type float

property sample_limits
(lower, upper) bounds of the sampling region of the box

Type list

property volume
volume of box, $V = l_1 l_2 l_3$

Type float

7.2.2 microstructpy.geometry.Circle

class microstructpy.geometry.**Circle**(***kwargs*)
Bases: *microstructpy.geometry.n_sphere.NSphere*

A 2D circle.

This class represents a two-dimensional circle. It is defined by a center point and size parameter, which can be either radius or diameter.

Without parameters, this returns a unit circle centered on the origin.

Parameters

- **r** (*float*) – *(optional)* The radius of the circle. Defaults to 1.
- **center** (*list*) – *(optional)* The coordinates of the center. Defaults to (0, 0).
- **diameter** – *(optional)* Alias for `2*r`.
- **radius** – *(optional)* Alias for `r`.
- **d** – *(optional)* Alias for `2*r`.
- **size** – *(optional)* Alias for `2*r`.
- **position** – *(optional)* Alias for `center`.

approximate()
Approximate the n-sphere with itself

Other geometries can be approximated by a set of circles or spheres. For the n-sphere, this approximation is exact.

Returns A list containing [(x, y, z, ..., r)]

Return type list

classmethod area_expectation(***kwargs*)

Expected value of area.

This function computes the expected value for the area of a circle. The keyword arguments are the same as the class parameters. The values can be constants (ints or floats), or a distribution from the SciPy `scipy.stats` module.

The expected value is computed by the following formula:

$$\mathbb{E}[A] = \pi\mathbb{E}[R^2] = \pi(\mu_R^2 + \sigma_R^2)$$

For example:

```
>>> from microstructpy.geometry import Circle
>>> Circle.area_expectation(r=1)
3.141592653589793
>>> from scipy.stats import norm
>>> Circle.area_expectation(r=norm(1, 1))
6.283185307179586
```

Parameters ****kwargs** – Keyword arguments, see *Circle*.

Returns Expected value of the area of the circle.

Return type float

classmethod best_fit(*points*)

Find n-sphere of best fit for set of points.

This function takes a list of points and computes an n-sphere of best fit, in an algebraic sense. This method was developed using the a published writeup, which was extended from 2D to ND.[1]

Parameters **points** (*list, numpy.ndarray*) – List of points to fit.

Returns An instance of the class that fits the points.

Return type *NSphere*

plot(***kwargs*)

Plot the circle.

This function adds a `matplotlib.patches.Circle` to the current axes. The keyword arguments are passed through to the circle patch.

Parameters ****kwargs** (*dict*) – Keyword arguments for matplotlib.

reflect(*points*)

Reflect points across surface.

This function reflects a point or set of points across the surface of the n-sphere. Points at the center of the n-sphere are not reflected.

Parameters **points** (*list or numpy.ndarray*) – Points to reflect.

Returns Reflected points.

Return type numpy.ndarray

[1] Circle fitting writup by Randy Bullock, https://dtcenter.org/met/users/docs/write_ups/circle_fit.pdf

within(*points*)
Test if points are within n-sphere.

This function tests whether a point or set of points are within the n-sphere. For the set of points, a list of booleans is returned to indicate which points are within the n-sphere.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

property area
area of cirle, $A = \pi r^2$

Type float

property bound_max
maximum bounding n-sphere

Type tuple

property bound_min
minimum interior n-sphere

Type tuple

property d
diameter of n-sphere.

Type float

property diameter
diameter of n-sphere.

Type float

property limits
list of (lower, upper) bounds for the bounding box

Type list

property n_dim
number of dimensions, 2

Type int

property position
position of n-sphere.

Type list

property radius
radius of n-sphere.

Type float

property sample_limits
list of (lower, upper) bounds for the sampling region

Type list

property size
size (diameter) of n-sphere.

Type float

property volume
alias for area

Type float

7.2.3 microstructpy.geometry.Cube

class microstructpy.geometry.**Cube**(***kwargs*)
A cube.

This class contains a generic, 3D cube. It is derived from the *Box* and contains the `side_length` property, rather than multiple side lengths.

Without any parameters, this is a unit cube centered on the origin.

Parameters

- **side_length** (*float*) – *(optional)* Side length.
- **center** (*list*, *tuple*, *numpy.ndarray*) – *(optional)* Center of box.
- **corner** (*list*, *tuple*, *numpy.ndarray*) – *(optional)* Bottom-left corner.

plot(***kwargs*)
Plot the box.

This function adds an `mpl_toolkits.mplot3d.art3d.Poly3DCollection` to the current axes. The keyword arguments are passed through to the Poly3DCollection.

Parameters ****kwargs** (*dict*) – Keyword arguments for Poly3DCollection.

within(*points*)
Test if points are within n-box.

This function tests whether a point or set of points are within the n-box. For the set of points, a list of booleans is returned to indicate which points are within the n-box.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

property bounds
(lower, upper) bounds of the box

Type list

property corner
bottom-left corner

Type list

property limits
(lower, upper) bounds of the box

Type list

property n_dim
number of dimensions, 3

Type int

property n_vol
area, volume of n-box

Type float

property sample_limits
(lower, upper) bounds of the sampling region of the box

Type list

property side_length
length of the side of the cube.

Type float

property volume
volume of box, $V = l_1 l_2 l_3$

Type float

7.2.4 microstructpy.geometry.Ellipse

class microstructpy.geometry.**Ellipse**(***kwargs*)
Bases: object

A 2-D ellipse geometry.

This class contains a 2-D ellipse. It is defined by a center point, axes and an orientation.

Without any parameters, the ellipse defaults to a unit circle.

Parameters

- **a** (*float*) – *(optional)* Semi-major axis of ellipse. Defaults to 1.
- **b** (*float*) – *(optional)* Semi-minor axis of ellipse. Defaults to 1.
- **center** (*list*) – *(optional)* The ellipse center. Defaults to (0, 0).
- **axes** (*list*) – *(optional)* A 2-element list of semi-axes, equivalent to `[a, b]`. Defaults to [1, 1].
- **size** (*float*) – *(optional)* The diameter of a circle with equivalent area. Defaults to 1.
- **aspect_ratio** (*float*) – *(optional)* The ratio of x-axis to y-axis length. Defaults to 1.
- **angle** (*float*) – *(optional)* The counterclockwise rotation angle, in degrees, measured from the +x axis.
- **angle_deg** (*float*) – *(optional)* The rotation angle, in degrees.
- **angle_rad** (*float*) – *(optional)* The rotation angle, in radians.
- **matrix** (*numpy.ndarray*) – *(optional)* The 2x2 rotation matrix.
- **orientation** (*numpy.ndarray*) – *(optional)* Alias for `matrix`.

approximate(*x1=None*)
Approximate ellipse with a set of circles.

This function converts an ellipse into a set of circles. It implements a published algorithm by Ilin and Bernacki.[1]

[1] Ilin, D.N., and Bernacki, M., "Advancing Layer Algorithm of Dense Ellipse Packing for Generating Statistically Equivalent Polygonal Structures," Granular Matter, vol. 18(3), pp. 43, 2016.

Example

```
>>> import matplotlib.pyplot as plt
>>> import microstructpy as msp
>>> import numpy as np
>>> ellipse = msp.geometry.Ellipse(a=3, b=1)
>>> approx = ellipse.approximate(0.7)
>>> approx
array([[ 0.        ,  0.        ,  1.        ],
       [ 0.7       ,  0.        ,  0.96889112],
       [ 1.38067777,  0.        ,  0.87276349],
       [ 2.00213905,  0.        ,  0.7063497 ],
       [ 2.5234414 ,  0.        ,  0.45169729],
       [ 2.66666667,  0.        ,  0.33333333],
       [-0.7       ,  0.        ,  0.96889112],
       [-1.38067777,  0.        ,  0.87276349],
       [-2.00213905,  0.        ,  0.7063497 ],
       [-2.5234414 ,  0.        ,  0.45169729],
       [-2.66666667,  0.        ,  0.33333333]])
>>> ellipse.plot(edgecolor='k', facecolor='none', lw=3)
>>> t = np.linspace(0, 2 * np.pi)
>>> for x, y, r in approx:
...     plt.plot(x + r * np.cos(t), y + r * np.sin(t), 'b')
>>> plt.xticks(np.unique(np.concatenate((approx[:, 0], (-3, 3)))))
>>> plt.yticks(np.unique(np.concatenate((approx[:, 1], (-1, 1)))))
>>> plt.axis('scaled')
>>> plt.grid(True, linestyle=':')
>>> plt.show()
```

Executing the code above produces Fig. 7.1.

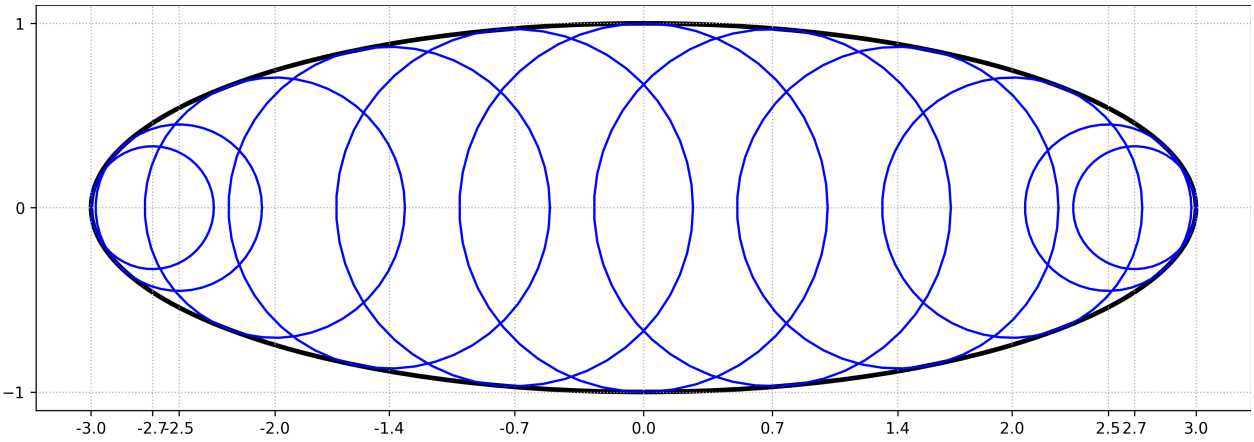

Fig. 7.1: Circular approximation of ellipse, after Ilin and Bernacki.

Parameters **x1** (*float or None*) – *(optional)* Position of the first circle along the +x axis. Defaults to 0.5x the shortest semi-axis.

Returns An Nx3 list of the (x, y, r) data of each circle approximating the ellipse.

Return type numpy.ndarray

Raises **AssertionError** – Thrown if max(a, b) < x1.

classmethod area_expectation(***kwargs*)

Expected value of area.

This function computes the expected value for the area of an ellipse. The keyword arguments are the same as the input parameters of the class. The keyword values can be either constants (ints or floats) or distributions from the SciPy `scipy.stats` module.

If an ellipse is specified by size, the expected value is computed as follows.

$$\begin{aligned}\mathbb{E}[A] &= \frac{\pi}{4}[S^2] \\ &= \frac{\pi}{4}(\mu_S^2 + \sigma_S^2)\end{aligned}$$

If the ellipse is specified by independent distributions for each semi-axis, the expected value is computed by:

$$\mathbb{E}[A] = \pi\,\mathbb{E}[AB] = \pi\mu_A\mu_B$$

If the ellipse is specified by the second semi-axis and the aspect ratio, the expected value is computed by:

$$\begin{aligned}\mathbb{E}[A] &= \pi\,\mathbb{E}[KB^2] \\ &= \pi\mu_K(\mu_B^2 + \sigma_B^2)\end{aligned}$$

Finally, if the ellipse is specified by the first semi-axis and the aspect ratio, the expected value is computed by Monte Carlo:

$$\begin{aligned}\mathbb{E}[A] &= \pi\,\mathbb{E}\left[\frac{A^2}{K}\right] \\ &\approx \frac{\pi}{n}\sum_{i=1}^{n}\frac{A_i}{K_i}\end{aligned}$$

where $n = 1000$.

