MicroStructPy Documentation

Release 1.0

Kenneth Hart

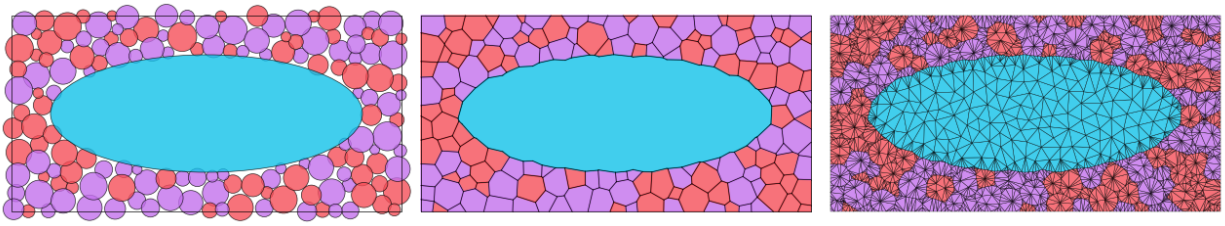
Oct 16, 2019

Contents

Repository Documentation PyPI

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



CHAPTER 1

Quick Start

To install MicroStructPy, download it from PyPI using:

```
pip install microstructpy
```

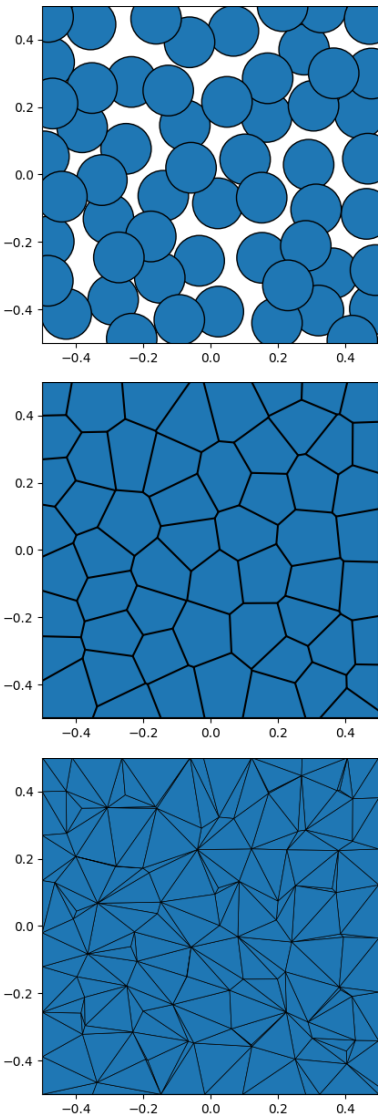
If there is an error with the install, try `pip install pybind11` first, then install MicroStructPy. This will create a command line executable and python package both named `microstructpy`. To use the command line interface, create a file called `input.xml` and copy this into it:

```
<?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:



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()

# 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
```

(continues on next page)

(continued from previous page)

```
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()
```

CHAPTER 2

License and Attribution

 MicroStructPy is a major part of Kenneth (Kip) Hart's doctoral thesis, advised by Prof. Julian Rimoli.

CHAPTER 3

Contents

3.1 Getting Started

This page covers topics for new users to get started using MicroStructPy.

3.1.1 Download & Installation

To install MicroStructPy, download it from PyPI using:

```
pip install microstructpy
```

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. This installs both the `microstructpy` Python package and the `microstructpy` command line interface (CLI).

To verify installation of the package, run `python` and enter the command:

```
>>> import microstructpy
```

If the import succeeds without an error message, then MicroStructPy installed successfully.

To verify that the command line interface has been installed, run `microstructpy --help` to see if the man page is printed. If it is not, then the install location may not be in your 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\`.

From Source

To install from source, navigate to the GitHub repository to clone or download the latest release. Unzip the source if necessary and change directories to the top-level folder. From this folder, run the command:

```
pip install -e .
```

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

3.1.2 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=docs_banner.xml
```

When a demo is run, the XML input file is copied to the current working directory. To see a full list of available demos, visit the *Examples* page.

3.1.3 Using the Python Package

The MicrostructPy package contains several classes and users can customize their workflows depending on their needs. For example:

```
import os

import matplotlib.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
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)
```

The commands above create a Voronoi diagram with 50 random seed points:

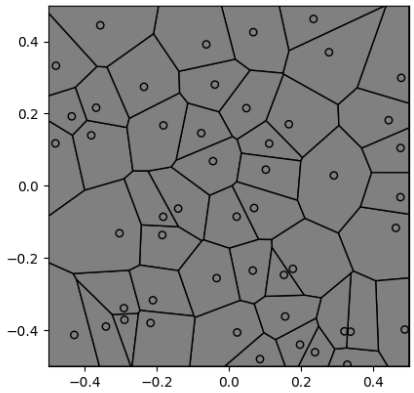


Fig. 1: Plot resulting from the Python script above.

More information about this example is available on the *Standard Voronoi Diagram* page. The *Python Package Guide* page has more details on writing scripts with the package and the standard data flow.

3.2 Examples

This page contains all of the examples for MicroStructPy. Below each image is a link to a page that describes each example in greater detail.

3.2.1 Input File Introduction

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>
```

(continues on next page)

(continued from previous page)

```
        </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>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

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

Material 1 - Matrix

```
<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>
```

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

Material 2 - Inclusions

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

The second material consists of circular inclusions with diameter 2.

Domain Geometry

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

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

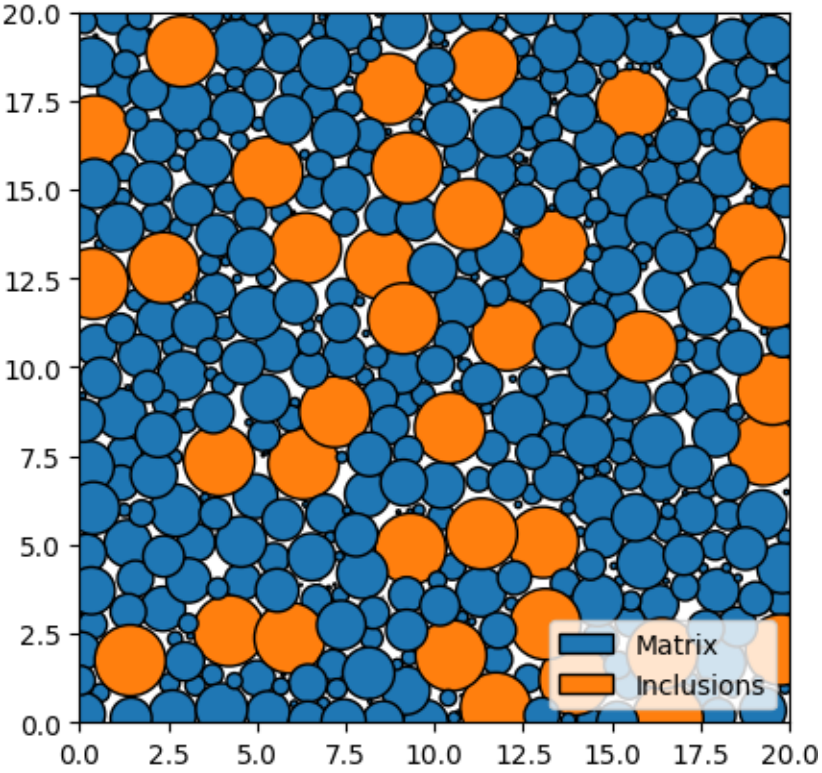
```
<settings>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
    </filetypes>

    <directory> intro_1_basic </directory>
    <verbose> True </verbose>
</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_1_basic`, in the current directory (i.e `./intro_1_basic`).

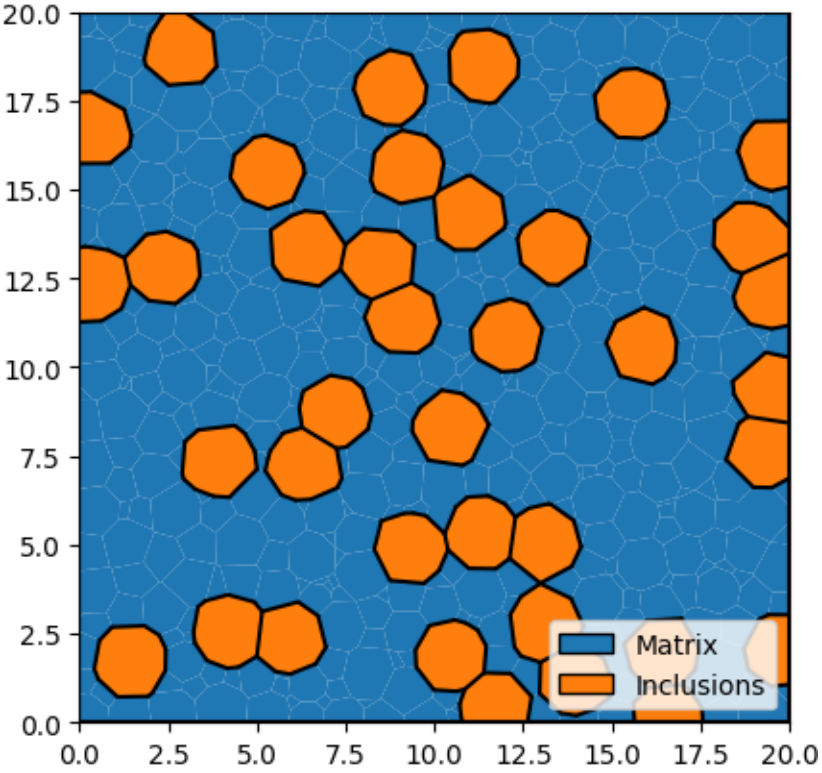
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

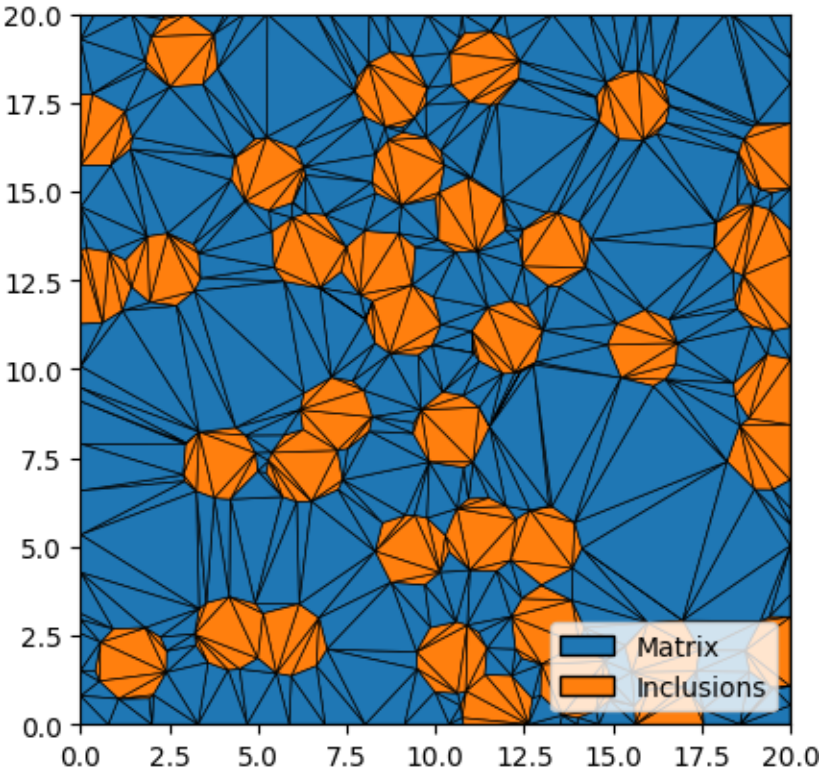
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

        <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>
```

Material 1 - Matrix

```
<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>
```

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

Material 2 - Inclusions

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

(continues on next page)

(continued from previous page)

```
    <diameter> 2 </diameter>
</material>
```

The second material consists of circular inclusions with diameter 2.

Domain Geometry

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

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

```
<settings>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
    </filetypes>