Parameters ****kwargs** – Keyword arguments, see *microstructpy.geometry.Ellipse*.

Returns Expected value of the area of the ellipse.

Return type float

best_fit(*points*)

Find ellipse of best fit for points

This function computes the ellipse of best fit for a set of points. It calls the least-squares-ellipse-fitting package, which implements a published fitting algorithm in Python.[2]

The current instance of the class is used as an initial guess for the ellipse of best fit. Since an ellipse can be expressed multiple ways (e.g. rotate 90 degrees and flip the axes), this initial guess is used to choose from the multiple parameter sets.

Parameters **points** (*list or numpy.ndarray*) – An Nx2 list of points to fit.

Returns An instance of the class that best fits the points.

Return type *Ellipse*

plot(***kwargs*)

Plot the ellipse.

This function adds a `matplotlib.patches.Ellipse` patch to the current axes using matplotlib. The keyword arguments are passed to the patch.

[2] Halir, R., Flusser, J., "Numerically Stable Direct Least Squares Fitting of Ellipses," *6th International Conference in Central Europe on Computer Graphics and Visualization*, Vol. 98, 1998. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.7559&rep=rep1&type=pdf)

Parameters ****kwargs** (*dict*) – Keyword arguments for matplotlib.

reflect(*points*)
Reflect points across surface.

This function reflects a point or set of points across the surface of the ellipse. Points at the center of the ellipse are not reflected.

Parameters **points** (*list or numpy.ndarray*) – Nx2 list of points to reflect.

Returns Reflected points.

Return type numpy.ndarray

within(*points*)
Test if points are within ellipse.

This function tests whether a point or set of points are within the ellipse. For the set of points, a list of booleans is returned to indicate which points are within the ellipse.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Set to True for points in ellipse.

Return type bool or numpy.ndarray

property angle_deg
Rotation angle, in degrees

Type float

property angle_rad
Rotation angle, in radians

Type float

property area
Area of ellipse, $A = \pi ab$

Type float

property aspect_ratio
Ratio of x-axis length to y-axis length

Type float

property axes
List of semi-axes.

Type tuple

property bound_max
Maximum bounding circle of ellipse, (x, y, r)

Type tuple

property bound_min
Minimum interior circle of ellipse, (x, y, r)

Type tuple

property limits
List of (lower, upper) bounds for the bounding box

Type list

property matrix
Rotation matrix

Type numpy.ndarray

property n_dim
Number of dimensions, 2

Type int

property orientation
Rotation matrix

Type numpy.ndarray

property sample_limits
List of (lower, upper) bounds for the sampling region

Type list

property size
Diameter of equivalent area circle

Type float

property volume
Same as *microstructpy.geometry.Ellipse.area*

Type float

7.2.5 microstructpy.geometry.Ellipsoid

class microstructpy.geometry.**Ellipsoid**(***kwargs*)
Bases: object

A 3D Ellipsoid

This class contains the data and functions for a 3D ellispoid. It is defined by its center, axes, and orientation.

If multiple keywords are given for the shape of the ellipsoid, there is no guarantee for which keywords are used.

Parameters

- **a** (*float*) – *(optional)* First semi-axis of ellipsoid. Default is 1.
- **b** (*float*) – *(optional)* Second semi-axis of ellipsoid. Default is 1.
- **c** (*float*) – *(optional)* Third semi-axis of ellipsoid. Default is 1.
- **center** (*list*) – *(optional)* The ellipsoid center. Defaults to (0, 0, 0).
- **axes** (*list*) – *(optional)* List of 3 semi-axes. Defaults to (1, 1, 1).
- **size** (*float*) – *(optional)* The diameter of a sphere with equal volume. Defaults to 2.
- **ratio_ab** (*float*) – *(optional)* The ratio of a to b.
- **ratio_ac** (*float*) – *(optional)* The ratio of a to c.
- **ratio_bc** (*float*) – *(optional)* The ratio of b to c.
- **ratio_ba** (*float*) – *(optional)* The ratio of b to a.
- **ratio_ca** (*float*) – *(optional)* The ratio of c to a.
- **ratio_cb** (*float*) – *(optional)* The ratio of c to b.

- **rot_seq** (*list*) – *(optional)* List of rotations (deg). Each element of the list should be an (axis, angle) tuple. The options for the axis are: 'x', 'y', 'z', 1, 2, or 3. For example:

```
rot_seq = [('x', 10), (2, -20), ('z', 85), ('x', 21)]
```

- **rot_seq_deg** (*list*) – *(optional)* Alias for `rot_seq`, with degrees stated explicitly.
- **rot_seq_rad** (*list*) – *(optional)* Same format as `rot_seq`, except the angles are expressed in radians.
- **matrix** (*numpy.ndarray*) – *(optional)* A 3x3 rotation matrix expressing the orientation of the ellipsoid. Defaults to the identity.
- **position** – *(optional)* Alias for `center`.
- **orientation** – *(optional)* Alias for `matrix`.

approximate(*x1=None*)

Approximate Ellipsoid with Spheres

This function approximates the ellipsoid by a set of spheres. It does so by approximating the x-z and y-z elliptical cross sections with circles, then scaling those circles and promoting them to spheres.

See the documentation for *microstructpy.geometry.Ellipse.approximate()* for more details.

Parameters **x1** (*float*) – *(optional)* Center position of the first sphere. Default is 0.75x the minimum semi-axis.

Returns An Nx4 list of the (x, y, z, r) data of the spheres that approximate the ellipsoid.

Return type numpy.ndarray

best_fit(*points*)

Find ellipsoid of best fit.

This function takes a list of 3D points and computes the ellipsoid of best fit for the points. It uses a published algorithm to fit the ellipsoid, then attempts to define the axes in such a way that they most align with this ellipsoid's axes.[1]

Parameters **points** (*list*) – Points to fit ellipsoid

Returns The ellipsoid that best fits the points.

Return type *Ellipsoid*

plot(***kwargs*)

Plot the ellipsoid.

This function uses the `mpl_toolkits.mplot3d.Axes3D.plot_surface()` method to add an ellipsoid to the current axes. The keyword arguments are passes through to the plot_surface function.

Parameters ****kwargs** (*dict*) – Keyword arguments for matplotlib.

reflect(*points*)

Reflect points across surface.

This function reflects a point or set of points across the surface of the ellipsoid. Points at the center of the ellipsoid are not reflected.

Parameters **points** (*list or numpy.ndarray*) – Points to reflect.

Returns Reflected points.

[1] Turner, D. A., Anderson, I. J., Mason, J. C., and Cox, M. G., "An Algorithm for Fitting an Ellipsoid to Data," *National Physical Laboratory*, 1999, The United Kingdom. (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.2773&rep=rep1&type=pdf)

Return type numpy.ndarray

classmethod volume_expectation(***kwargs*)

Expected value of volume.

This function computes the expected value for the volume of an ellipsoid. The keyword arguments are the same as the input parameters for the class, *microstructpy.geometry.Ellipsoid*. The values for these keywords can be either constants or distributions from the SciPy `scipy.stats` module.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi ABC] \\ &= \frac{4}{3}\pi\mathbb{E}[A]\mathbb{E}[B]\mathbb{E}[C] \\ &= \frac{4}{3}\pi\mu_A\mu_B\mu_C\end{aligned}$$

If the ellisoid is specified by size and aspect ratios, then the expected volume is computed by:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{\pi}{6}S^3] \\ &= \frac{\pi}{6}(\mu_S^3 + 3\mu_S\sigma_S^2 + \gamma_{1,S}\sigma_S^3)\end{aligned}$$

If the ellipsoid is specified using a combination of semi-axes and aspect ratios, then the expected volume is the mean of 1000 random samples:

$$\mathbb{E}[V] \approx \frac{1}{n}\sum_{i=1}^{n} V_i$$

where $n = 1000$.

Parameters ****kwargs** – Keyword arguments, see *microstructpy.geometry.Ellipsoid*.

Returns Expected value of the volume of the sphere.

Return type float

within(*points*)

Test if points are within ellipsoid.

This function tests whether a point or set of points are within the ellipsoid. For the set of points, a list of booleans is returned to indicate which points are within the ellipsoid.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

property axes

the 3 semi-axes of the ellipsoid

Type tuple

property bound_max

maximum bounding sphere, (x, y, z, r)

Type tuple

property bound_min

minimum interior sphere, (x, y, z, r)

Type tuple

property coefficients
coefficients of equation, $(A, B, C, D, E, F, G, H, K, L)$ in $Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + Gx + Hy + Kz + L = 0$

Type tuple

property limits
List of (lower, upper) bounds for the bounding box

Type list

property matrix
A 3x3 rotation matrix

Type numpy.ndarray

property matrix_quadeq
Matrix of the quadratic equation

Type numpy.ndarray

property matrix_quadform
Matrix of the quadratic form

Type numpy.ndarray

property n_dim
number of dimensions, 3

Type int

property orientation
A 3x3 rotation matrix

Type numpy.ndarray

property ratio_ab
ratio of x-axis length to y-axis length

Type float

property ratio_ac
ratio of x-axis length to z-axis length

Type float

property ratio_ba
ratio of y-axis length to x-axis length

Type float

property ratio_bc
ratio of y-axis length to z-axis length

Type float

property ratio_ca
ratio of z-axis length to x-axis length

Type float

property ratio_cb
ratio of z-axis length to y-axis length

Type float

property rot_seq_deg
rotation sequence, with angles in degrees

Type list

property rot_seq_rad
rotation sequence, with angles in radiands

Type list

property sample_limits
List of (lower, upper) bounds for the sampling region

Type list

property size
diameter of equivalent volume sphere

Type float

property volume
volume of ellipsoid, $V = \frac{4}{3}\pi abc$

Type float

7.2.6 microstructpy.geometry.n_box.NBox

class microstructpy.geometry.n_box.**NBox**(***kwargs*)
Bases: object

N-dimensional box

This class contains a generic, n-dimensinoal box.

Parameters

- **side_lengths** (*list*) – *(optional)* Side lengths.
- **center** (*list*) – *(optional)* Center of box.
- **corner** (*list*) – *(optional)* Bottom-left corner.
- **bounds** (*list*) – *(optional)* Bounds of box. Expected in the form [(xmin, xmax), (ymin, ymax), ...].
- **limits** – Alias for *bounds*.
- **matrix** (*list, numpy.ndarray*) – *(optional)* Rotation matrix, nxn

within(*points*)
Test if points are within n-box.

This function tests whether a point or set of points are within the n-box. For the set of points, a list of booleans is returned to indicate which points are within the n-box.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

property bounds
(lower, upper) bounds of the box

Type list

property corner
bottom-left corner

Type list

property limits
(lower, upper) bounds of the box

Type list

property n_vol
area, volume of n-box

Type float

property sample_limits
(lower, upper) bounds of the sampling region of the box

Type list

7.2.7 microstructpy.geometry.n_sphere.NSphere

class microstructpy.geometry.n_sphere.**NSphere**(***kwargs*)
Bases: object

An N-dimensional sphere.

This class represents a generic, n-dimensional sphere. It is defined by a center point and size parameter, which can be either radius or diameter.

If multiple size or position keywords are given, there is no guarantee whhich keywords are used to create the geometry.