    <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>
```

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_2_quality`, in the current directory (i.e `./intro_2_quality`).

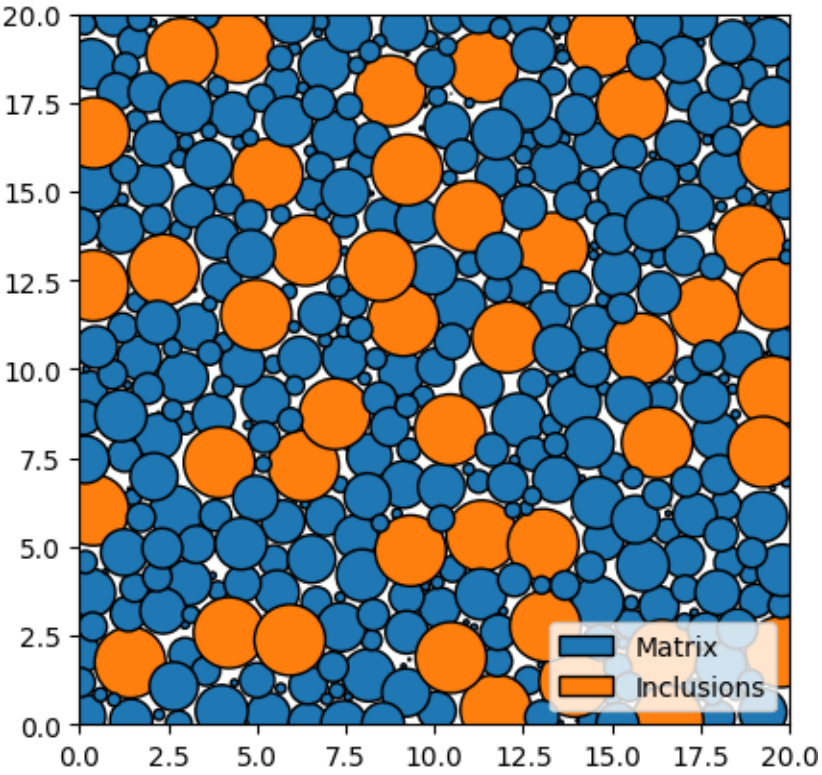
The minimum interior angle of the elements is 25 degrees, ensuring lower aspect ratios compared to the first example. The maximum area of the elements is also limited to 1, which populates the matrix with smaller elements. Finally, The maximum edge length of elements at interfaces is set to 0.1, which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

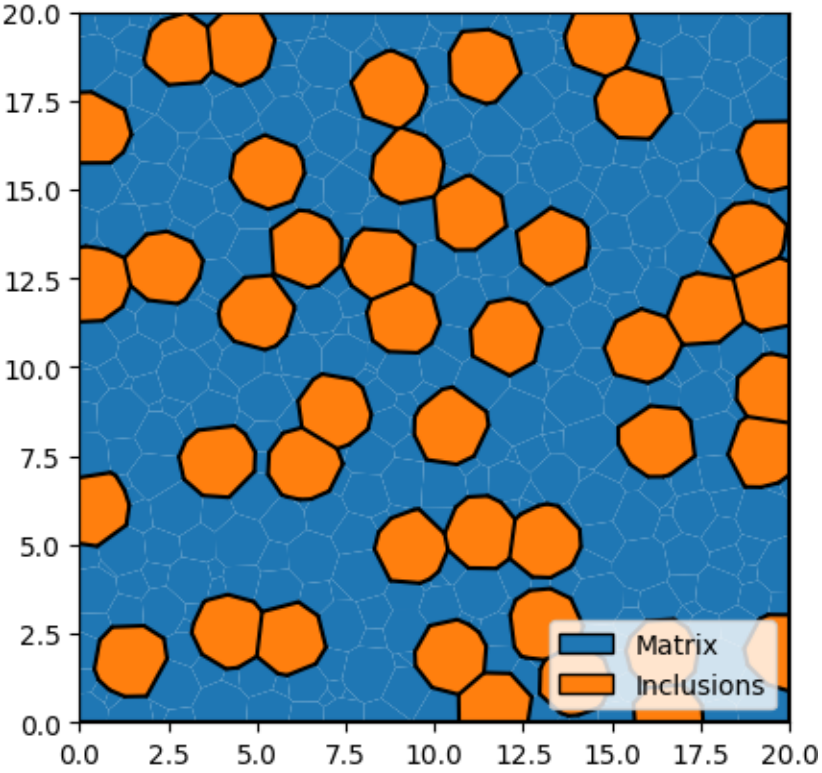
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

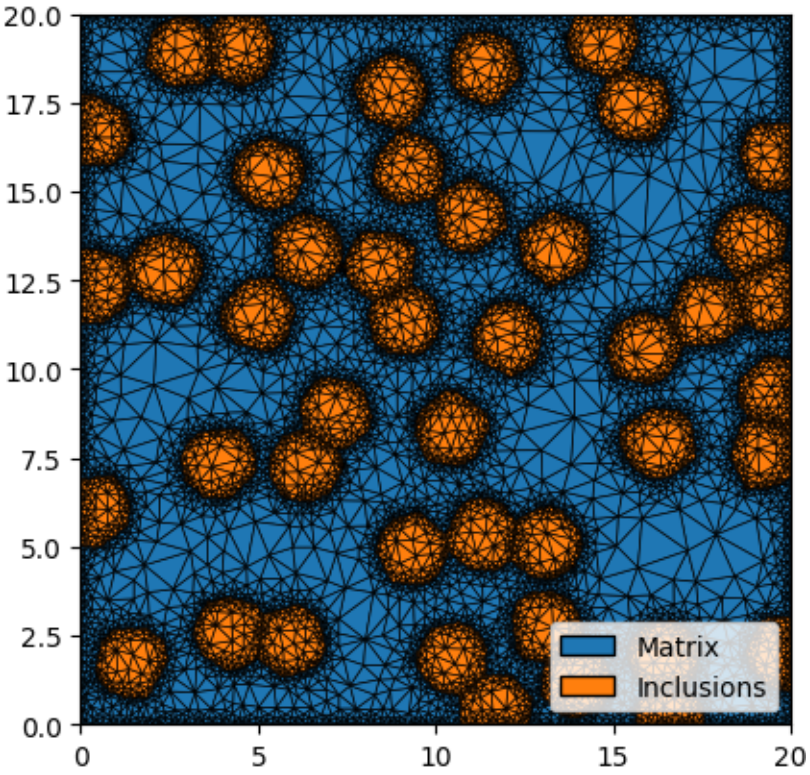
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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>
            <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> 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>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

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

Material 1 - Matrix

```
<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>
```

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

Material 2 - Inclusions

```
<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>
```

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

```
        <scale> 2 </scale>
        <c> 1 </c>
    </size>
    <aspect_ratio>
        <dist_type> uniform </dist_type>
```

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

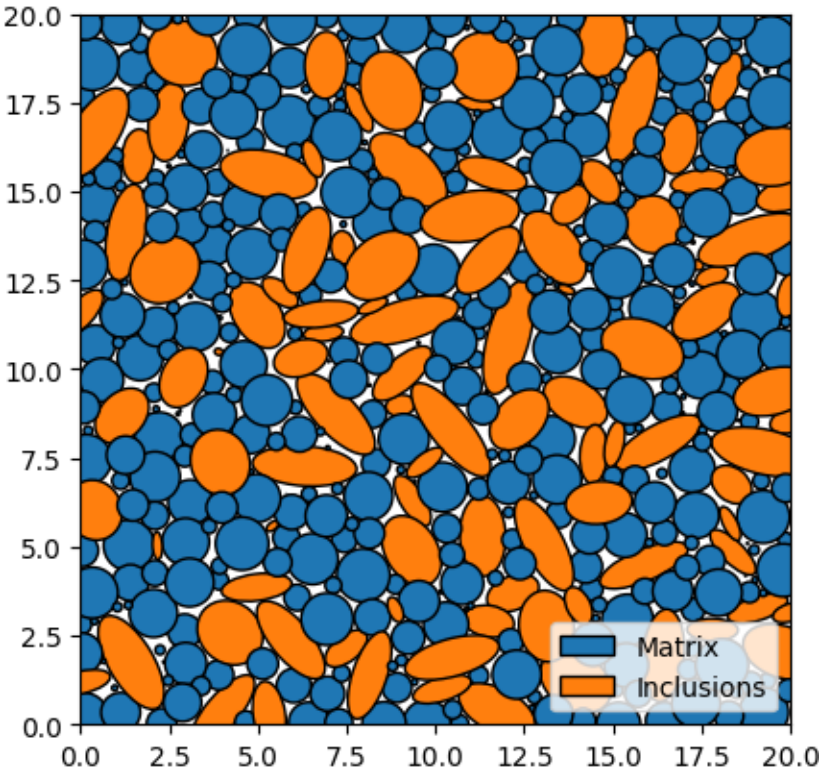
```
        <scale> 2 </scale>
    </aspect_ratio>
    <angle> random </angle>
</material>

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

<settings>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
```

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_3_size_shape`, in the current directory (i.e `./intro_3_size_shape`).

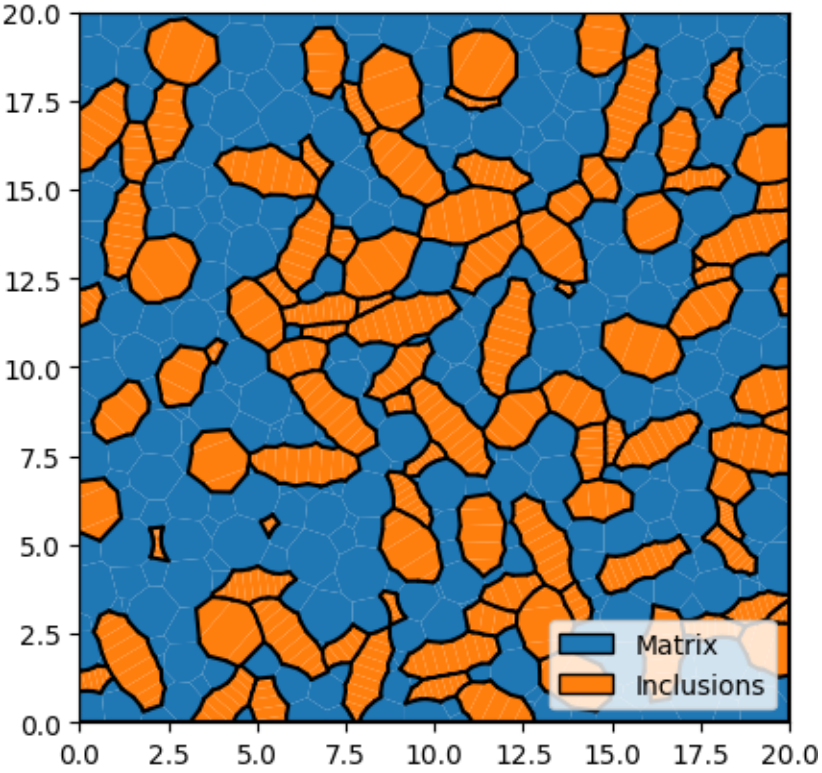
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

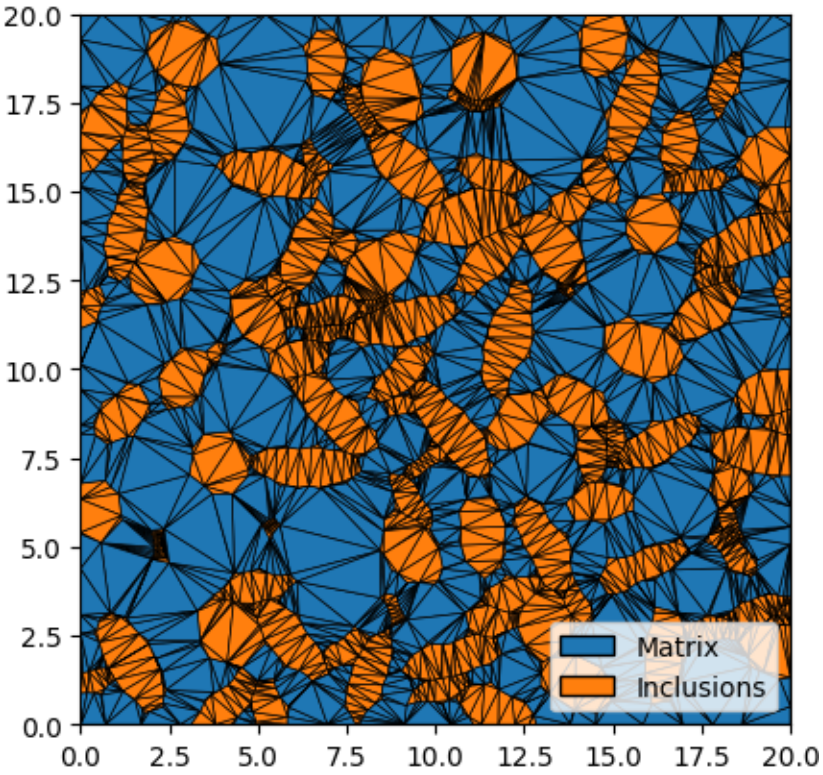
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



Oriented Grains and Amorphous Phases

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>
```

(continues on next page)

(continued from previous page)

```
        <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>

    <settings>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

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

Material 1 - Matrix

```
<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>
```

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

Material 2 - Inclusions

```
<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>
```

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

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

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

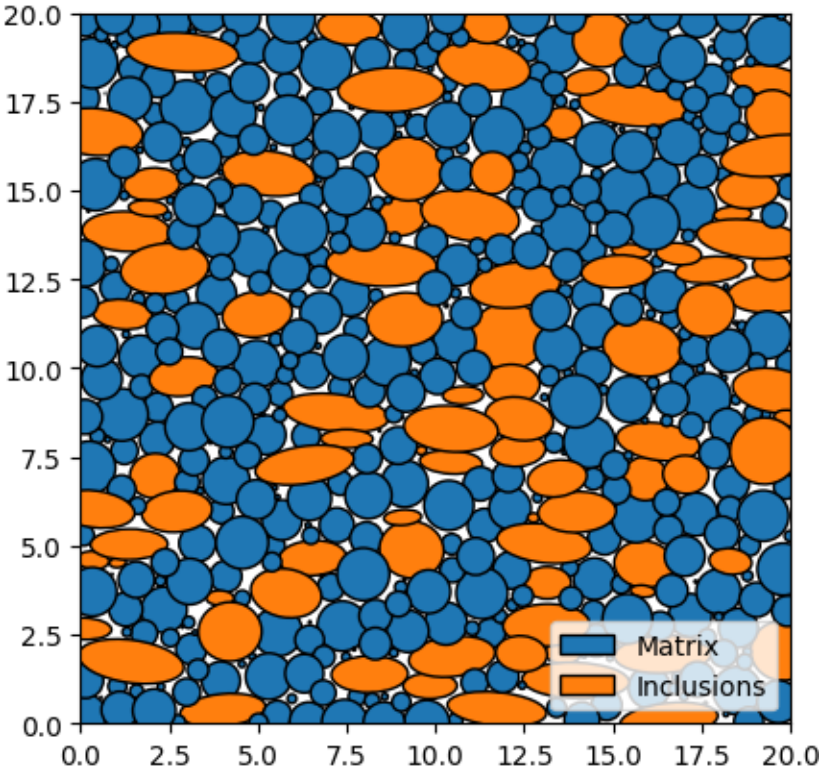
```
<settings>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
    </filetypes>

    <directory> intro_4_oriented </directory>
    <verbose> True </verbose>
</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_4_oriented`, in the current directory (i.e `./intro_4_oriented`).