Parameters

- **r** (*float*) – *(optional)* The radius of the n-sphere. Defaults to 1.
- **center** (*list*) – *(optional)* The coordinates of the center. Defaults to [].
- **radius** – Alias for `r`.
- **d** – Alias for `2*r`.
- **diameter** – Alias for `2*r`.
- **size** – Alias for `2*r`.
- **position** – Alias for `center`.

approximate()
Approximate the n-sphere with itself

Other geometries can be approximated by a set of circles or spheres. For the n-sphere, this approximation is exact.

Returns A list containing [(x, y, z, ..., r)]

Return type list

classmethod best_fit(*points*)
Find n-sphere of best fit for set of points.

This function takes a list of points and computes an n-sphere of best fit, in an algebraic sense. This method was developed using the a published writeup, which was extended from 2D to ND.[1]

[1] Circle fitting writup by Randy Bullock, https://dtcenter.org/met/users/docs/write_ups/circle_fit.pdf

Parameters **points** (*list, numpy.ndarray*) – List of points to fit.

Returns An instance of the class that fits the points.

Return type *NSphere*

reflect(*points*)
Reflect points across surface.

This function reflects a point or set of points across the surface of the n-sphere. Points at the center of the n-sphere are not reflected.

Parameters **points** (*list or numpy.ndarray*) – Points to reflect.

Returns Reflected points.

Return type numpy.ndarray

within(*points*)
Test if points are within n-sphere.

This function tests whether a point or set of points are within the n-sphere. For the set of points, a list of booleans is returned to indicate which points are within the n-sphere.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

property bound_max
maximum bounding n-sphere

Type tuple

property bound_min
minimum interior n-sphere

Type tuple

property d
diameter of n-sphere.

Type float

property diameter
diameter of n-sphere.

Type float

property limits
list of (lower, upper) bounds for the bounding box

Type list

property position
position of n-sphere.

Type list

property radius
radius of n-sphere.

Type float

property sample_limits
list of (lower, upper) bounds for the sampling region

Type list

property size
size (diameter) of n-sphere.

Type float

7.2.8 microstructpy.geometry.Rectangle

class microstructpy.geometry.**Rectangle**(***kwargs*)
Bases: *microstructpy.geometry.n_box.NBox*

This class contains a generic, 2D rectangle. The position and dimensions of the box can be specified using any of the parameters below.

Without parameters, this returns a unit square centered on the origin.

Parameters

- **length** (*float*) – *(optional)* Length of the rectangle.
- **width** (*float*) – *(optional)* Width of the rectangle. *(optional)*
- **side_lengths** (*list*) – *(optional)* Side lengths. Defaults to (1, 1).
- **center** (*list*) – *(optional)* Center of rectangle. Defaults to (0, 0).
- **corner** (*list*) – *(optional)* Bottom-left corner.
- **bounds** (*list*) – *(optional)* Bounds of rectangle. Expected to be in the format [(xmin, xmax), (ymin, ymax)].
- **limits** – Alias for *bounds*.
- **angle** (*float*) – *(optional)* The rotation angle, in degrees.
- **angle_deg** (*float*) – *(optional)* The rotation angle, in degrees.
- **angle_rad** (*float*) – *(optional)* The rotation angle, in radians.
- **matrix** (*numpy.ndarray*) – *(optional)* The 2x2 rotation matrix.

approximate(*x1=None*)
Approximate rectangle with a set of circles.

This method approximates a rectangle with a set of circles. These circles are spaced uniformly along the long axis of the rectangle with distance `x1` between them.

Example

For a rectangle with length=2.5, width=1, and x1=0.3, the approximation would look like Fig. 7.2.

Parameters **x1** (*float or None*) – *(optional)* Spacing between the circles. If not specified, the spacing is 0.25x the length of the shortest side.

Returns An Nx3 array, where each row is a circle and the columns are x, y, and r.

Return type numpy.ndarray

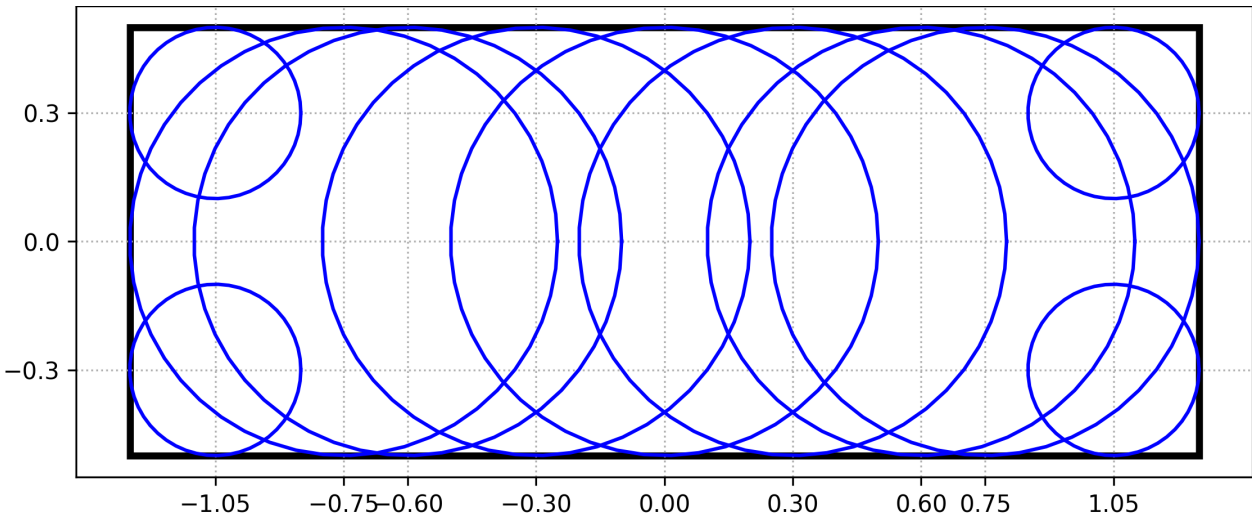

Fig. 7.2: Circular approximation of rectangle.

classmethod area_expectation(***kwargs*)

Expected area of rectangle

This method computes the expected area of a rectangle. There are two main ways to define the size of a rectangle: by the length and width and by the bounds. If the input rectangle is defined by length and width, the expected area is:

$$\mathbb{E}[A] = \mathbb{E}[LW] = \mu_L \mu_W$$

For the case where it is defined by upper and lower bounds:

$$\mathbb{E}[A] = \mathbb{E}[(X_U - X_L)(Y_U - Y_L)]$$

$$\mathbb{E}[A] = \mu_{X_U}\mu_{Y_U} - \mu_{X_U}\mu_{Y_L} - \mu_{X_L}\mu_{Y_U} + \mu_{X_L}\mu_{Y_L}$$

Example

```
>>> import scipy.stats
>>> import microstructpy as msp
>>> L = scipy.stats.uniform(loc=1, scale=2)
>>> W = scipy.stats.norm(loc=3.2, scale=5.1)
>>> L.mean() * W.mean()
6.4
>>> msp.geometry.Rectangle.area_expectation(length=L, width=W)
6.4
```

Parameters ****kwargs** – Keyword arguments, same as *Rectangle* but the inputs can be from the `scipy.stats` module.

Returns Expected/average area of rectangle.

Return type float

best_fit(*points*)

Find rectangle of best fit for points

Parameters **points** (*list*) – List of points to fit.

Returns an instance of the class that best fits the points.

Return type *Rectangle*

plot(***kwargs*)
Plot the rectangle.

This function adds a matplotlib.patches.Rectangle patch to the current axes. The keyword arguments are passed through to the patch.

Parameters ****kwargs** (*dict*) – Keyword arguments for the patch.

within(*points*)
Test if points are within n-box.

This function tests whether a point or set of points are within the n-box. For the set of points, a list of booleans is returned to indicate which points are within the n-box.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

property angle
Rotation angle of rectangle - degrees

Type float

property angle_deg
Rotation angle of rectangle - degrees

Type float

property angle_rad
Rotation angle of rectangle - radians

Type float

property area
Area of rectangle

Type float

property bounds
(lower, upper) bounds of the box

Type list

property corner
bottom-left corner

Type list

property length
Length of rectangle, side length along 1st axis

Type float

property limits
(lower, upper) bounds of the box

Type list

property n_dim
Number of dimensions, 2

Type int

property n_vol
area, volume of n-box

Type float

property sample_limits
(lower, upper) bounds of the sampling region of the box

Type list

property width
Width of rectangle, side length along 2nd axis

Type float

7.2.9 microstructpy.geometry.Sphere

class microstructpy.geometry.**Sphere**(***kwargs*)
Bases: *microstructpy.geometry.n_sphere.NSphere*

A 3D sphere.

This class represents a three-dimensional circle. It is defined by a center point and size parameter, which can be either radius or diameter.

Without input parameters, this defaults to a unit sphere centered at the origin.

Parameters

- **r** (*float*) – *(optional)* The radius of the sphere. Defaults to 1.
- **radius** (*float*) – *(optional)* Same as `r`.
- **d** (*float*) – *(optional)* Alias for `2*r`.
- **diameter** (*float*) – *(optional)* Alias for `2*r`.
- **size** (*float*) – *(optional)* Alias for `2*r`.
- **center** (*list*, *float*, *numpy.ndarray*) – *(optional)* The coordinates of the center. Defaults to `[0, 0, 0]`.
- **position** (*list*, *float*, *numpy.ndarray*) – *(optional)* Alias for `center`.

approximate()
Approximate the n-sphere with itself

Other geometries can be approximated by a set of circles or spheres. For the n-sphere, this approximation is exact.

Returns A list containing [(x, y, z, ..., r)]

Return type list

classmethod best_fit(*points*)
Find n-sphere of best fit for set of points.

This function takes a list of points and computes an n-sphere of best fit, in an algebraic sense. This method was developed using the a published writeup, which was extended from 2D to ND.[1]

[1] Circle fitting writup by Randy Bullock, https://dtcenter.org/met/users/docs/write_ups/circle_fit.pdf

Parameters **points** (*list, numpy.ndarray*) – List of points to fit.

Returns An instance of the class that fits the points.

Return type *NSphere*

plot(***kwargs*)
Plot the sphere.

This function uses the mpl_toolkits.mplot3d.Axes3D.plot_surface() method to add the sphere to the current axes. The keyword arguments are passed through to plot_surface.

Parameters ****kwargs** (*dict*) – Keyword arguments for plot_surface.

reflect(*points*)
Reflect points across surface.

This function reflects a point or set of points across the surface of the n-sphere. Points at the center of the n-sphere are not reflected.

Parameters **points** (*list or numpy.ndarray*) – Points to reflect.

Returns Reflected points.

Return type numpy.ndarray

classmethod volume_expectation(***kwargs*)
Expected value of volume.

This function computes the expected value for the volume of a sphere. The keyword arguments are identical to the *Sphere* function. The values for these keywords can be either constants or scipy.stats distributions.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi R^3] \\ &= \frac{4}{3}\pi\mathbb{E}[R^3] \\ &= \frac{4}{3}\pi(\mu_R^3 + 3\mu_R\sigma_R^2 + \gamma_{1,R}\sigma_R^3)\end{aligned}$$

Parameters ****kwargs** – Keyword arguments, see *microstructpy.geometry.Sphere*.

Returns Expected value of the volume of the sphere.

Return type float

within(*points*)
Test if points are within n-sphere.

This function tests whether a point or set of points are within the n-sphere. For the set of points, a list of booleans is returned to indicate which points are within the n-sphere.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

property bound_max
maximum bounding n-sphere

Type tuple

property bound_min
minimum interior n-sphere

Type tuple

property d
diameter of n-sphere.

Type float

property diameter
diameter of n-sphere.

Type float

property limits
list of (lower, upper) bounds for the bounding box

Type list

property n_dim
number of dimensions, 3

Type int

property position
position of n-sphere.

Type list

property radius
radius of n-sphere.