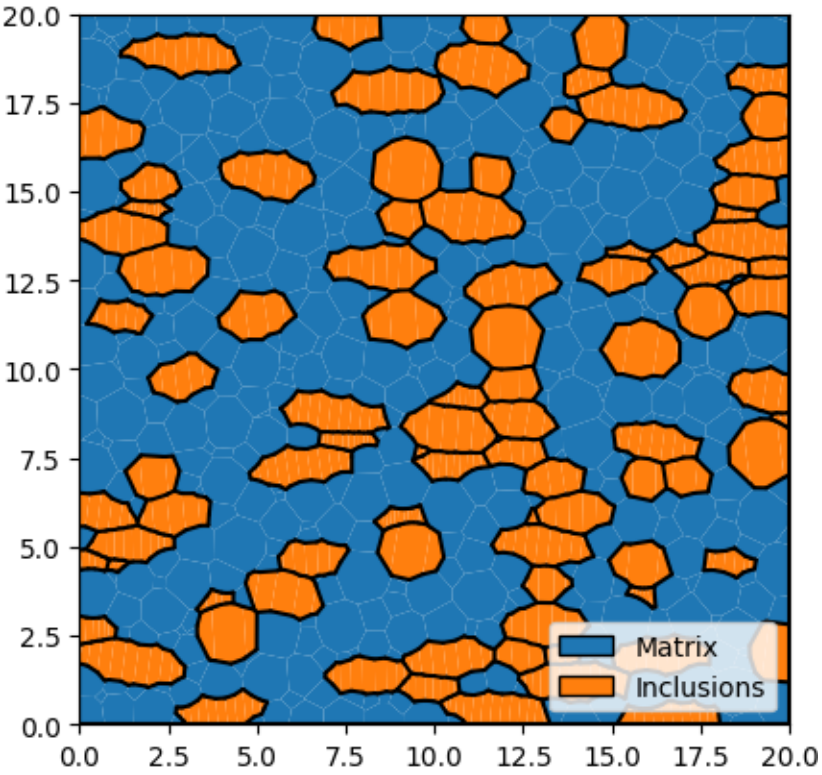
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

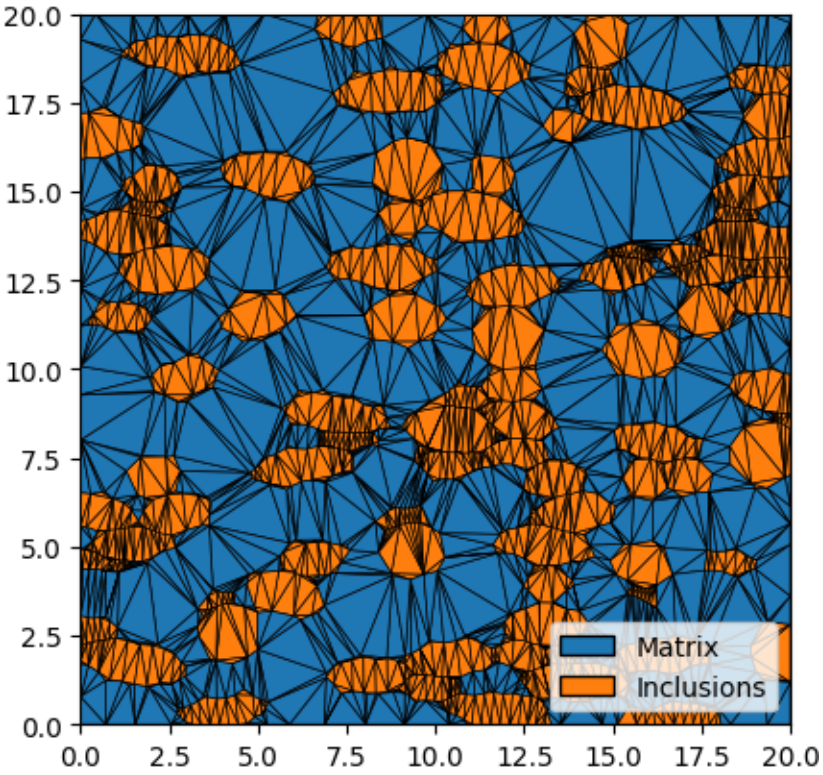
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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.

Listing 1: Plot controls example input file

```
<?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>
```

(continues on next page)

(continued from previous page)

```
    <material>
        <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>
```

Material 1 - Matrix

Listing 2: Matrix material section of input file

```
<material>
    <material_type> matrix </material_type>
    <fraction> 2 </fraction>

    <shape> circle </shape>
```

(continues on next page)

(continued from previous page)

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

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

Material 2 - Inclusions

Listing 3: Inclusions material section of input file

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

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

Domain Geometry

Listing 4: Domain geometry section of input file

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

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

Listing 5: Settings section of input file

```
<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>
```

(continues on next page)

(continued from previous page)

```
    <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>
```

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.

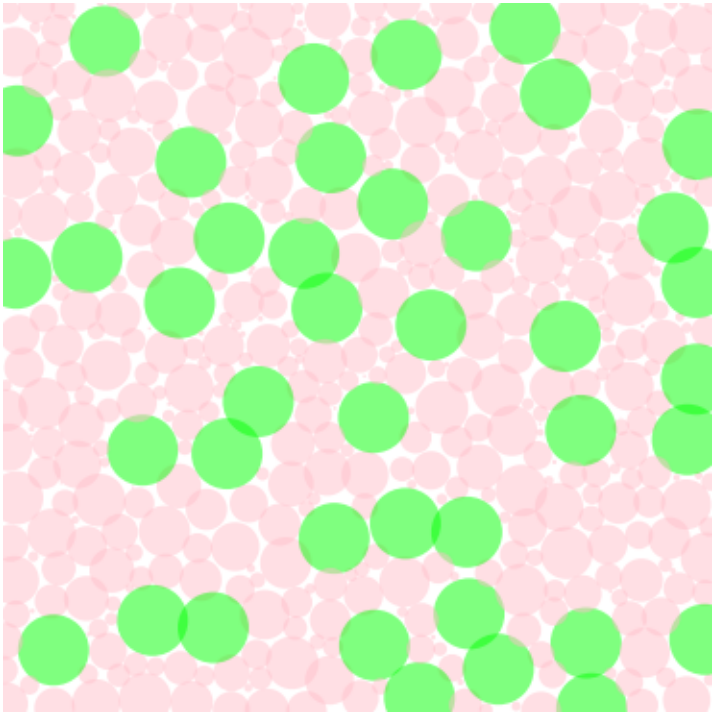
The seeds are plotted with transparency to show the overlap between them. The poly mesh is plotted with thick purple edges and the tri mesh is plotted with thin navy edges.

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

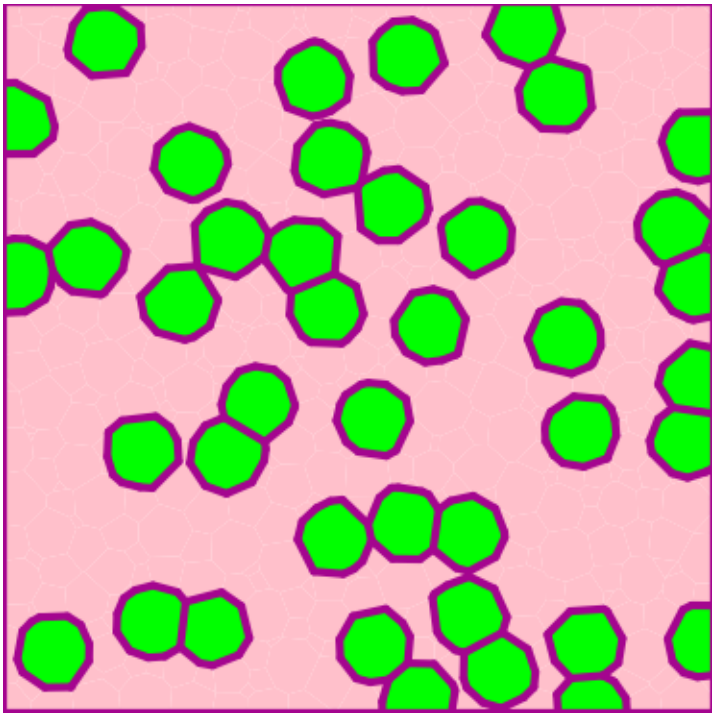
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

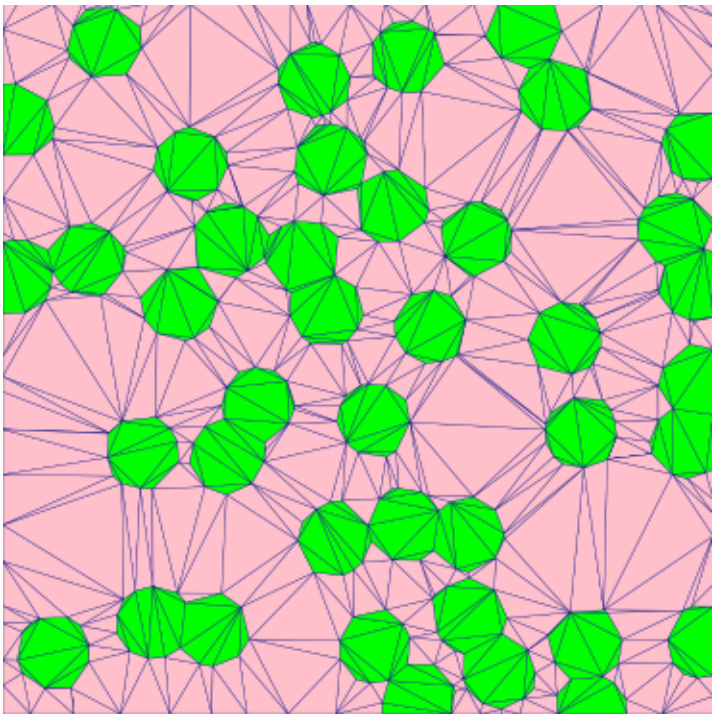
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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>
```

(continues on next page)

(continued from previous page)

```
        <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>

    <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>
```

(continues on next page)

(continued from previous page)

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

Material 1 - Matrix

```
<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>
```

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

Material 2 - Inclusions

```
<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>
```

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

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

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