Type float

property sample_limits
list of (lower, upper) bounds for the sampling region

Type list

property size
size (diameter) of n-sphere.

Type float

property volume
volume of sphere

Type float

7.2.10 microstructpy.geometry.Square

class microstructpy.geometry.**Square**(***kwargs*)
A square.

This class contains a generic, 2D square. It is derived from the *microstructpy.geometry.Rectangle* class and contains the `side_length` property, rather than multiple side lengths.

Parameters

- **side_length** (*float*) – *(optional)* Side length. Defaults to 1.
- **center** (*list*) – *(optional)* Center of rectangle. Defaults to (0, 0).
- **corner** (*list*) – *(optional)* Bottom-left corner.

approximate(*x1=None*)

Approximate square with a set of circles

This method approximates a square with a set of circles. These circles are spaced uniformly along the edges of the square with distance `x1` between them.

Example

For a square with side_length=1, and x1=0.2, the approximation would look like Fig. 7.3.

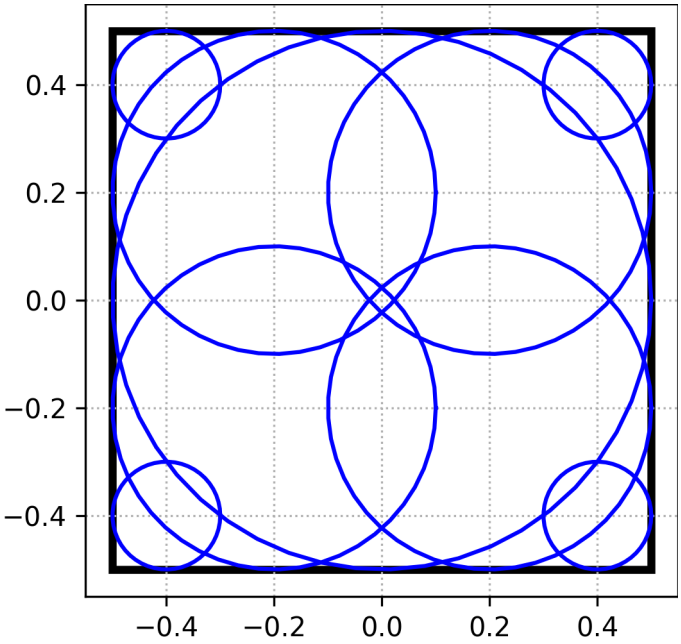

Fig. 7.3: Circular approximation of square.

Parameters **x1** (*float or None*) – *(optional)* Spacing between the circles. If not specified, the spacing is 0.25x the side length.

Returns An Nx3 array, where each row is a circle and the columns are x, y, and r.

Return type numpy.ndarray

classmethod area_expectation(***kwargs*)

Expected area of square

This method computes the expected area of a square with distributed side length. The expectation is:

$$\mathbb{E}[A] = \mathbb{E}[S^2] = \mu_S^2 + \sigma_S^2$$

Example

```
>>> import scipy.stats
>>> import microstructpy as msp
>>> S = scipy.stats.expon(scale=2)
>>> S.mean()^2 + S.var()
8.0
>>> msp.geometry.Square.area_expectation(side_length=S)
8.0
```

Parameters ****kwargs** – Keyword arguments, same as *Square* but the inputs can be from the `scipy.stats` module.

Returns Expected/average area of the square.

Return type float

best_fit(*points*)

Find rectangle of best fit for points

Parameters **points** (*list*) – List of points to fit.

Returns an instance of the class that best fits the points.

Return type *Rectangle*

plot(***kwargs*)

Plot the rectangle.

This function adds a `matplotlib.patches.Rectangle` patch to the current axes. The keyword arguments are passed through to the patch.

Parameters ****kwargs** (*dict*) – Keyword arguments for the patch.

within(*points*)

Test if points are within n-box.

This function tests whether a point or set of points are within the n-box. For the set of points, a list of booleans is returned to indicate which points are within the n-box.

Parameters **points** (*list or numpy.ndarray*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

property angle

Rotation angle of rectangle - degrees

Type float

property angle_deg

Rotation angle of rectangle - degrees

Type float

property angle_rad

Rotation angle of rectangle - radians

Type float

property area

Area of rectangle

Type float

property bounds
(lower, upper) bounds of the box

Type list

property corner
bottom-left corner

Type list

property length
Length of rectangle, side length along 1st axis

Type float

property limits
(lower, upper) bounds of the box

Type list

property n_dim
Number of dimensions, 2

Type int

property n_vol
area, volume of n-box

Type float

property sample_limits
(lower, upper) bounds of the sampling region of the box

Type list

property side_length
length of the side of the square.

Type float

property width
Width of rectangle, side length along 2nd axis

Type float

7.2.11 microstructpy.geometry.factory

geometry.**factory**(***kwargs*)
Factory method for geometries.

This function returns a geometry based on a string containing the name of the geometry and keyword arguments defining the geometry.

Note: The function call is `factory(name, **kwargs)`. Sphinx autodocs has dropped the first parameter.

Parameters

- **name** (*str*) – {'box' | 'cube' | 'ellipse' | 'ellipsoid' | 'circle' | 'rectangle' | 'square' | 'sphere'}
  Name of geometry.

- ****kwargs** (*dict*) – Arguments defining the geometry.

7.3 microstructpy.meshing

The meshing module contains three mesh classes, the *PolyMesh*, the *RasterMesh* and the *TriMesh*. The polygonal mesh contains a 2D or 3D tessellation of the microstructure domain, while the raster aand triangular meshes are more suitable for direct numerical simulation (finite element analysis).

7.3.1 microstructpy.meshing.PolyMesh

class microstructpy.meshing.**PolyMesh**(*points*, *facets*, *regions*, *seed_numbers=None*, *phase_numbers=None*, *facet_neighbors=None*, *volumes=None*)

Bases: object

Polygonal/Polyhedral mesh.

The PolyMesh class contains the points, edges, regions, etc. in a polygon (2D) or polyhedron (3D) mesh.

The points attribute is a numpy array containing the (x, y) or (x, y, z) coordinates of each point in the mesh. This is the only attribute that contains floating point numbers. The rest contain indices/integers.

The facets attribute describes the interfaces between the polygons/ polyhedra. In 2D, these interfaces are line segments and each facet contains the indices of the points at each end of the line segment. These indices are unorderd. In 3D, the interfaces are polygons so each facet contains the indices of the points on that polygon. These indices are ordered such that neighboring keypoints are connected by line segments that form the polygon.

The regions attribute contains the area (2D) or volume (3D). In 2D, a region is given by an ordered list of facets, or edges, that enclose the polygon. In 3D, the region is given by an un-ordered list of facets, or polygons, that enclose the polyhedron.

For each region, there is also an associated seed number and material phase. These data are stored in the seed_number and phase_number attributes, which have the same length as the regions list.

Parameters

- **points** (*list or numpy.ndarray*) – An Nx2 or Nx3 array of coordinates in the mesh.
- **facets** (*list*) – List of facets between regions. In 2D, this is a list of edges (Nx2). In 3D, this is a list of 3D polygons.
- **regions** (*list*) – A list of polygons (2D) or polyhedra (3D), with each element of the list being a list of facet indices.
- **seed_numbers** (*list or numpy.ndarray*) – *(optional)* The seed number associated with each region. Defaults to 0 for all regions.
- **phase_numbers** (*list or numpy.ndarray*) – *(optional)* The phase number associated with each region. Defaults to 0 for all regions.
- **facet_neighbors** (*list or numpy.ndarray*) – *(optional)* The region numbers on either side of each facet. If not givien, a neighbor list is computed from `regions`.
- **volumes** (*list or numpy.ndarray*) – *(optional)* The area/volume of each region. If not given, region volumes are calculated based on `points`, `facets`, and `regions`.

classmethod from_file(*filename*)

Read PolyMesh from file.

This function reads in a polygon mesh from a file and creates an instance from that file. Currently the only supported file type is the output from *write()* with the `format='txt'` option.

Parameters **filename** (*str*) – Name of file to read from.

Returns The instance of the class written to the file.

Return type *PolyMesh*

classmethod from_seeds(*seedlist*, *domain*, *edge_opt=False*, *n_iter=100*, *verbose=False*)
Create from *SeedList* and a domain.

This function creates a polygon/polyhedron mesh from a seed list and a domain. It relies on the pyvoro package, which wraps Voro++. The mesh is a Voronoi power diagram / Laguerre tessellationself.

The pyvoro package operates on rectangular domains, so other domains are meshed in 2D by meshing in a bounding box then the boundary cells are clipped to the domain boundary. Currently non-rectangular domains in 3D are not supported.

This function also includes the option to maximize the shortest edges in the polygonal/polyhedral mesh. Short edges cause numerical issues in finite element analysis - setting *edge_opt* to True can improve mesh quality with minimal changes to the microstructure.

Parameters

- **seedlist** (*SeedList*) – A list of seeds in the microstructure.
- **domain** (from *microstructpy.geometry*) – The domain to be filled by the seed.
- **edge_opt** (*bool*) – *(optional)* This option will maximize the minimum edge length in the PolyMesh. The seeds associated with the shortest edge are displaced randomly to find improvement and this process iterates until *n_iter* attempts have been made for a given edge. Defaults to False.
- **n_iter** (*int*) – *(optional)* Maximum number of iterations per edge during optimization. Ignored if *edge_opt* set to False. Defaults to 100.
- **verbose** (*bool*) – *(optional)* Print status of edge optimization to screen. Defaults to False.

Returns A polygon/polyhedron mesh.

Return type *PolyMesh*

plot(*index_by='seed'*, *material=[]*, *loc=0*, ***kwargs*)
Plot the mesh.

This function plots the polygon mesh. In 2D, this creates a class:*matplotlib.collections.PolyCollection* and adds it to the current axes. In 3D, it creates a `mpl_toolkits.mplot3d.art3d.Poly3DCollection` and adds it to the current axes. The keyword arguments are passed though to matplotlib.

Parameters

- **index_by** (*str*) – *(optional)* {'facet' | 'material' | 'seed'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='material', color=['blue', 'red'])` will plot the regions with `phase_number` equal to 0 in blue, and regions with `phase_number` equal to 1 in red. The facet option is only available for 3D plots. Defaults to 'seed'.
- **material** (*list*) – *(optional)* Names of material phases. One entry per material phase (the `index_by` argument is ignored). If this argument is set, a legend is added to the plot with one entry per material.

- **loc** (*int or str*) – *(optional)* The location of the legend, if 'material' is specified. This argument is passed directly through to `matplotlib.pyplot.legend()`. Defaults to 0, which is 'best' in matplotlib.
- ****kwargs** – Keyword arguments for matplotlib.

plot_facets(*index_by='seed'*, *hide_interior=True*, ***kwargs*)

Plot PolyMesh facets.

This function plots the facets of the polygon mesh, rather than the regions. In 2D, it adds a `matplotlib.collections.LineCollection` to the current axes. In 3D, it adds a `mpl_toolkits.mplot3d.art3d.Poly3DCollection` with `facecolors='none'`. The keyword arguments are passed though to matplotlib.

Parameters

- **index_by** (*str*) – *(optional)* {'facet' | 'material' | 'seed'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='material', color=['blue', 'red'])` will plot the regions with `phase_number` equal to 0 in blue, and regions with `phase` equal to 1 in red. The facet option is only available for 3D plots. Defaults to 'seed'.
- **hide_interior** (*bool*) – If True, removes interior facets from the output plot. This avoids occasional matplotlib issue where interior facets are shown in output plots.
- ****kwargs** (*dict*) – Keyword arguments for matplotlib.

write(*filename*, *format='txt'*)

Write the mesh to a file.

This function writes the polygon/polyhedron mesh to a file. See the *Polygonal Mesh* section of the *Output File Formats* guide for more information about the available output file formats.

Parameters

- **filename** (*str*) – Name of the file to be written.
- **format** (*str*) – *(optional)* {'txt' | 'poly' | 'ply' | 'vtk' } Format of the data in the file. Defaults to `'txt'`.

7.3.2 microstructpy.meshing.RasterMesh

class microstructpy.meshing.**RasterMesh**(*points*, *elements*, *element_attributes=None*, *facets=None*, *facet_attributes=None*)

Raster mesh.

The RasterMesh class contains the points and elements in a raster mesh, also called an regular grid.

The points attribute is an Nx2 or Nx3 list of points in the mesh. The elements attribute contains the Nx4 or Nx8 list of the points at the corners of each pixel/voxel. A list of facets can also be included, though it is optional and does not need to include every facet in the mesh. Attributes can also be assigned to the elements and facets, though they are also optional.

Parameters

- **points** (*list, numpy.ndarray*) – List of coordinates in the mesh.
- **elements** (*list, numpy.ndarray*) – List of indices of the points at the corners of each element. The shape should be Nx3 in 2D or Nx4 in 3D.
- **element_attributes** (*list, numpy.ndarray*) – *(optional)* A number associated with each element. Defaults to None.

- **facets** (*list, numpy.ndarray*) – *(optional)* A list of facets in the mesh. The shape should be Nx2 in 2D or Nx3 in 3D. Defaults to None.
- **facet_attributes** (*list, numpy.ndarray*) – *(optional)* A number associated with each facet. Defaults to None.

as_array(*element_attributes=True*)

numpy.ndarray containing element attributes.

Array contains -1 where there are no elements (e.g. circular domains).

Parameters **element_attributes** (*bool*) – *(optional)* Flag to return element attributes in the array. Set to True return attributes and set to False to return element indices. Defaults to True.

Returns Array of values of element atttributes, or indices.

Return type numpy.ndarray

classmethod from_file(*filename*)

Read TriMesh from file.

This function reads in a triangular mesh from a file and creates an instance from that file. Currently the only supported file type is the output from *write()* with the `format='str'` option.

Parameters **filename** (*str*) – Name of file to read from.

Returns An instance of the class.

Return type *TriMesh*

classmethod from_polymesh(*polymesh*, *mesh_size*, *phases=None*)

Create RasterMesh from PolyMesh.

This constuctor creates a raster mesh from a polygon mesh (*PolyMesh*). Polygons of the same seed number are merged and the element attribute is set to the seed number it is within. The facets between seeds are saved to the mesh and the index of the facet is stored in the facet attributes.

Since the PolyMesh can include phase numbers for each region, additional information about the phases can be included as an input. The "phases" input should be a list of material phase dictionaries, formatted according to the *Phase Dictionaries* guide.

The mesh_size option determines the side length of each pixel/voxel. Element attributes are sampled at the center of each pixel/voxel. If an edge of a domain is not an integer multiple of the mesh_size, it will be clipped. For example, if mesh_size is 3 and an edge has bounds [0, 11], the sides of the pixels will be at 0, 3, 6, and 9 while the centers of the pixels will be at 1.5, 4.5, 7.5.

The phase type option can take one of several values, described below.

- **crystalline**: granular, solid
- **amorphous**: glass, matrix
- **void**: crack, hole

The **crystalline** option creates a mesh where cells of the same seed number are merged, but cells are not merged across seeds. _This is the default material type._

The **amorphous** option creates a mesh where cells of the same phase number are merged to create an amorphous region in the mesh.

Finally, the **void** option will merge neighboring void cells and treat them as holes in the mesh.

Parameters

- **polymesh** (*PolyMesh*) – A polygon/polyhedron mesh.

- **mesh_size** (*float*) – The side length of each pixel/voxel.
- **phases** (*list*) – *(optional)* A list of dictionaries containing options for each phase. Default is `{'material_type': 'solid', 'max_volume': float('inf')}`.

plot(*index_by='element'*, *material=[]*, *loc=0*, ***kwargs*)

Plot the mesh.

This method plots the mesh using matplotlib. In 2D, this creates a `matplotlib.collections.PolyCollection` and adds it to the current axes. In 3D, it creates a `mpl_toolkits.mplot3d.axes3d.Axes3D.voxels()` and adds it to the current axes. The keyword arguments are passed though to matplotlib.

Parameters

- **index_by** (*str*) – *(optional)* {'element' | 'attribute'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='attribute', color=['blue', 'red'])` will plot the elements with `element_attribute` equal to 0 in blue, and elements with `element_attribute` equal to 1 in red. Defaults to 'element'.
- **material** (*list*) – *(optional)* Names of material phases. One entry per material phase (the `index_by` argument is ignored). If this argument is set, a legend is added to the plot with one entry per material. Note that the `element_attributes` must be the material numbers for the legend to be formatted properly.
- **loc** (*int or str*) – *(optional)* The location of the legend, if 'material' is specified. This argument is passed directly through to `matplotlib.pyplot.legend()`. Defaults to 0, which is 'best' in matplotlib.
- ****kwargs** – Keyword arguments that are passed through to matplotlib.

write(*filename*, *format='txt'*, *seeds=None*, *polymesh=None*)

Write mesh to file.

This function writes the contents of the mesh to a file. The format options are 'abaqus', 'txt', and 'vtk'. See the *Triangular Mesh* section of the *Output File Formats* guide for more details on these formats.

VTK files use the *RECTILINEAR_GRID* data type.

Parameters

- **filename** (*str*) – The name of the file to write.
- **format** (*str*) – {'abaqus' | 'txt' | 'vtk'} *(optional)* The format of the output file. Default is 'txt'.
- **seeds** (`SeedList`) – *(optional)* List of seeds. If given, VTK files will also include the phase number of of each element in the mesh. This assumes the `element_attributes` field contains the seed number of each element.
- **polymesh** (`PolyMesh`) – *(optional)* Polygonal mesh used for generating the raster mesh. If given, will add surface unions to Abaqus files - for easier specification of boundary conditions.

property mesh_size

Side length of elements.

7.3.3 microstructpy.meshing.TriMesh

class microstructpy.meshing.**TriMesh**(*points*, *elements*, *element_attributes=None*, *facets=None*, *facet_attributes=None*)

Bases: object

Triangle/Tetrahedron mesh.

The TriMesh class contains the points, facets, and elements in a triangle/ tetrahedron mesh, also called an unstructured grid.

The points attribute is an Nx2 or Nx3 list of points in the mesh. The elements attribute contains the Nx3 or Nx4 list of the points at the corners of each triangle/tetrahedron. A list of facets can also be included, though it is optional and does not need to include every facet in the mesh. Attributes can also be assigned to the elements and facets, though they are also optional.

Parameters

- **points** (*list, numpy.ndarray*) – List of coordinates in the mesh.
- **elements** (*list, numpy.ndarray*) – List of indices of the points at the corners of each element. The shape should be Nx3 in 2D or Nx4 in 3D.
- **element_attributes** (*list, numpy.ndarray*) – *(optional)* A number associated with each element. Defaults to None.
- **facets** (*list, numpy.ndarray*) – *(optional)* A list of facets in the mesh. The shape should be Nx2 in 2D or Nx3 in 3D. Defaults to None.
- **facet_attributes** (*list, numpy.ndarray*) – *(optional)* A number associated with each facet. Defaults to None.

classmethod from_file(*filename*)

Read TriMesh from file.

This function reads in a triangular mesh from a file and creates an instance from that file. Currently the only supported file type is the output from *write()* with the `format='str'` option.

Parameters **filename** (*str*) – Name of file to read from.

Returns An instance of the class.

Return type *TriMesh*

classmethod from_polymesh(*polymesh*, *phases=None*, *mesher='Triangle/Tetgen'*, *min_angle=0*, *max_volume=inf*, *max_edge_length=inf*, *mesh_size=inf*)

Create TriMesh from PolyMesh.

This constuctor creates a triangle/tetrahedron mesh from a polygon mesh (*PolyMesh*). Polygons of the same seed number are merged and the element attribute is set to the seed number it is within. The facets between seeds are saved to the mesh and the index of the facet is stored in the facet attributes.

Since the PolyMesh can include phase numbers for each region, additional information about the phases can be included as an input. The "phases" input should be a list of material phase dictionaries, formatted according to the *Phase Dictionaries* guide.

The minimum angle, maximum volume, and maximum edge length options provide quality controls for the mesh. The phase type option can take one of several values, described below.

- **crystalline**: granular, solid
- **amorphous**: glass, matrix
- **void**: crack, hole

The **crystalline** option creates a mesh where cells of the same seed number are merged, but cells are not merged across seeds. _This is the default material type._

The **amorphous** option creates a mesh where cells of the same phase number are merged to create an amorphous region in the mesh.

Finally, the **void** option will merge neighboring void cells and treat them as holes in the mesh.

Parameters

- **polymesh** (PolyMesh) – A polygon/polyhedron mesh.
- **phases** (*list*) – *(optional)* A list of dictionaries containing options for each phase. Default is `{'material_type': 'solid', 'max_volume': float('inf')}`.
- **mesher** (*str*) – {'Triangle/TetGen' | 'Triangle' | 'TetGen' | 'gmsh'} specify the mesh generator. Default is 'Triangle/TetGen'.
- **min_angle** (*float*) – The minimum interior angle, in degrees, of an element. This option is used with Triangle or TetGen and in 3D is the minimum *dihedral* angle. Defaults to 0.
- **max_volume** (*float*) – The default maximum cell volume, used if one is not set for each phase. This option is used with Triangle or TetGen. Defaults to infinity, which turns off this control.
- **max_edge_length** (*float*) – The maximum edge length of elements along grain boundaries. This option is used with Triangle and gmsh. Defaults to infinity, which turns off this control.
- **mesh_size** (*float*) – The target size of the mesh elements. This option is used with gmsh. Default is infinity, whihch turns off this control.

plot(*index_by='element'*, *material=[]*, *loc=0*, ***kwargs*)

Plot the mesh.

This method plots the mesh using matplotlib. In 2D, this creates a matplotlib.collections.PolyCollection and adds it to the current axes. In 3D, it creates a mpl_toolkits.mplot3d.art3d.Poly3DCollection and adds it to the current axes. The keyword arguments are passed though to matplotlib.

Parameters

- **index_by** (*str*) – *(optional)* {'element' | 'attribute'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='attribute', color=['blue', 'red'])` will plot the elements with `element_attribute` equal to 0 in blue, and elements with `element_attribute` equal to 1 in red. Note that in 3D the facets are plotted instead of the elements, so kwarg lists must be based on `facets` and `facet_attributes`. Defaults to 'element'.
- **material** (*list*) – *(optional)* Names of material phases. One entry per material phase (the `index_by` argument is ignored). If this argument is set, a legend is added to the plot with one entry per material. Note that the `element_attributes` in 2D or the `facet_attributes` in 3D must be the material numbers for the legend to be formatted properly.
- **loc** (*int or str*) – *(optional)* The location of the legend, if 'material' is specified. This argument is passed directly through to matplotlib.pyplot.legend(). Defaults to 0, which is 'best' in matplotlib.
- ****kwargs** – Keyword arguments that are passed through to matplotlib.

write(*filename*, *format='txt'*, *seeds=None*, *polymesh=None*)

Write mesh to file.

This function writes the contents of the mesh to a file. The format options are 'abaqus', 'tet/tri', 'txt', and 'vtk'. See the *Triangular Mesh* section of the *Output File Formats* guide for more details on these formats.