```
<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>
```

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.

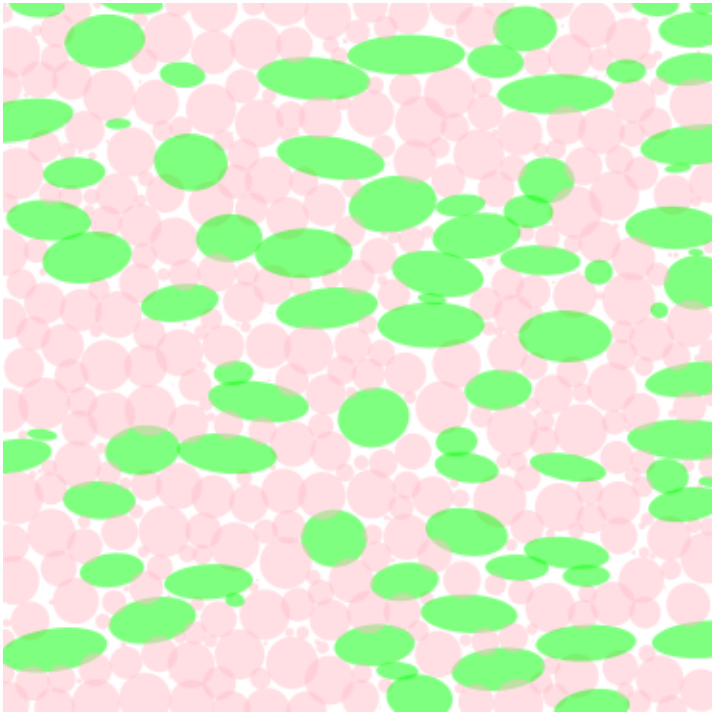
The minimum interior angle of the elements is 25 degrees, ensuring lower aspect ratios compared to the first example. The maximum area of the elements is also limited to 1, which populates the matrix with smaller elements. Finally, The maximum edge length of elements at interfaces is set to 0.1, which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

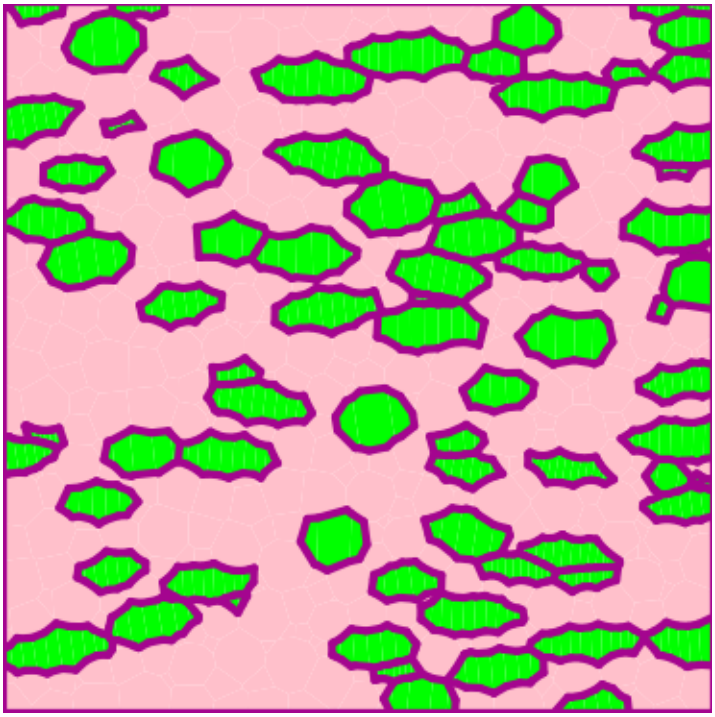
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

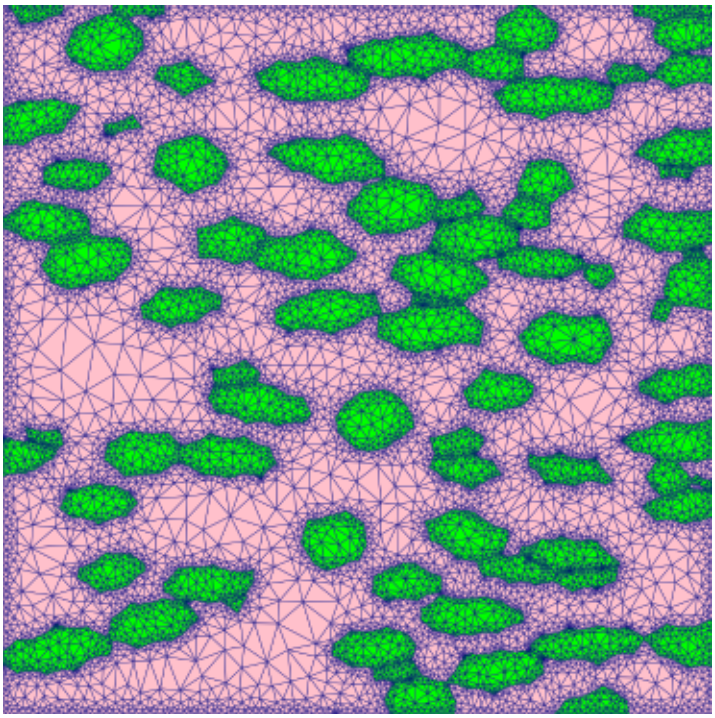
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



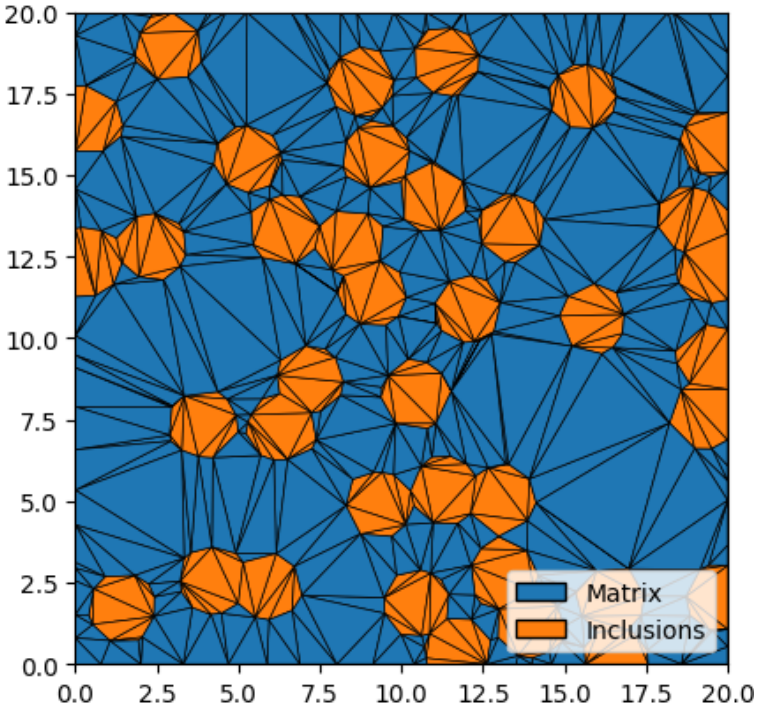


Fig. 2: *Basic Example*

3.2.2 Examples Using the CLI

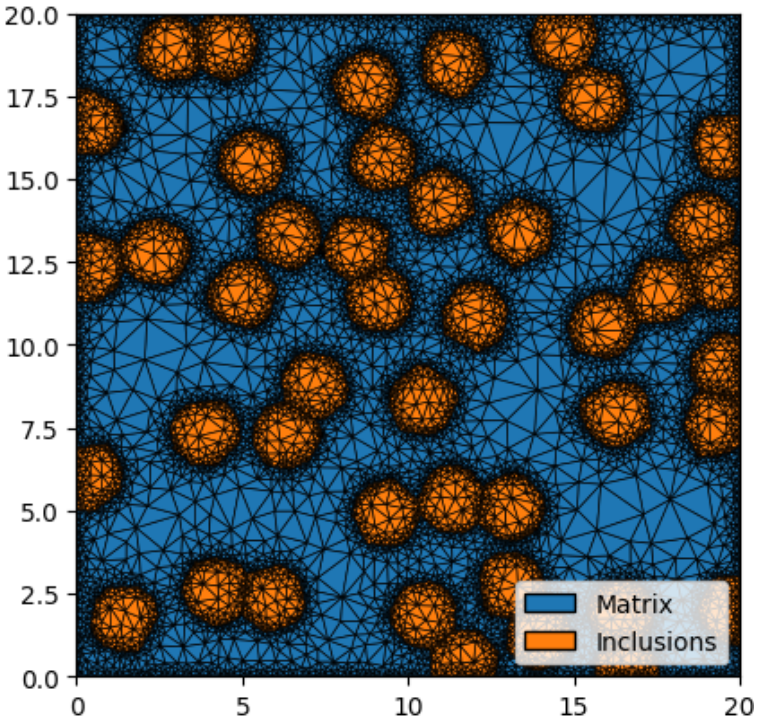


Fig. 3: *Quality Controls*

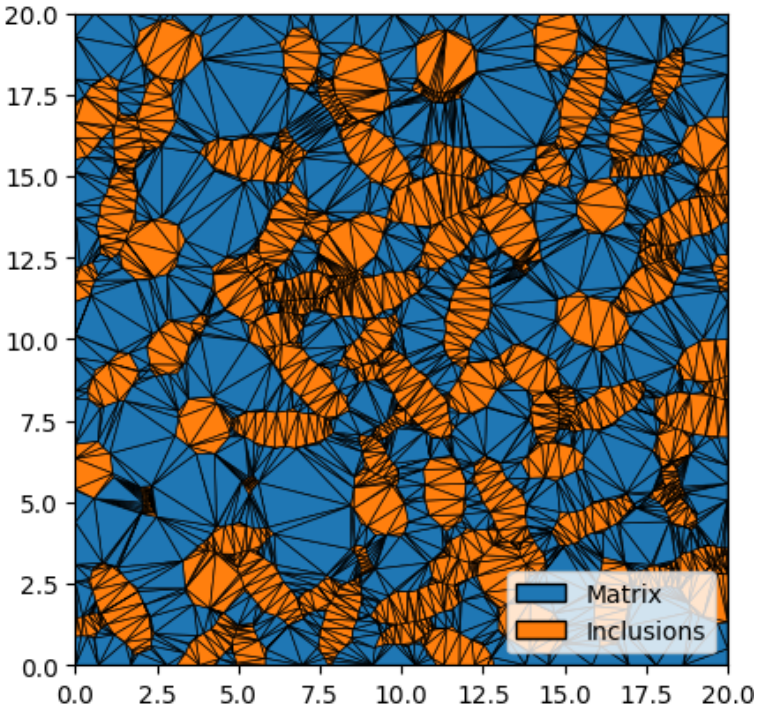


Fig. 4: *Size & Shape*

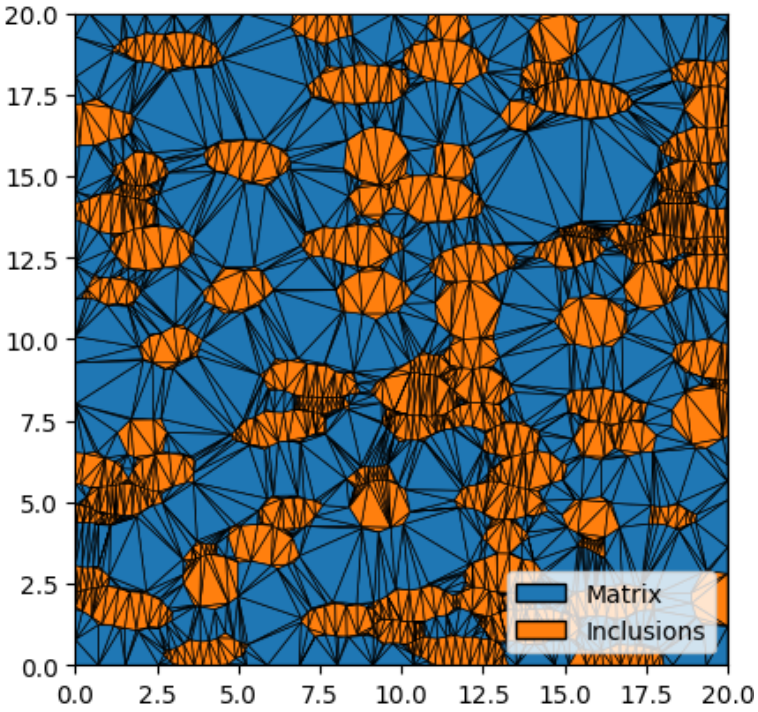


Fig. 5: *Oriented Grains and Amorphous Phases*

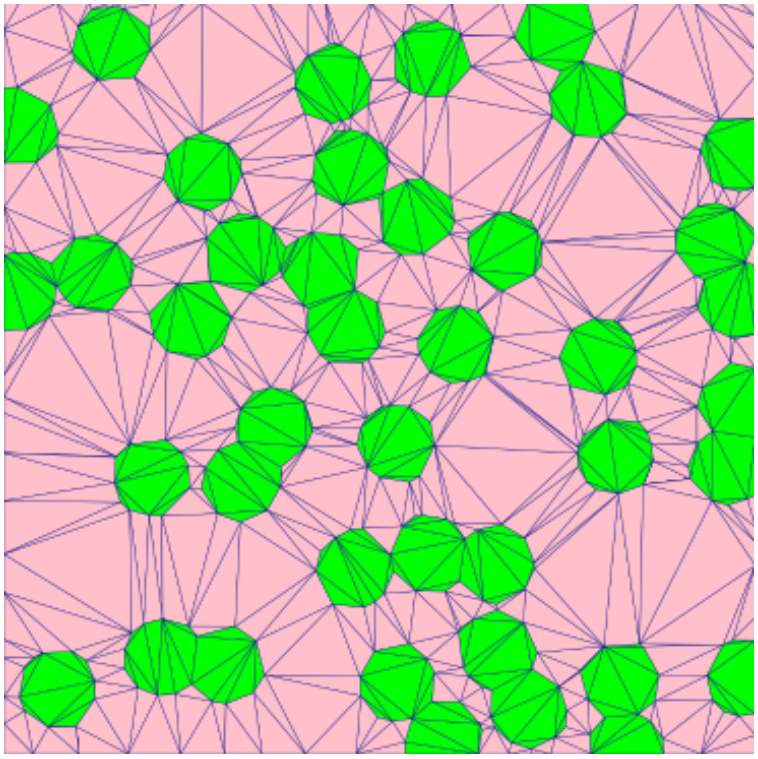


Fig. 6: *Plot Controls*

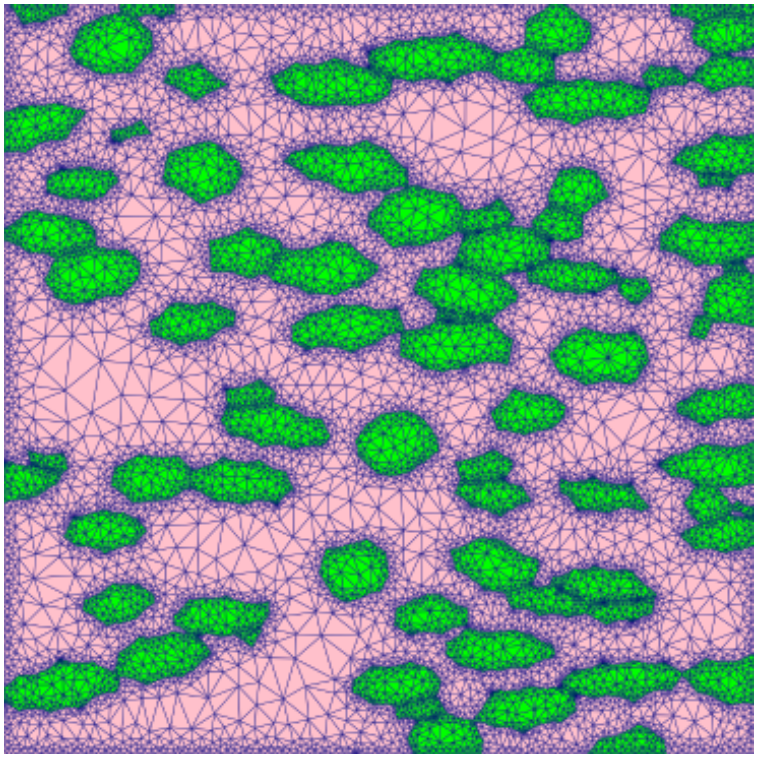


Fig. 7: *Culmination*

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="1.0" encoding="UTF-8"?>
<input>
    <material>
        <fraction> 1 </fraction>
        <shape> ellipse </shape>
        <a>
            <dist_type> uniform </dist_type>
            <loc> 0.05 </loc>
            <scale> 0.35 </scale>
        </a>
        <b> 0.05 </b>
        <orientation> random </orientation>
        <color> #eaaa00 </color>
    </material>

    <material>
        <fraction> 2 </fraction>
        <shape> circle </shape>
```

(continues on next page)

(continued from previous page)

```
        <diameter>
            <dist_type> lognorm </dist_type>
            <scale> 0.06 </scale>
            <s> 0.6 </s>
        </diameter>
        <color> #00263A </color>
    </material>

    <domain>
        <shape> rectangle </shape>
        <side_lengths> (2, 3) </side_lengths>
    </domain>

    <settings>
        <mesh_min_angle> 20 </mesh_min_angle>

        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

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

        <seeds_kwargs>
            <edgecolor> w </edgecolor>
            <linewidth> 0.5 </linewidth>
        </seeds_kwargs>

        <poly_kwargs>
            <edgecolors> w </edgecolors>
            <linewidth> 0.5 </linewidth>
        </poly_kwargs>

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

Material 1 - Ellipses

```
<material>
    <fraction> 1 </fraction>
    <shape> ellipse </shape>
    <a>
        <dist_type> uniform </dist_type>
        <loc> 0.05 </loc>
        <scale> 0.35 </scale>
    </a>
    <b> 0.05 </b>
    <orientation> random </orientation>
```

(continues on next page)

(continued from previous page)

```
    <color> #eaaa00 </color>
</material>
```

There are two materials, in a 1:2 ratio based on volume. The first material consists of ellipses and the semi-major axes are uniformly distributed, $A \sim U(0.05, 0.40)$. The semi-minor axes are fixed at 0.05, meaning the aspect ratio of these seeds are 1-8. The orientation angles of the ellipses are uniform random in distribution.

Material 2 - Circles