Parameters

- **filename** (*str*) – The name of the file to write. In the cases of TetGen/Triangle, this is the basename of the files.
- **format** (*str*) – {'abaqus' | 'tet/tri' | 'txt' | 'vtk'} *(optional)* The format of the output file. Default is 'txt'.
- **seeds** (SeedList) – *(optional)* List of seeds. If given, VTK files will also include the phase number of of each element in the mesh. This assumes the `element_attributes` field contains the seed number of each element.
- **polymesh** (PolyMesh) – *(optional)* Polygonal mesh used for generating the triangular mesh. If given, will add surface unions to Abaqus files - for easier specification of boundary conditions.

7.4 microstructpy.seeding

The seeding module contains two classes: *Seed* and *SeedList*. The single *Seed* contains the geometry, phase number, and position of a seed, while a *SeedList* functions much like a list of *Seed* instances, but with more methods.

7.4.1 microstructpy.seeding.Seed

class microstructpy.seeding.**Seed**(*seed_geometry*, *phase=0*, *breakdown=None*, *position=None*)

Bases: object

Seed particle

The Seed class contains the information about a single seed in the mesh. These seeds have a geometry (from *microstructpy.geometry*), a phase number, a breakdown, and a position.

Parameters

- **seed_geometry** (from *microstructpy.geometry*) – The geometry of the seed.
- **phase** (*int*) – *(optional)* The phase number of the seed. Defaults to 0.
- **breakdown** (*list or numpy.ndarray*) – *(optional)* The circle/sphere approximation of this grain. The format for this input is:

```
#                  x   y  r
breakdown_2D = [( 2,  3, 1),
                ( 0,  0, 4),
                (-2,  4, 8)]

#                  x   y   z  r
breakdown_3D = [( 3, -1,  2, 1),
                ( 0,  2, -1, 1)]
```

  The default behavior is to call the `approximate()` function of the geometry.

- **position** (*list or numpy.ndarray*) – *(optional)* The coordinates of the seed. See *position* for more details. Defaults to the origin.

classmethod factory(*seed_type*, *phase=0*, *breakdown=None*, *position=None*, ***kwargs*)
Factory method for seeds

This function returns a seed based on the seed type and keyword arguments associated with that type. The currently supported types are:

- circle
- ellipse
- ellipsoid
- rectangle
- sphere
- square

If the seed_type is not on this list, an error is thrown.

Parameters

- **seed_type** (*str*) – type of seed, from list above.
- **phase** (*int*) – *(optional)* Material phase number of seed. Defaults to 0.
- **breakdown** (*list or numpy.ndarray*) – *(optional)* List of circles or spheres that approximate the geometry. The list should be formatted as follows:

```
breakdown = [(x1, y1, z1, r1),
             (x2, y2, z2, r2),
             ...]
```

  The breakdown will be automatically generated if not provided.
- **position** (*list or numpy.ndarray*) – *(optional)* The coordinates of the seed. Default is the origin.
- ****kwargs** – Keyword arguments that define the size, shape, etc of the seed geometry.

Returns An instance of the class.

Return type *Seed*

classmethod from_str(*seed_str*)
Create seed from a string.

This method creates a seed particle from a string representation. This is used when reading in seeds from a file.

Parameters **seed_str** (*str*) – String representation of the seed.

Returns An instance of a Seed derived class.

Return type *Seed*

plot(***kwargs*)
Plot the seed

This function plots the geometry of the seed. The keyword arguments are passed through to matplotlib. See the plot methods in *microstructpy.geometry* for more details.

Parameters ****kwargs** – Plotting keyword arguments.

plot_breakdown(***kwargs*)
Plot breakdown of seed

This function plots the circle/sphere breakdown of the seed. In 2D, this adds a matplotlib.collections.PatchCollection to the current axes.

Parameters ****kwargs** – Matplotlib keyword arguments.

property limits
The (lower, upper) bounds of the seed

Type list

property position
Position of the seed

This is the location of the seed center.

Note: If the breakdown of the seed has been populated, the setter function will update the position of the center and translate the breakdown circles/spheres.

---

property volume
The area (2D) or volume (3D) of the seed

Type float

7.4.2 microstructpy.seeding.SeedList

class microstructpy.seeding.**SeedList**(*seeds=[]*)
Bases: object

List of seed geometries.

The SeedList is similar to a standard Python list, but contains instances of the *Seed* class. It can be generated from a list of Seeds, by creating enough seeds to fill a given volume, or by reading the content of a cache text file.

Parameters **seeds** (*list*) – *(optional)* List of *Seed* instances.

append(*seed*)
Append seed

This function appends a seed to the list.

Parameters **seed** (Seed) – The seed to append to the list

extend(*seeds*)
Extend seed list

This function adds a list of seeds to the end of the seed list.

Parameters **seeds** (*list or* SeedList) – List of seeds

classmethod from_file(*filename*)
Create seed list from file containing list of seeds

This function creates a seed list from a file containing a list of seeds. This file should contain the string representations of seeds, separated by a newline character (which is the behavior of *write()*).

Parameters **filename** (*str*) – File containing the seed list.

Returns Instance of class.

Return type *SeedList*

classmethod from_info(*phases*, *volume*, *rng_seeds={}*)
Create seed list from microstructure information

This function creates a seed list from information about the microstruture. The "phases" input should be a list of material phase dictionaries, formatted according to the *Phase Dictionaries* guide.

The "volume" input is the minimum volume of the list of seeds. Seeds will be added to the list until this volume threshold is crossed.

Finally, the "rng_seeds" input is a dictionary of random number generator (RNG) seeds for each parameter of the seed geometries. For example, if one of the phases uses "size" to define the seeds, then "size" could be a keyword of the "rng_seeds" input. The value should be a non-negative integer, to seed the RNG for size. The default RNG seed is 0.

Note: If two or more parameters have the same RNG seed and the same kernel of the distribution, those parameters will **not** be correlated. This method updates RNG seeds based on the order that distributions are sampled to avoid correlation between independent random variables.

Parameters

- **phases** (*dict*) – Dictionary of phase information, see *Phase Dictionaries* for a guide.
- **volume** (*float*) – The total area/volume of the seeds in the list.
- **rng_seeds** (*dict*) – *(optional)* Dictionary of RNG seeds for each step in the seeding process. The dictionary keys should match shape parameters in `phases`. For example:

```
rng_seeds = {
    'size': 0,
    'angle': 3,
}
```

Returns An instance of the class containing seeds prescribed by the phase information and filling the given volume.

Return type *SeedList*

plot(*index_by='seed'*, *material=[]*, *loc=0*, ***kwargs*)
Plot the seeds in the seed list.

This function plots the seeds contained in the seed list. In 2D, the seeds are grouped into matplotlib collections to reduce the computational load. In 3D, matplotlib does not have patches, so each seed is rendered as its own surface.

Additional keyword arguments can be specified and passed through to matplotlib. These arguments should be either single values (e.g. `edgecolors='k'`), or lists of values that have the same length as the seed list.

Parameters

- **index_by** (*str*) – *(optional)* {'material' | 'seed'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='material', color=['blue', 'red'])` will plot the seeds with `phase` equal to 0 in blue, and seeds with `phase` equal to 1 in red. Defaults to 'seed'.
- **material** (*list*) – *(optional)* Names of material phases. One entry per material phase (the `index_by` argument is ignored). If this argument is set, a legend is added to the plot with one entry per material.

- **loc** (*int or str*) – *(optional)* The location of the legend, if 'material' is specified. This argument is passed directly through to `matplotlib.pyplot.legend()`. Defaults to 0, which is 'best' in matplotlib.
- ****kwargs** – Keyword arguments to pass to matplotlib

plot_breakdown(*index_by='seed'*, *material=[]*, *loc=0*, ***kwargs*)

Plot the breakdowns of the seeds in seed list.

This function plots the breakdowns of seeds contained in the seed list. In 2D, the breakdowns are grouped into matplotlib collections to reduce the computational load. In 3D, matplotlib does not have patches, so each breakdown is rendered as its own surface.

Additional keyword arguments can be specified and passed through to matplotlib. These arguments should be either single values (e.g. `edgecolors='k'`), or lists of values that have the same length as the seed list.

Parameters

- **index_by** (*str*) – *(optional)* {'material' | 'seed'} Flag for indexing into the other arrays passed into the function. For example, `plot(index_by='material', color=['blue', 'red'])` will plot the seeds with `phase` equal to 0 in blue, and seeds with `phase` equal to 1 in red. Defaults to 'seed'.
- **material** (*list*) – *(optional)* Names of material phases. One entry per material phase (the `index_by` argument is ignored). If this argument is set, a legend is added to the plot with one entry per material.
- **loc** (*int or str*) – *(optional)* The location of the legend, if 'material' is specified. This argument is passed directly through to `matplotlib.pyplot.legend()`. Defaults to 0, which is 'best' in matplotlib.
- ****kwargs** – Keyword arguments to pass to matplotlib

position(*domain*, *pos_dists={}*, *rng_seed=0*, *hold=[]*, *max_attempts=10000*, *rtol='fit'*, *verbose=False*)

Position seeds in a domain

This method positions the seeds within a domain. The "domain" should be a geometry instance from the *microstructpy.geometry* module.

The "pos_dist" input is for phases with custom position distributions, the default being a uniform random distribution. For example:

```
import scipy.stats
mu = [0.5, -0.2]
sigma = [[2.0, 0.3], [0.3, 0.5]]
pos_dists = {2: scipy.stats.multivariate_normal(mu, sigma),
             3: ['random',
                 scipy.stats.norm(0, 1)]
             }
```

Here, phases 0 and 1 have the default distribution, phase 2 has a bivariate normal position distribution, and phase 3 is uniform in the x and normally distributed in the y. Multivariate distributions are described in the multivariate section of the `scipy.stats` documentation.

The position of certain seeds can be held fixed during the positioning process using the "hold" input. This should be a list of booleans, where False indicates a seed should not be held fixed and True indicates that it should be held fixed. The default behavior is to not hold any seeds fixed.

The "rtol" parameter governs the relative overlap tolerable between seeds. Setting rtol to 0 means that there is no overlap, while a value of 1 means that one seed's center is on the edge of another seed. The default

value is 'fit', which determines a tolerance between 0 and 1 based on the ratio of standard deviation to mean in grain volumes.

Parameters

- **domain** (from *microstructpy.geometry*) – The domain of the microstructure.
- **pos_dists** (*dict*) – *(optional)* Position distributions for each phase, formatted like the example above. Defaults to uniform random throughout the domain.
- **rng_seed** (*int*) – *(optional)* Random number generator (RNG) seed for positioning the seeds. Should be a non-negative integer.
- **hold** (*list or numpy.ndarray*) – *(optional)* List of booleans for holding the positions of seeds. Defaults to False for all seeds.
- **max_attempts** (*int*) – *(optional)* Number of random trials before removing a seed from the list. Defaults to 10,000.
- **rtol** (*str or float*) – *(optional)* The relative overlap tolerance between seeds. This parameter should be between 0 and 1. Using the 'fit' option, a function will determine the value for rtol based on the mean and standard deviation in seed volumes.
- **verbose** (*bool*) – *(optional)* This option will print a running counter of how many seeds have been positioned. Defaults to False.

write(*filename*, *format='txt'*)

Write seed list to a text file

This function writes out the seed list to a file. The format of the file can be either 'txt' or 'vtk'. The content of the 'txt' file is human-readable and can be read by the *SeedList.from_file()* method. The 'vtk' option creates a VTK legacy file with the grain geometries.

For grains that are non-spherical, the spherical breakdown of the seed is output instead of the seed itself.

Parameters **filename** (*str*) – File to write the seed list.

7.5 microstructpy.verification

Verification

This module contains functions related to mesh verification.

microstructpy.verification.**error_stats**(*fit_seeds*, *seeds*, *phases*, *poly_mesh=None*, *verif_mask=None*)

Error statistics for seeds

This function creates a dictionary of error statistics for each of the input distributions in the phases.

Parameters

- **fit_seeds** (SeedList) – List of seeds of best fit.
- **seeds** (SeedList) – List of seeds.
- **phases** (*list*) – List of input phase dictionaries.
- **poly_mesh** (PolyMesh) – *(optional)* Polygonal/polyhedral mesh.
- **verif_mask** (*list or numpy.ndarray*) – *(optional)* Mask for seeds to be included in the analysis. Defaults to all True.

Returns List with the same size and dictionary keywords as phases, but with error statistics dictionaries in each entry.

Return type list

microstructpy.verification.**mle_phases**(*seeds*, *phases*, *poly_mesh=None*, *verif_mask=None*)

Get maximum likelihood estimators (MLEs) for phases

This function finds distributions in the list of phases and computes the MLE parameters for those distributions. The returned value is a list of phases with the same length and dictionary keywords, except the distributions are replaced with MLE distributions (based on the seeds). Constant values are replaced with the mean of the seed values.

Note that the directional statistics are not used - so the results for orientation angles and matrices are unreliable.

Also note that SciPy currently does not support MLEs for discrete random variables. Any discrete distributions will be given a histogram output.

Note: Directional statistics are not used and as such the results for orientation angles and matrices are unreliable. The only exception is normally distributed orientation angles.

Parameters

- **seeds** (SeedList) – List of seeds.
- **phases** (*list*) – List of input phase dictionaries.
- **poly_mesh** (PolyMesh) – *(optional)* Polygonal/polyhedral mesh.
- **verif_mask** (*list or numpy.ndarray*) – *(optional)* Mask for which seeds to include in the MLE parameter calculation. Default is True for all seeds.

microstructpy.verification.**plot_distributions**(*seeds*, *phases*, *dirname='.'*, *ext='png'*, *poly_mesh=None*, *verif_mask=None*)

Plot comparison between input and output distributions

This function takes seeds and compares them against the input phases. A polygon mesh can be included for cases where grains are given an area or volume distribution, rather than size/shape/etc.

This function creates both PDF and CDF plots.

Parameters

- **seeds** (SeedList) – List of seeds to compare.
- **phases** (*list*) – List of phase dictionaries.
- **dirname** (*str*) – *(optional)* Plot output directory. Defaults to `.`.
- **ext** (*str or list*) – *(optional)* File extension(s) of the output plots. Defaults to `'png'`.
- **poly_mesh** (PolyMesh) – *(optional)* Polygonal mesh, useful for phases with an area or volume distribution.

Returns none, creates plot files.

microstructpy.verification.**plot_volume_fractions**(*vol_fracs*, *phases*, *filename='volume_fractions.png'*)

Plot volume fraction verification

This function creates a bar chart comparing the input and output volume fractions. If the input volume fraction is distributed, the top of the bar will be a curve representing the CDF of the distribution.

Parameters

- **vol_fracs** (*list or numpy.ndarray*) – Output volume fractions.

- **phases** (*list*) – List of phase dictionaries
- **filename** (*str or list*) – *(optional)* Filename(s) to save the plot. Defaults to `volume_fractions.png`.

Returns none, writes plot to file.

`microstructpy.verification.`**seeds_of_best_fit**(*seeds*, *phases*, *pmesh*, *tmesh*)
Calculate seed geometries of best fit

This function computes the seeds of best fit for the resultant polygonal and triangular meshes. It calls the the `best_fit` function of each seed's geometry, then copies the other seed attributes to create a new *SeedList*.

The points on the faces of the grains are used to determine a fit geometry. Points on the exterior of the domain are not used since they would alter the shape of the best fit seed.

Parameters

- **seeds** (SeedList) – List of seed geometries.
- **phases** (*list*) – List of material phases. See *Phase Dictionaries* for more information on formatting.
- **pmesh** (PolyMesh) – Resultant polygonal/polyhedral mesh.
- **tmesh** (TriMesh) – Resultant triangular/tetrahedral mesh.

Returns List of seeds of best fit.

Return type *SeedList*

`microstructpy.verification.`**volume_fractions**(*poly_mesh*, *n_phases*)
Verify volume fractions

This function computes the volume fractions of each phase in the output mesh. It does so by summing the volumes of the cells in the polygonal mesh.

Parameters

- **poly_mesh** (PolyMesh) – The polygonal/polyhedral mesh.
- **n_phases** (*int*) – Number of phases.

Returns Volume fractions of each phase in the poly mesh.

Return type numpy.ndarray

`microstructpy.verification.`**write_error_stats**(*errs*, *phases*, *filename='error_stats.txt'*)
Write error statistics to file

This function takes previously computed error statistics and writes them to a human-readable text file.

Parameters

- **errs** (*list*) – List of error statistics for each input phase parameter. Organized the same as `phases`.
- **phases** (*list*) – List of input phases. See *Phase Dictionaries* for more details.
- **filename** (*str*) – *(optional)* The name of the file to contain the error statistics. Defaults to `error_stats.txt`.

`microstructpy.verification.`**write_mle_phases**(*inp_phases*, *out_phases*, *filename='mles.txt'*)
Write MLE parameters in a table

This function writes out a text file containing the input parameters and maximum likelihood estimators (MLEs) for the outputs.

Parameters

- **inp_phases** (*list*) – List of input phase dictionaries.
- **out_phases** (*list*) – List of output phase dictionaries.
- **filename** (*str*) – *(optional)* Filename of the output table. Defaults to `mles.txt`.

Returns none, writes file.

`microstructpy.verification.`**write_volume_fractions**(*vol_fracs*, *phases*, *filename='volume_fractions.txt'*)

Write volume fractions to a file

Write the volume fractions verification out to a file. The output columns are:

1. Phase number
2. Phase name
3. Input relative volume (average, if distributed)
4. Output relative volume
5. Input volume fraction (average, if distributed)
6. Output volume fraction

The first three lines of the output file are headings.

Parameters

- **vol_fracs** (*list or numpy.ndarray*) – Volume fractions of the output mesh.
- **phases** (*list*) – List of phase dictionaries.
- **filename** (*str*) – *(optional)* Name of file to write. Defaults to `volume_fractions.txt`.

Returns none, prints formatted volume fraction verification table to file

CHAPTER

EIGHT

TROUBLESHOOTING

This page addresses some problems that may be encountered with MicroStructPy. If this page does not address your problem, please submit an issue through the package GitHub page.

8.1 Installation

These are problems encountered when installing MicroStructPy.

8.1.1 Missing library for pygmsh on Linux

Problem Description

When running MicroStructPy for the first time on a Linux operating system, there is an error message like:

```
...
src/microstructpy/meshing/trimesh.py:19: in <module>
    import pygmsh as pg
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/pygmsh/__init__.py:1:␣
↪in <module>
    from . import geo, occ
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/pygmsh/geo/__init__.
↪py:1: in <module>
    from .geometry import Geometry
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/pygmsh/geo/geometry.
↪py:1: in <module>
    import gmsh
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/gmsh-4.6.0-Linux64-sdk/
↪lib/gmsh.py:39: in <module>
    lib = CDLL(libpath)
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/ctypes/__init__.py:364: in __init__
    self._handle = _dlopen(self._name, mode)
E   OSError: libGLU.so.1: cannot open shared object file: No such file or directory
```

Problem Solution

The libGLU library is misssing from the computer. To add it, run:

```
sudo apt-get install libglu1
```

8.1.2 MeshPy fails to install

Problem Description

When installing the package, either through PyPI as `pip install microstructpy` or from the source as `pip install -e .` in the top-level directory, an error message appears during the meshpy install. The error message indicates that Visual Studio cannot find the pybind11 headers.

Problem Solution

Install pybind11 first by running `pip install pybind11`, then try to install MicroStructPy.

8.2 Command Line Interface

These are problems encountered when running `microstructpy input_file.xml`.

8.2.1 Command not found on Linux

Problem Description

The MicroStructPy package installs without a problem, however on running `microstructpy example_file.xml` the following message appears:

```
microstructpy: command not found
```

Problem Solution

The command line interface (CLI) is install to a directory that is not in the PATH variable. Check for the CLI in `~/.local/bin` and if it is there, add the following to your `~/.bash_profile` file:

```
export PATH=$PATH:~/.local/bin
```

then source the .bash_profile file by running `source ~/.bash_profile`.

8.2.2 'tkinter' not found on Linux

Problem Description

The MicroStructPy package installs without a problem, however on running `microstructpy example_file.xml` the following error is raised:

```
ModuleNotFoundError: No module named 'tkinter'
```

Problem Solution

To install `tkinter` for Python 3 on Linux, run the following command:

```
sudo apt-get install python3-tk
```

For Python 2, run the following instead:

```
sudo apt-get install python-tk
```

8.2.3 Program quits/segfaults while calculating Voronoi diagram

Problem Description

During the calculating Voronoi diagram step, the program either quits or segfaults.

Problem Solution

This issue was experienced while running 32-bit Python with a large number of seeds. Python ran out of memory addresses and segfaulted. Switching from 32-bit to 64-bit Python solved the problem.

PYTHON MODULE INDEX

m

microstructpy.cli, 119
microstructpy.geometry, 123
microstructpy.meshing, 149
microstructpy.seeding, 156
microstructpy.verification, 161

INDEX

A

angle (*microstructpy.geometry.Rectangle property*), 142
angle (*microstructpy.geometry.Square property*), 147
angle_deg (*microstructpy.geometry.Ellipse property*), 132
angle_deg (*microstructpy.geometry.Rectangle property*), 142
angle_deg (*microstructpy.geometry.Square property*), 147
angle_rad (*microstructpy.geometry.Ellipse property*), 132
angle_rad (*microstructpy.geometry.Rectangle property*), 142
angle_rad (*microstructpy.geometry.Square property*), 147
append() (*microstructpy.seeding.SeedList method*), 158
approximate() (*microstructpy.geometry.Circle method*), 125
approximate() (*microstructpy.geometry.Ellipse method*), 129
approximate() (*microstructpy.geometry.Ellipsoid method*), 134
approximate() (*microstructpy.geometry.n_sphere.NSphere method*), 138
approximate() (*microstructpy.geometry.Rectangle method*), 140
approximate() (*microstructpy.geometry.Sphere method*), 143
approximate() (*microstructpy.geometry.Square method*), 145
area (*microstructpy.geometry.Circle property*), 127
area (*microstructpy.geometry.Ellipse property*), 132
area (*microstructpy.geometry.Rectangle property*), 142
area (*microstructpy.geometry.Square property*), 147
area_expectation() (*microstructpy.geometry.Circle class method*), 126
area_expectation() (*microstructpy.geometry.Ellipse class method*), 131
area_expectation() (*microstructpy.geometry.Rectangle class method*), 140
area_expectation() (*microstructpy.geometry.Square class method*), 146
as_array() (*microstructpy.meshing.RasterMesh method*), 152
aspect_ratio (*microstructpy.geometry.Ellipse property*), 132
axes (*microstructpy.geometry.Ellipse property*), 132
axes (*microstructpy.geometry.Ellipsoid property*), 135

B

best_fit() (*microstructpy.geometry.Circle class method*), 126
best_fit() (*microstructpy.geometry.Ellipse method*), 131
best_fit() (*microstructpy.geometry.Ellipsoid method*), 134
best_fit() (*microstructpy.geometry.n_sphere.NSphere class method*), 138
best_fit() (*microstructpy.geometry.Rectangle method*), 141
best_fit() (*microstructpy.geometry.Sphere class method*), 143
best_fit() (*microstructpy.geometry.Square method*), 147
bound_max (*microstructpy.geometry.Circle property*), 127
bound_max (*microstructpy.geometry.Ellipse property*), 132
bound_max (*microstructpy.geometry.Ellipsoid property*), 135
bound_max (*microstructpy.geometry.n_sphere.NSphere property*), 139
bound_max (*microstructpy.geometry.Sphere property*), 144
bound_min (*microstructpy.geometry.Circle property*), 127
bound_min (*microstructpy.geometry.Ellipse property*), 132
bound_min (*microstructpy.geometry.Ellipsoid property*), 135
bound_min (*microstructpy.geometry.n_sphere.NSphere property*), 139
bound_min (*microstructpy.geometry.Sphere property*),