```
<material>
    <fraction> 2 </fraction>
    <shape> circle </shape>
    <diameter>
        <dist_type> lognorm </dist_type>
        <scale> 0.06 </scale>
        <s> 0.6 </s>
    </diameter>
    <color> #00263A </color>
</material>
```

The second material consists of circles, which have a diameter that is log-normally distributed, $D \sim 0.06e^{N(0,0.5)}$.

Domain Geometry

```
<domain>
    <shape> rectangle </shape>
    <side_lengths> (2, 3) </side_lengths>
</domain>
```

These two materials fill a rectangular domain. The bottom-left corner of the rectangle is the origin, which puts the rectangle in the first quadrant. The width of the rectangle is 2 and the height is 3.

Settings

```
<settings>
    <mesh_min_angle> 20 </mesh_min_angle>

    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
    </filetypes>

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

    <seeds_kwargs>
        <edgecolor> w </edgecolor>
        <linewidth> 0.5 </linewidth>
    </seeds_kwargs>
```

(continues on next page)

(continued from previous page)

```
    <poly_kwargs>
        <edgecolors> w </edgecolors>
        <linewidth> 0.5 </linewidth>
    </poly_kwargs>

    <tri_kwargs>
        <edgecolor> w </edgecolor>
        <linewidth> 0.1 </linewidth>
    </tri_kwargs>
</settings>
```

The aspect ratio of elements in the triangular mesh is controlled by setting the minimum interior angle for the elements at 20 degrees.

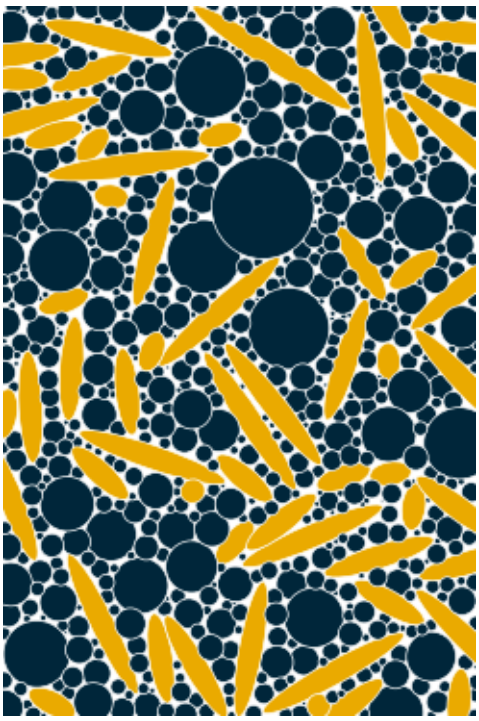
The function will output only plots of the microstructure process (no text files), and those plots are saved as PNGs. They are saved in a folder named `elliptical_grains`, in the current directory (i.e `./elliptical_grains`).

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

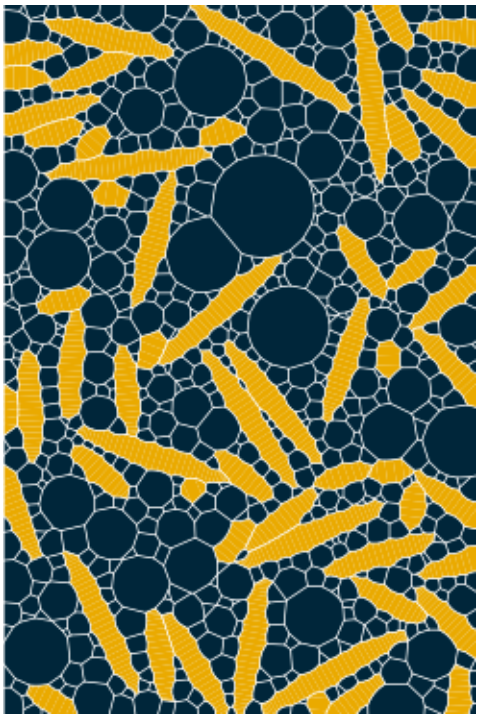
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

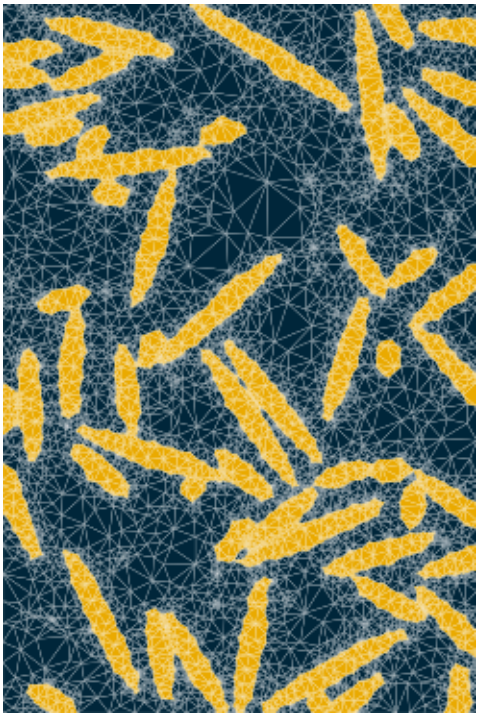
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
        </filetypes>

        <plot_axes> False </plot_axes>
        <color_by> seed number </color_by>
        <colormap> Paired </colormap>
    </settings>
</input>
```

Material 1

```
<material>
    <shape> circle </shape>
    <size> 0.09 </size>
</material>
```

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

Domain Geometry

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

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

Settings

```
<settings>
    <directory> minimal </directory>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
    </filetypes>

    <plot_axes> False </plot_axes>
    <color_by> seed number </color_by>
    <colormap> Paired </colormap>
</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`).

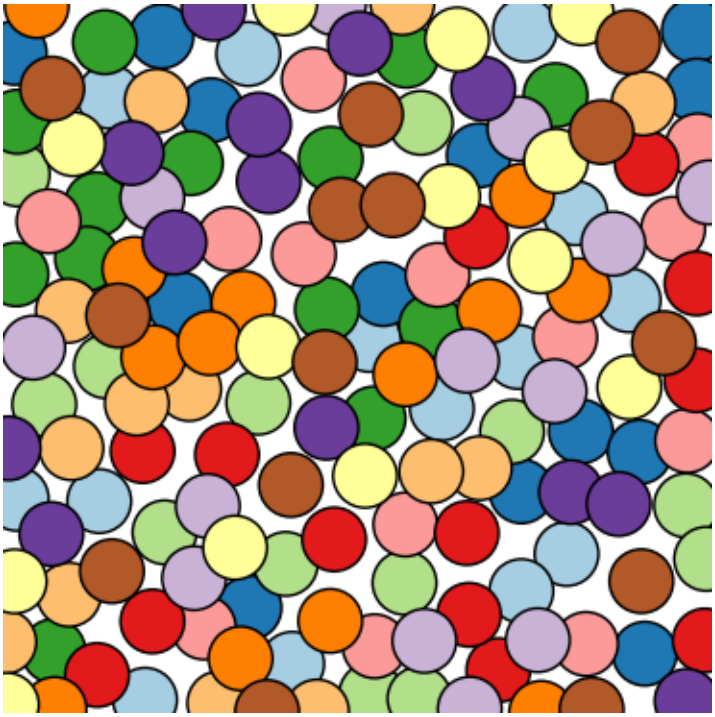
The axes are turned off in these plots, creating PNG files with minimal whitespace.

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

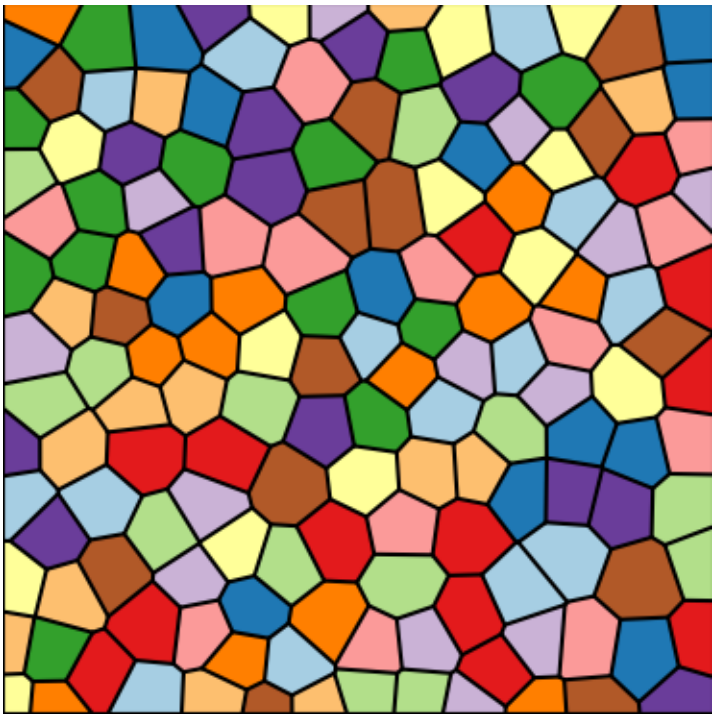
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

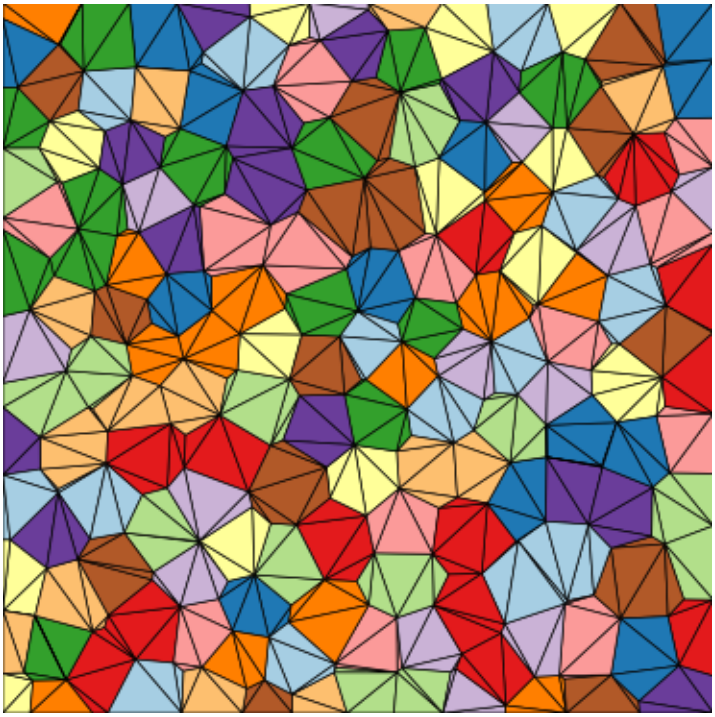
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



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>
        </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>
```

(continues on next page)

(continued from previous page)

```
        </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>
            <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>
```

(continues on next page)

(continued from previous page)

```
        <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>
            <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

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

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

```
<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>
```

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

```
<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>
        <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>
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

Settings

```
<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>
```

(continues on next page)

(continued from previous page)

```
    <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>
```

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`).

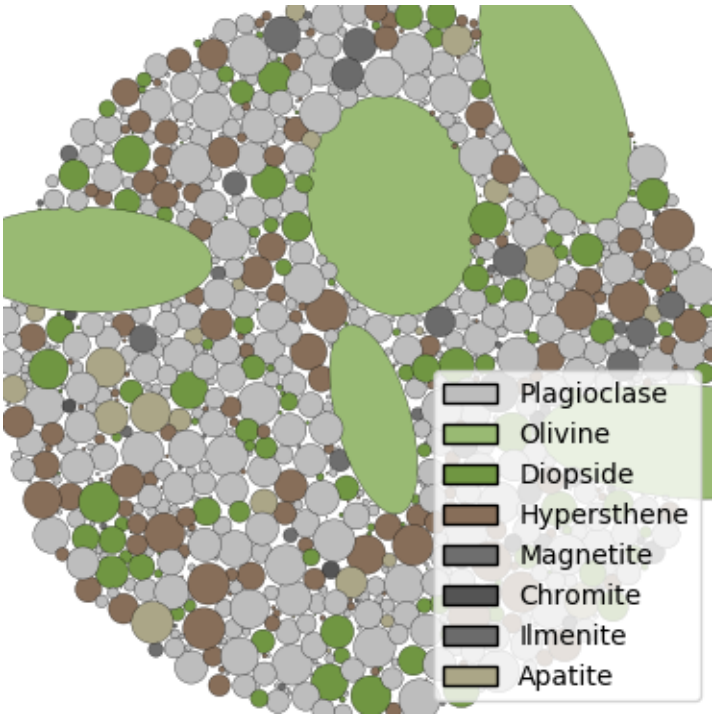
The axes are turned off in these plots, creating PNG files with minimal whitespace.

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

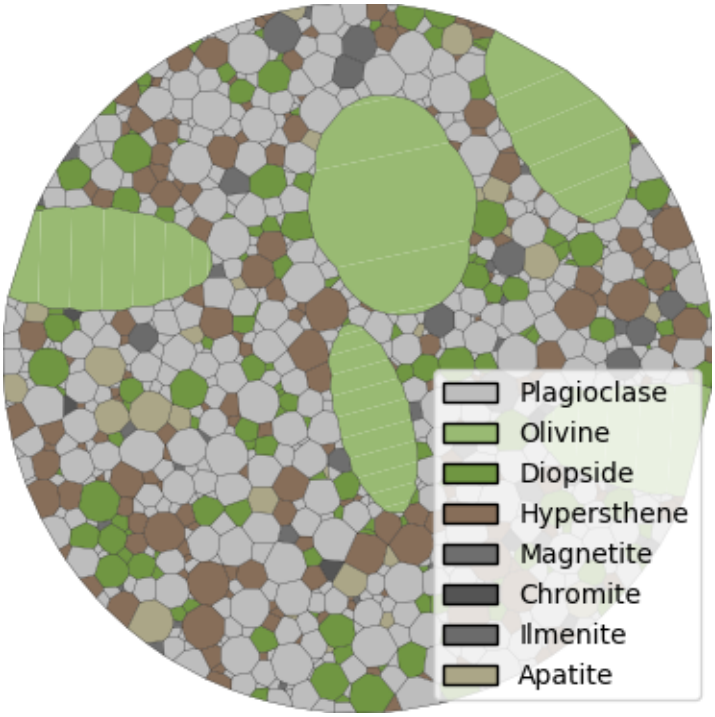
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh. Verification plots are also generated, since the `<verify>` flag is on.