144
bounds (*microstructpy.geometry.Box property*), 124
bounds (*microstructpy.geometry.Cube property*), 128
bounds (*microstructpy.geometry.n_box.NBox property*), 137
bounds (*microstructpy.geometry.Rectangle property*), 142
bounds (*microstructpy.geometry.Square property*), 148
Box (*class in microstructpy.geometry*), 124

C

Circle (*class in microstructpy.geometry*), 125
coefficients (*microstructpy.geometry.Ellipsoid property*), 136
corner (*microstructpy.geometry.Box property*), 124
corner (*microstructpy.geometry.Cube property*), 128
corner (*microstructpy.geometry.n_box.NBox property*), 137
corner (*microstructpy.geometry.Rectangle property*), 142
corner (*microstructpy.geometry.Square property*), 148
Cube (*class in microstructpy.geometry*), 128

D

d (*microstructpy.geometry.Circle property*), 127
d (*microstructpy.geometry.n_sphere.NSphere property*), 139
d (*microstructpy.geometry.Sphere property*), 145
diameter (*microstructpy.geometry.Circle property*), 127
diameter (*microstructpy.geometry.n_sphere.NSphere property*), 139
diameter (*microstructpy.geometry.Sphere property*), 145
dict_convert() (*in module microstructpy.cli*), 119

E

Ellipse (*class in microstructpy.geometry*), 129
Ellipsoid (*class in microstructpy.geometry*), 133
error_stats() (*in module microstructpy.verification*), 161
extend() (*microstructpy.seeding.SeedList method*), 158

F

factory() (*microstructpy.geometry method*), 148
factory() (*microstructpy.seeding.Seed class method*), 156
from_file() (*microstructpy.meshing.PolyMesh class method*), 149
from_file() (*microstructpy.meshing.RasterMesh class method*), 152
from_file() (*microstructpy.meshing.TriMesh class method*), 154
from_file() (*microstructpy.seeding.SeedList class method*), 158
from_info() (*microstructpy.seeding.SeedList class method*), 159
from_polymesh() (*microstructpy.meshing.RasterMesh class method*), 152
from_polymesh() (*microstructpy.meshing.TriMesh class method*), 154
from_seeds() (*microstructpy.meshing.PolyMesh class method*), 150
from_str() (*microstructpy.seeding.Seed class method*), 157

I

input2dict() (*in module microstructpy.cli*), 119

L

length (*microstructpy.geometry.Rectangle property*), 142
length (*microstructpy.geometry.Square property*), 148
limits (*microstructpy.geometry.Box property*), 125
limits (*microstructpy.geometry.Circle property*), 127
limits (*microstructpy.geometry.Cube property*), 128
limits (*microstructpy.geometry.Ellipse property*), 132
limits (*microstructpy.geometry.Ellipsoid property*), 136
limits (*microstructpy.geometry.n_box.NBox property*), 138
limits (*microstructpy.geometry.n_sphere.NSphere property*), 139
limits (*microstructpy.geometry.Rectangle property*), 142
limits (*microstructpy.geometry.Sphere property*), 145
limits (*microstructpy.geometry.Square property*), 148
limits (*microstructpy.seeding.Seed property*), 158

M

main() (*in module microstructpy.cli*), 120
matrix (*microstructpy.geometry.Ellipse property*), 132
matrix (*microstructpy.geometry.Ellipsoid property*), 136
matrix_quadeq (*microstructpy.geometry.Ellipsoid property*), 136
matrix_quadform (*microstructpy.geometry.Ellipsoid property*), 136
mesh_size (*microstructpy.meshing.RasterMesh property*), 153
microstructpy.cli
    module, 119
microstructpy.geometry
    module, 123
microstructpy.meshing
    module, 149
microstructpy.seeding
    module, 156
microstructpy.verification
    module, 161

mle_phases() (*in module microstructpy.verification*), 162
module
    microstructpy.cli, 119
    microstructpy.geometry, 123
    microstructpy.meshing, 149
    microstructpy.seeding, 156
    microstructpy.verification, 161

N

n_dim (*microstructpy.geometry.Box property*), 125
n_dim (*microstructpy.geometry.Circle property*), 127
n_dim (*microstructpy.geometry.Cube property*), 128
n_dim (*microstructpy.geometry.Ellipse property*), 133
n_dim (*microstructpy.geometry.Ellipsoid property*), 136
n_dim (*microstructpy.geometry.Rectangle property*), 142
n_dim (*microstructpy.geometry.Sphere property*), 145
n_dim (*microstructpy.geometry.Square property*), 148
n_vol (*microstructpy.geometry.Box property*), 125
n_vol (*microstructpy.geometry.Cube property*), 128
n_vol (*microstructpy.geometry.n_box.NBox property*), 138
n_vol (*microstructpy.geometry.Rectangle property*), 143
n_vol (*microstructpy.geometry.Square property*), 148
NBox (*class in microstructpy.geometry.n_box*), 137
NSphere (*class in microstructpy.geometry.n_sphere*), 138

O

orientation (*microstructpy.geometry.Ellipse property*), 133
orientation (*microstructpy.geometry.Ellipsoid property*), 136

P

plot() (*microstructpy.geometry.Box method*), 124
plot() (*microstructpy.geometry.Circle method*), 126
plot() (*microstructpy.geometry.Cube method*), 128
plot() (*microstructpy.geometry.Ellipse method*), 131
plot() (*microstructpy.geometry.Ellipsoid method*), 134
plot() (*microstructpy.geometry.Rectangle method*), 142
plot() (*microstructpy.geometry.Sphere method*), 144
plot() (*microstructpy.geometry.Square method*), 147
plot() (*microstructpy.meshing.PolyMesh method*), 150
plot() (*microstructpy.meshing.RasterMesh method*), 153
plot() (*microstructpy.meshing.TriMesh method*), 155
plot() (*microstructpy.seeding.Seed method*), 157
plot() (*microstructpy.seeding.SeedList method*), 159
plot_breakdown() (*microstructpy.seeding.Seed method*), 157
plot_breakdown() (*microstructpy.seeding.SeedList method*), 160
plot_distributions() (*in module microstructpy.verification*), 162
plot_facets() (*microstructpy.meshing.PolyMesh method*), 151
plot_poly() (*in module microstructpy.cli*), 120
plot_seeds() (*in module microstructpy.cli*), 120
plot_tri() (*in module microstructpy.cli*), 120
plot_volume_fractions() (*in module microstructpy.verification*), 162
PolyMesh (*class in microstructpy.meshing*), 149
position (*microstructpy.geometry.Circle property*), 127
position (*microstructpy.geometry.n_sphere.NSphere property*), 139
position (*microstructpy.geometry.Sphere property*), 145
position (*microstructpy.seeding.Seed property*), 158
position() (*microstructpy.seeding.SeedList method*), 160

R

radius (*microstructpy.geometry.Circle property*), 127
radius (*microstructpy.geometry.n_sphere.NSphere property*), 139
radius (*microstructpy.geometry.Sphere property*), 145
RasterMesh (*class in microstructpy.meshing*), 151
ratio_ab (*microstructpy.geometry.Ellipsoid property*), 136
ratio_ac (*microstructpy.geometry.Ellipsoid property*), 136
ratio_ba (*microstructpy.geometry.Ellipsoid property*), 136
ratio_bc (*microstructpy.geometry.Ellipsoid property*), 136
ratio_ca (*microstructpy.geometry.Ellipsoid property*), 136
ratio_cb (*microstructpy.geometry.Ellipsoid property*), 136
read_input() (*in module microstructpy.cli*), 121
Rectangle (*class in microstructpy.geometry*), 140
reflect() (*microstructpy.geometry.Circle method*), 126
reflect() (*microstructpy.geometry.Ellipse method*), 132
reflect() (*microstructpy.geometry.Ellipsoid method*), 134
reflect() (*microstructpy.geometry.n_sphere.NSphere method*), 139
reflect() (*microstructpy.geometry.Sphere method*), 144
rot_seq_deg (*microstructpy.geometry.Ellipsoid property*), 136
rot_seq_rad (*microstructpy.geometry.Ellipsoid property*), 137
run() (*in module microstructpy.cli*), 121
run_file() (*in module microstructpy.cli*), 123

S

sample_limits (*microstructpy.geometry.Box property*), 125
sample_limits (*microstructpy.geometry.Circle property*), 127
sample_limits (*microstructpy.geometry.Cube property*), 129
sample_limits (*microstructpy.geometry.Ellipse property*), 133
sample_limits (*microstructpy.geometry.Ellipsoid property*), 137
sample_limits (*microstructpy.geometry.n_box.NBox property*), 138
sample_limits (*microstructpy.geometry.n_sphere.NSphere property*), 139
sample_limits (*microstructpy.geometry.Rectangle property*), 143
sample_limits (*microstructpy.geometry.Sphere property*), 145
sample_limits (*microstructpy.geometry.Square property*), 148
Seed (*class in microstructpy.seeding*), 156
SeedList (*class in microstructpy.seeding*), 158
seeds_of_best_fit() (*in module microstructpy.verification*), 163
side_length (*microstructpy.geometry.Cube property*), 129
side_length (*microstructpy.geometry.Square property*), 148
size (*microstructpy.geometry.Circle property*), 127
size (*microstructpy.geometry.Ellipse property*), 133
size (*microstructpy.geometry.Ellipsoid property*), 137
size (*microstructpy.geometry.n_sphere.NSphere property*), 140
size (*microstructpy.geometry.Sphere property*), 145
Sphere (*class in microstructpy.geometry*), 143
Square (*class in microstructpy.geometry*), 145

T

TriMesh (*class in microstructpy.meshing*), 154

V

volume (*microstructpy.geometry.Box property*), 125
volume (*microstructpy.geometry.Circle property*), 127
volume (*microstructpy.geometry.Cube property*), 129
volume (*microstructpy.geometry.Ellipse property*), 133
volume (*microstructpy.geometry.Ellipsoid property*), 137
volume (*microstructpy.geometry.Sphere property*), 145
volume (*microstructpy.seeding.Seed property*), 158
volume_expectation() (*microstructpy.geometry.Ellipsoid class method*), 135
volume_expectation() (*microstructpy.geometry.Sphere class method*), 144
volume_fractions() (*in module microstructpy.verification*), 163

W

width (*microstructpy.geometry.Rectangle property*), 143
width (*microstructpy.geometry.Square property*), 148
within() (*microstructpy.geometry.Box method*), 124
within() (*microstructpy.geometry.Circle method*), 126
within() (*microstructpy.geometry.Cube method*), 128
within() (*microstructpy.geometry.Ellipse method*), 132
within() (*microstructpy.geometry.Ellipsoid method*), 135
within() (*microstructpy.geometry.n_box.NBox method*), 137
within() (*microstructpy.geometry.n_sphere.NSphere method*), 139
within() (*microstructpy.geometry.Rectangle method*), 142
within() (*microstructpy.geometry.Sphere method*), 144
within() (*microstructpy.geometry.Square method*), 147
write() (*microstructpy.meshing.PolyMesh method*), 151
write() (*microstructpy.meshing.RasterMesh method*), 153
write() (*microstructpy.meshing.TriMesh method*), 155
write() (*microstructpy.seeding.SeedList method*), 161
write_error_stats() (*in module microstructpy.verification*), 163
write_mle_phases() (*in module microstructpy.verification*), 163
write_volume_fractions() (*in module microstructpy.verification*), 164