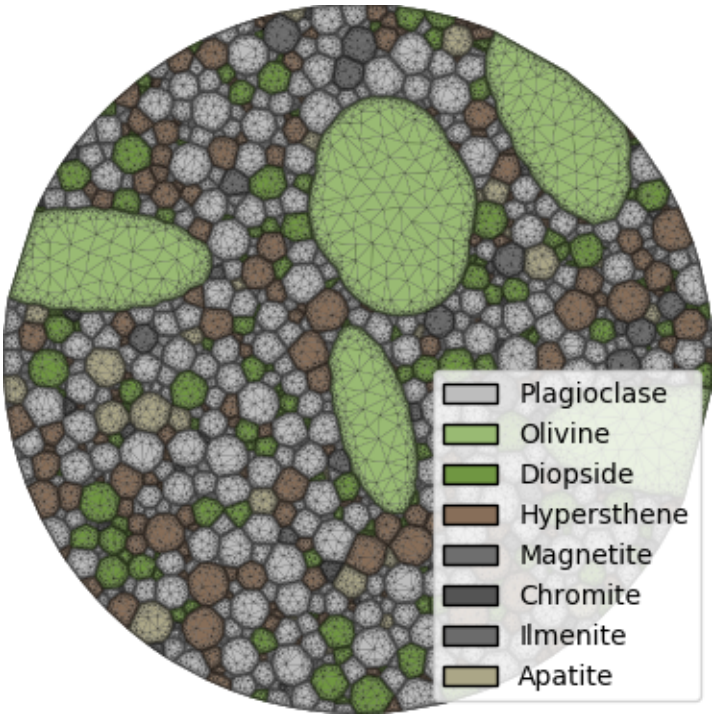
Seeding Plot



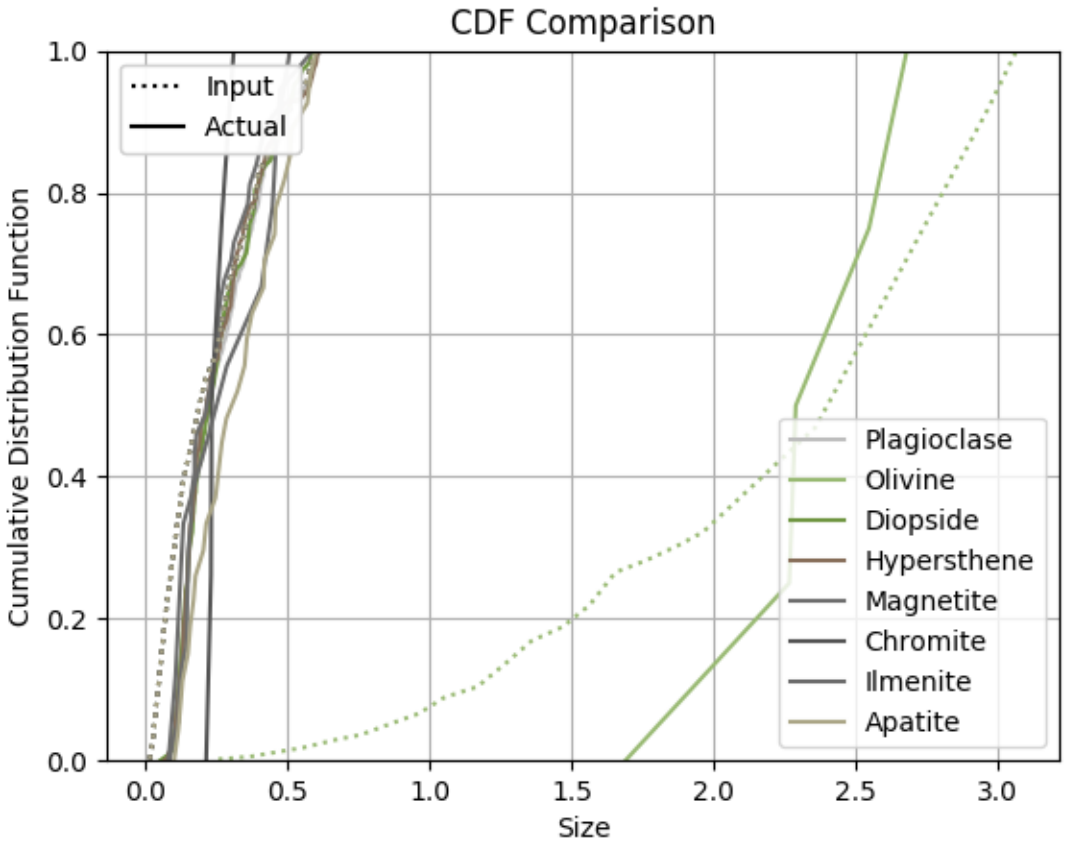
Polygon Mesh Plot



Triangular Mesh Plot



Crystal Size Distribution Comparison Plot



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:

```
<?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>
```

(continues on next page)

(continued from previous page)

```
    </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>
    </domain>

    <settings>
        <directory> two_phase_3D </directory>
        <filetypes>
            <seeds_plot> png </seeds_plot>
            <poly_plot> png </poly_plot>
            <tri_plot> png </tri_plot>
            <verify_plot> png </verify_plot>
        </filetypes>
        <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>
```

Material 1

```
<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>
```

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

Material 2

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

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

Domain Geometry

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

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

Settings

```
<settings>
    <directory> two_phase_3D </directory>
    <filetypes>
        <seeds_plot> png </seeds_plot>
        <poly_plot> png </poly_plot>
        <tri_plot> png </tri_plot>
        <verify_plot> png </verify_plot>
    </filetypes>
    <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>
```

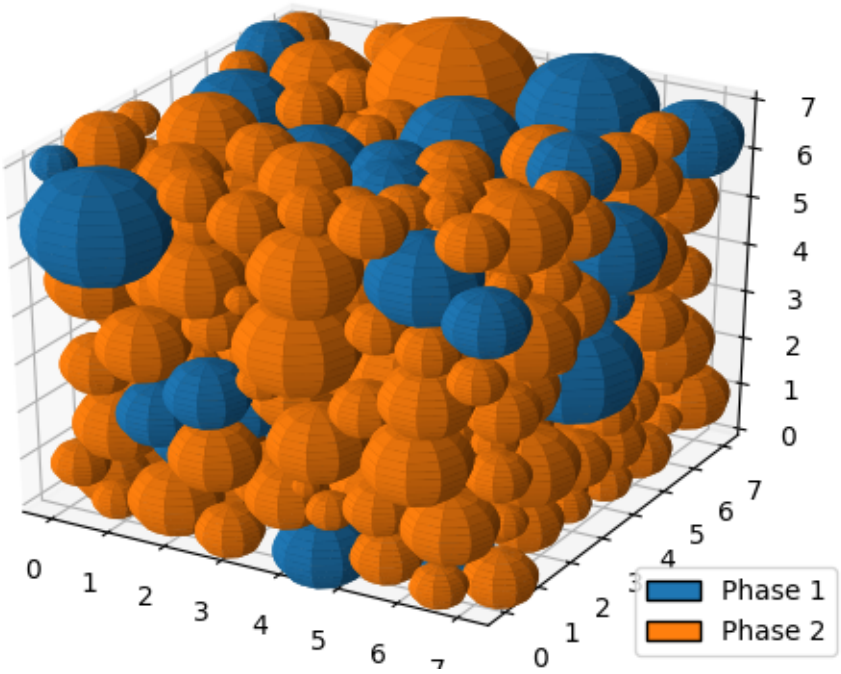
The function will output plots of the microstructure process and those plots are saved as PNGs. They are saved in a folder named `two_phase_3D`, in the current directory (i.e `./two_phase_3D`).

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

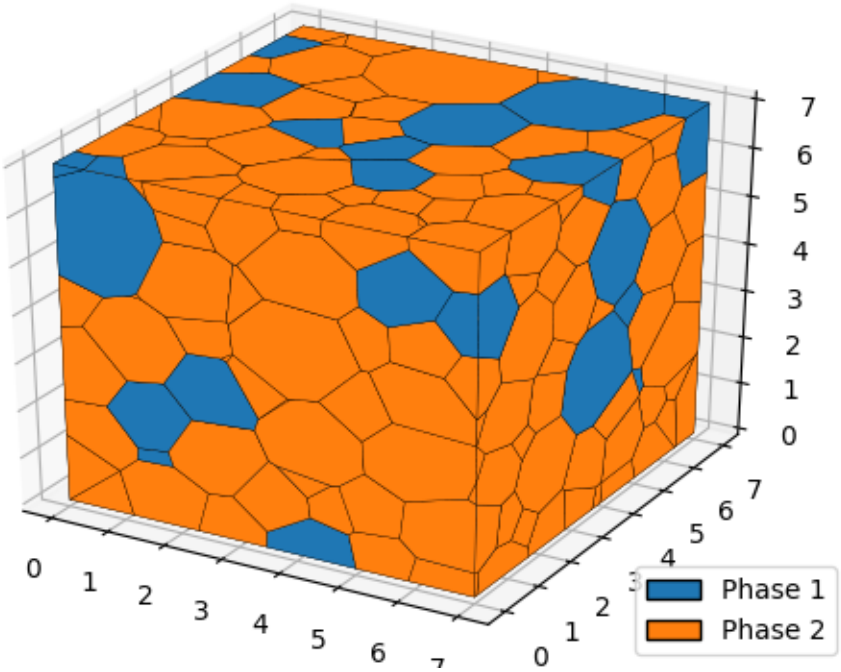
Output Files

The three plots that this file generates are the seeding, the polygon mesh, and the triangular mesh.

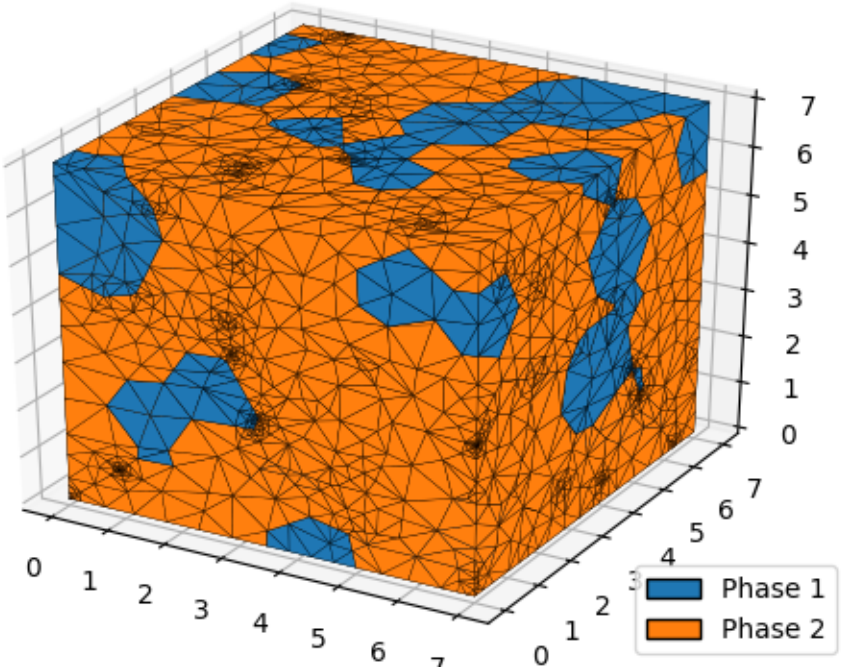
Seeding Plot



Polygon Mesh Plot



Triangular Mesh Plot



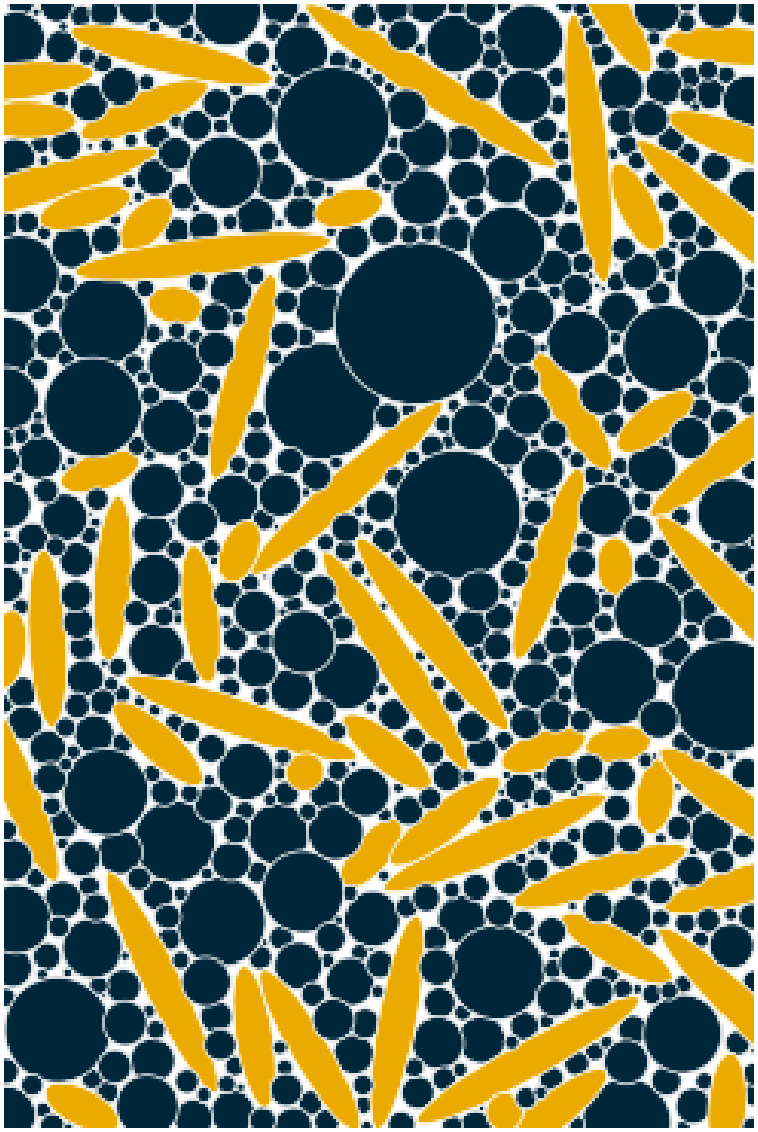


Fig. 8: *Elliptical Grains*

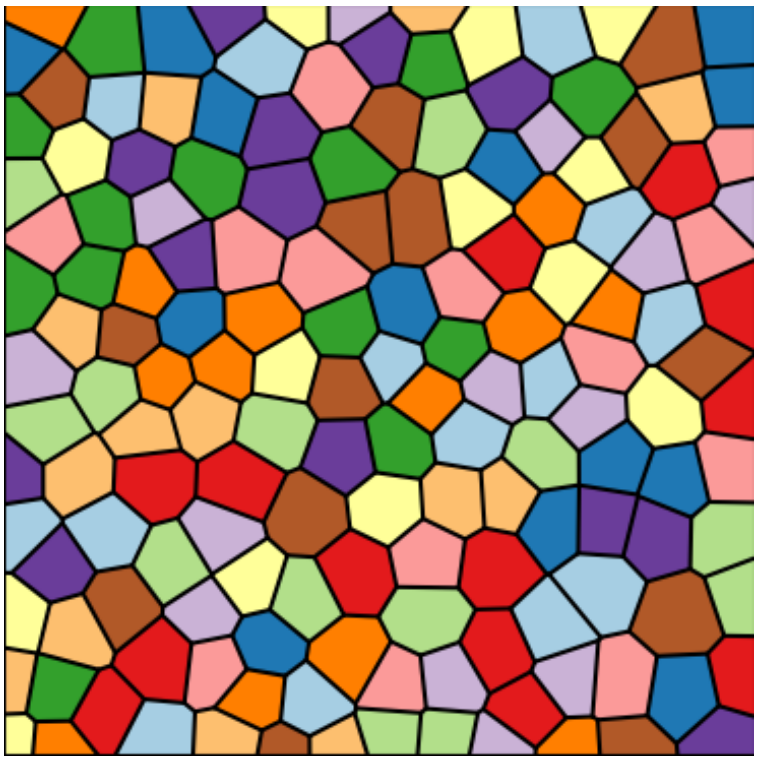


Fig. 9: *Minimal Example*

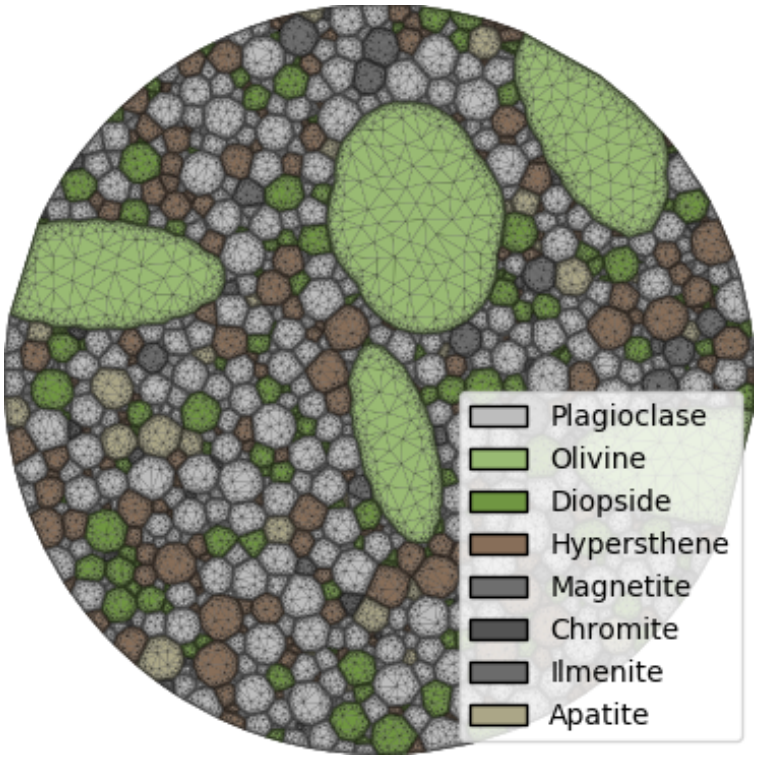


Fig. 10: *Picritic Basalt*

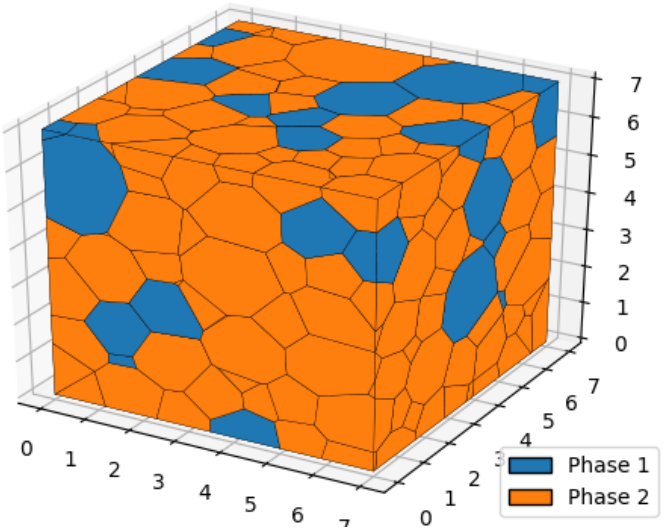


Fig. 11: *Two Phase 3D Example*

3.2.3 Examples Using the Python Package

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

import matplotlib.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
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')
```

(continues on next page)

(continued from previous page)

```
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)
```

Domain

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

Seeds

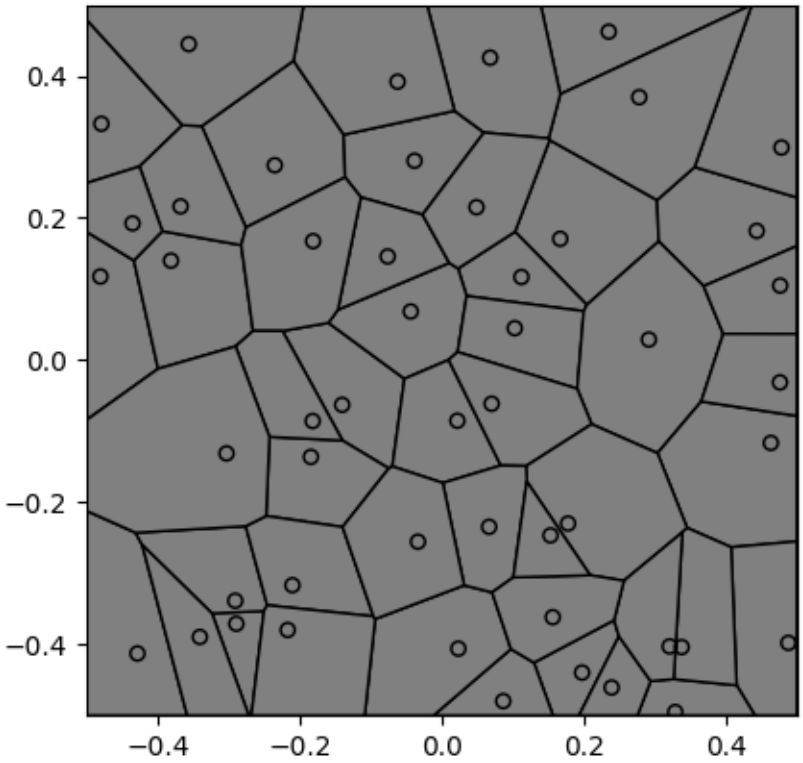
A set of 50 seed circles with small radius is initially created. Calling the *position()* method positions the points according to random uniform distributions in the domain.

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.

Output File

The output PNG file of this script is:



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


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

```

(continues on next page)

(continued from previous page)

```
# 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)
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])
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')
```

(continues on next page)

(continued from previous page)

```
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)
```

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.

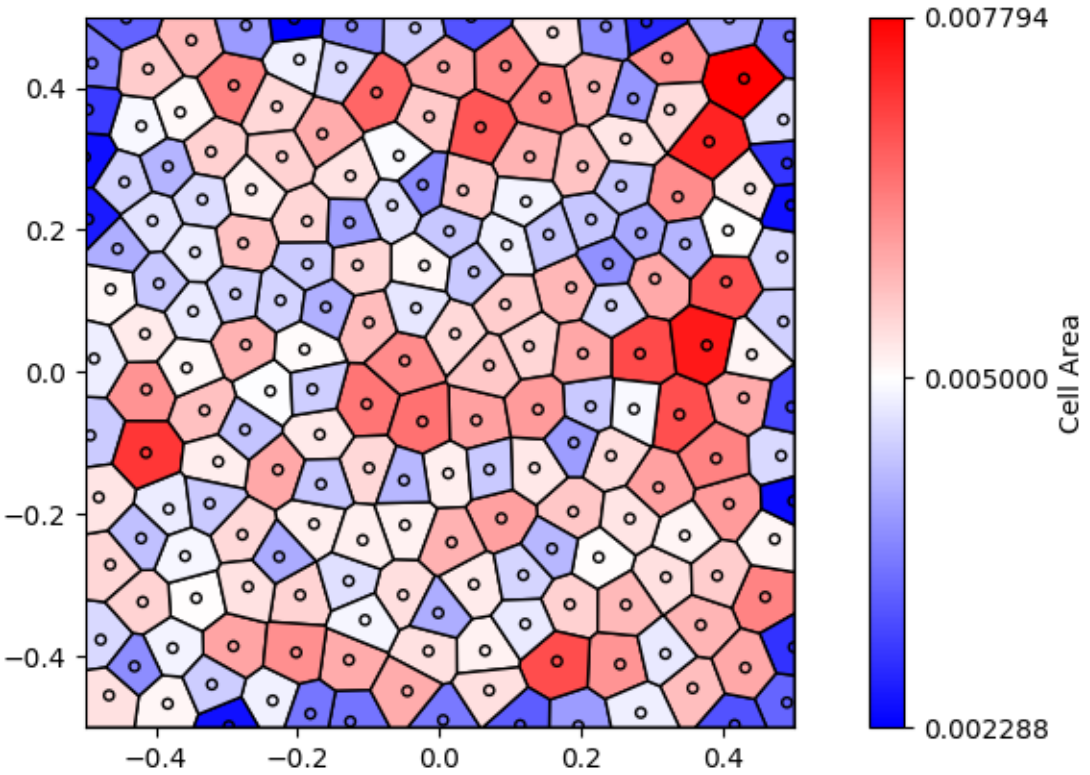
Plotting

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.

Output File

The output PNG file of this script is:

[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)



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 matplotlib.pyplot as plt
import microstructpy as msp
import numpy as np
import scipy.integrate
import scipy.stats


# Define the domain
domain = msp.geometry.Rectangle(corner=(0, 0), side_lengths=(8, 12))

```

(continues on next page)

(continued from previous page)

```
# Define the material phases
r_dist = scipy.stats.lognorm(s=0.7, scale=0.1)
matrix_phase = {'fraction': 2,
                'material_type': 'matrix',
                'shape': 'circle',
                'r': r_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)
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])
```

(continues on next page)

(continued from previous page)

```
            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[a].phase) for a 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)
if not os.path.exists(dirs):
    os.makedirs(dirs)
plt.savefig(filename)
```

Domain

The domain of the microstructure is a `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

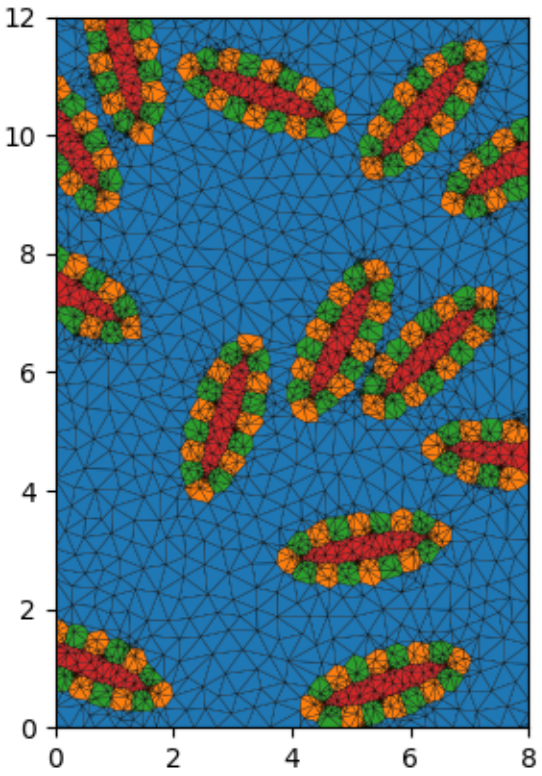
The seeds are converted into a triangular mesh by first constructing a polygon mesh. Each material is solid, except for the first which is designated as a matrix phase. Mesh quality controls are specified to prevent high aspect ratio triangles.

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.

Output File

The output PNG file of this script is:



Microstructure from Image

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 matplotlib.image as mpim
import matplotlib.pyplot as plt
import microstructpy as msp
import numpy as np


# 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]

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]
```

(continues on next page)

(continued from previous page)

```
        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:
            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),
```

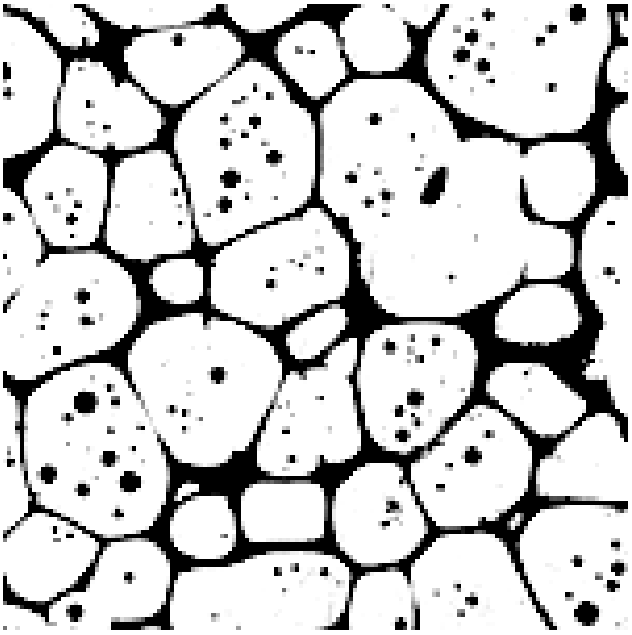
(continues on next page)

(continued from previous page)

```
                                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)

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='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:



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

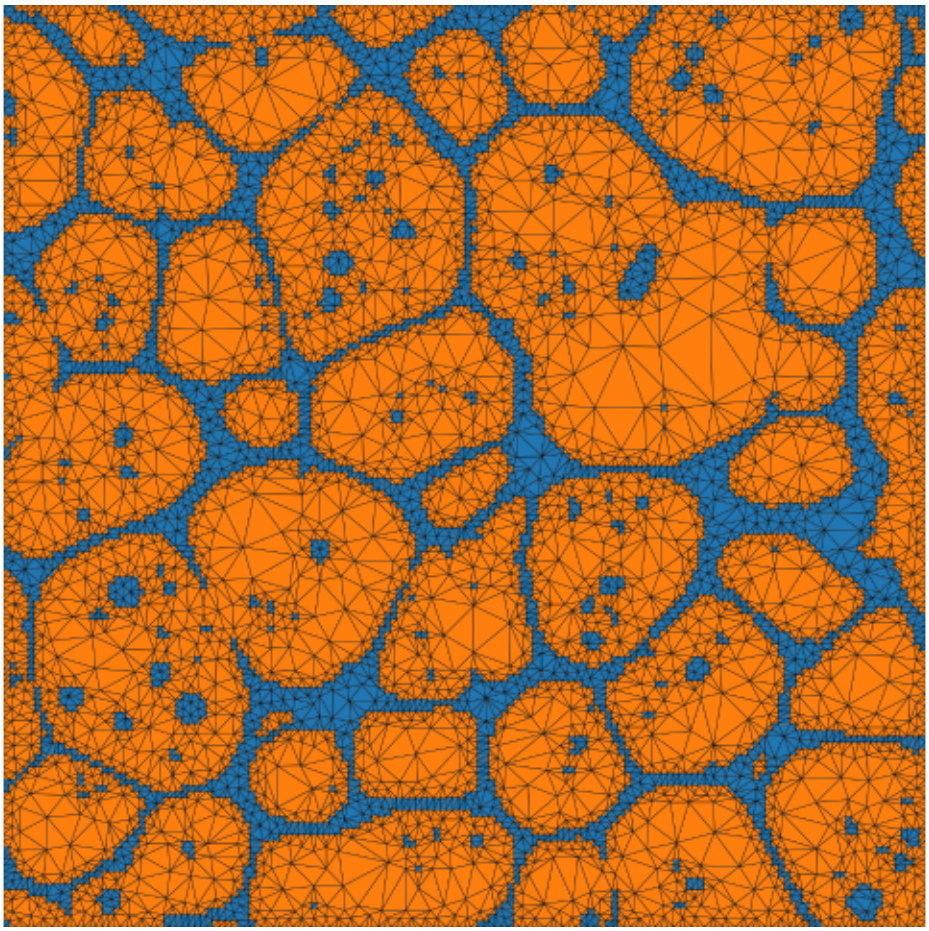
The axes of the plot are switched off, then the triangle mesh is plotted. The color of each triangle is set by the phase.

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.

Output File

The output PNG file of this script is:



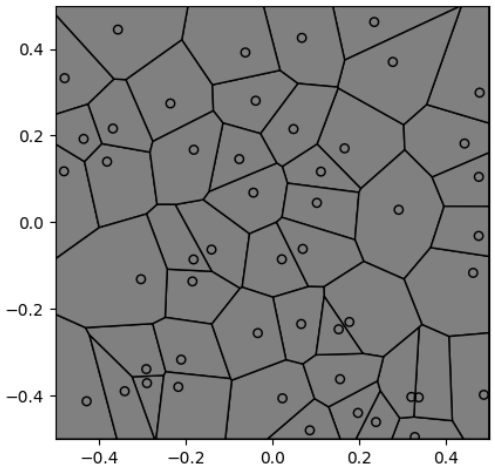


Fig. 12: *Standard Voronoi Diagram*

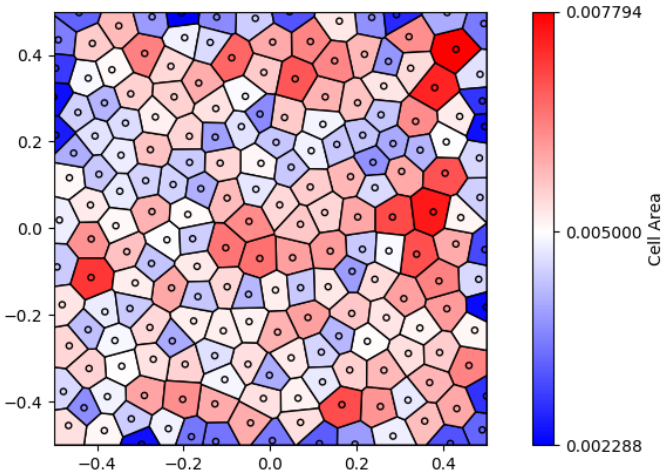


Fig. 13: *Uniform Seeding Voronoi Diagram*

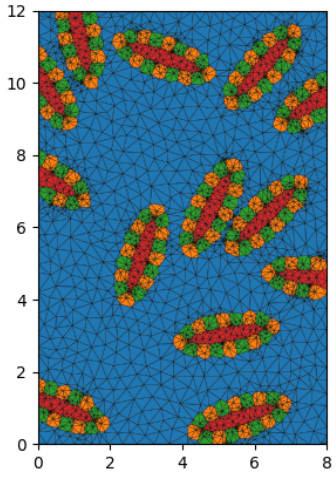


Fig. 14: *Grain Neighborhoods*

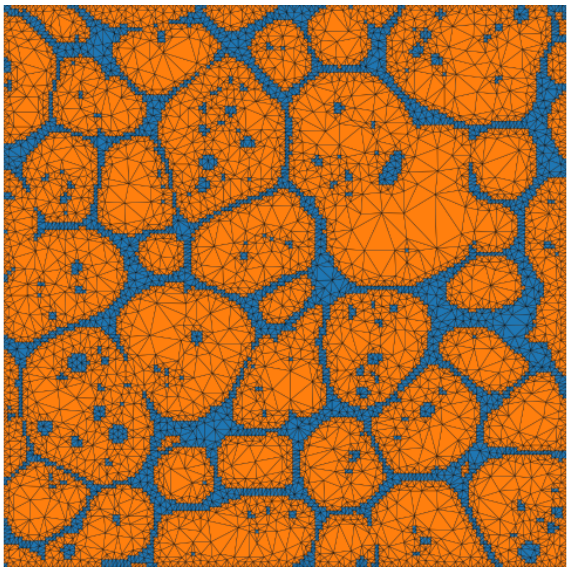


Fig. 15: *Microstructure from Image*

3.3 Command Line Guide

3.3.1 Basics

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.

For example, to run the XML file that creates the microstructure on the front page, run:

```
microstructpy --demo=docs_banner.xml
```

This command will copy `docs_banner.xml` to the current working directory, then run that XML file. For your own input files, run the command:

```
microstructpy path/to/my/input_file.xml
```

Note that relative and absolute file paths are acceptable.

To run multiple input files:

```
microstructpy file_1.xml file_2.xml file_3.xml
```

or:

```
microstructpy file_*.xml
```

To run all of the demonstration files, use the command:

```
microstructpy --demo=all
```

Note that this may take some time.

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 a 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).

Minimal Input File

Input files for MicroStructPy must be in XML format and included a minimum of 2 pieces of information: the material phases and the domain. A minimal input file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<input>
    <material>
        <shape> circle </shape>
        <size> 0.1 </size>
    </material>

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

This will create a microstructure with approximately circular grains that fill a domain that is 10x larger. MicroStructPy will output three files: `seeds.txt`, `polymesh.txt`, and `trimesh.txt`. To output plots requires addition settings, described in the *Settings* section.

The *Material Phases* section describes more advanced materials specification, while the *Domain* section discusses specifying the geometry of the domain.

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

Also note that the order of fields in an XML file is not strictly important, since the file is converted into a dictionary.

3.3.2 Material Phases

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.

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>
        <size>
            <dist_type> lognorm </dist_type>
```

(continues on next page)

(continued from previous page)

```
            <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 one of SciPy's statistical functions, 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+1) bin locations and the second has the (n) bin heights. A PDF file could contain, for example:

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

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.

Grain Geometries

MicroStructPy supports the following grain geometries:

- Circle (2D)
- Ellipse (2D)
- Ellipsoid (3D)
- Rectangle (2D)

- Sphere (3D)
- Square (2D)

Each geometry 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

- radius (or r)
- diameter (or d)
- size (same as d)
- area

Ellipse

- a
- b
- size
- aspect_ratio
- angle_deg
- angle_rad
- angle (same as angle_deg)
- axes (equivalent to [a, b])
- matrix
- orientation (same as matrix)

Ellipsoid

- a
- b
- c
- size
- ratio_ab or ratio_ba
- ratio_ac or ratio_ca
- ratio_bc or ratio_cb
- rot_seq_deg
- rot_seq_rad
- rot_seq (same as rot_seq_deg)

- axes (equivalent to [a, b, c])
- matrix
- orientation (same as matrix)

Rectangle

- length
- width
- side_lengths (equivalent to [length, width])
- angle_deg
- angle_rad
- angle (same as angle_deg)
- matrix

Sphere

- radius (or r)
- diameter (or d)
- size (same as d)
- volume

Square

- side_length
- angle_deg
- angle_rad
- angle (same as angle_deg)
- matrix

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>
        </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>
```

Material Type

There are three types of materials supported by MicroStructPy: crystalline, amorphous, and void. For amorphous phases, the facets between cells of the same material type are removed before unstructured meshing. Several aliases are available for each type, given in the list below.

- **crystalline**
  - solid
  - granular
- **amorphous**
  - matrix
  - glass

- **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>
        <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.

Multiple amorphous and void phases can be present in the material.

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 on SciPy's statistical functions page.

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>
            <loc> 0.5 </loc>
            <n> 9 </n>
```

(continues on next page)

(continued from previous page)

```
            <p> 0.5 </p>
        </position>
        <position> <!-- y -->
            <dist_type> uniform </dist>
            <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.

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 page.

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>
```

3.3.3 Domain

MicroStructPy supports the following domain geometries:

- Box (3D)
- Circle (2D)
- Cube (3D)
- Ellipse (2D)
- Rectangle (2D)
- Square (2D)

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.

Box

The parameters available for defining a 3D box domain are:

- side_lengths
- center
- corner (i.e. $(x, y, z)_{min}$)
- limits (i.e. $[[x_{min}, x_{max}], [y_{min}, y_{max}], [z_{min}, z_{max}]]$)
- bounds (same as limits)

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>
```

Circle

The parameters available for defining a 2D circle domain are:

- radius (or r)
- diameter (or d)
- size (same as diameter)
- area
- center

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>
```

Cube

The parameters available for defining a 3D cube domain are:

- side_length
- center
- corner (i.e. $(x, y, z)_{min}$)

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>
```

(continues on next page)

(continued from previous page)

```
        <side_length> 10 </side_length>
        <corner> (0, 0, 0) </corner>
    </domain>

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

Ellipse

The parameters available for defining a 2D ellipse domain are:

- a
- b
- axes
- size
- aspect_ratio
- angle_deg
- angle_rad
- angle (same as angle_deg)
- matrix
- orientation (same as matrix)
- center

Below are some example ellipse domain definitions.

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

(continues on next page)

(continued from previous page)

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

Rectangle

The parameters available to define a 2D rectangle domain are:

- length
- width
- side_lengths
- center
- corner (i.e. $(x, y)_{min}$)
- limits (i.e. $[[x_{min}, x_{max}], [y_{min}, y_{max}]]$)
- bounds (same as limits)

Below are some example rectangle domain definitions.

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

Square

The parameters available to define a 2D square domain are:

- side_length
- center
- corner (i.e. $(x, y)_{min}$)

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

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

3.3.4 Settings

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>
        <verbose> False </verbose>
        <restart> True </restart>
        <directory> . </directory>

        <filetypes>
            <seeds> txt </seeds>
            <poly> txt </poly>
            <tri> txt </tri>
        </filetypes>
```

(continues on next page)

(continued from previous page)

```
        <rng_seeds>
            <position> 0 </position>
        </rng_seeds>

        <rtol> fit </rtol>
        <mesh_max_volume> inf </mesh_max_volume>
        <mesh_min_angle> 0 </mesh_min_angle>
        <mesh_max_edge_length> inf </mesh_max_edge_length>

        <verify> True </verify>

        <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>
```

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.

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.

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

  Currently the only option is to output the seed geometries as a cache txt file.

- 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** (3D only)

  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 surfaces between grains.

- 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, the default behavior is not to save that output. The exception is, if `<restart> True </restart>`, then the seeds, poly mesh, and tri mesh will all be output to txt files. The subsections below describe the options for each subfield.

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.

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>
```

(continues on next page)

(continued from previous page)

```
    <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.

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 page.

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>
    </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.

3.4 Python Package Guide

The python package for MicroStructPy includes the following classes:

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

Seeds are given a geometry and a material number, SeedLists are lists of Seeds, the PolyMesh can be created from a SeedList, and finally the TriMesh can be created from a PolyMesh. This is the flow of information built into the MicroStructPy command line interface (CLI). Custom algorithms for seeding or meshing can be implemented using the classes above and a few key methods.

The following describes the 3-step process of generating a microstructure mesh in MicroStructPy, including the relevant classes and methods. For a complete list of MicroStructPy classes, visit the *API* page. Visit the *Examples* page for examples using the API.

3.4.1 0. List of Seed Geometries

The list of seed geometries is a `SeedList`. The SeedList can be created from a list of `Seed` instances, which each contain a geometry and a phase.

A SeedList can also be generated from a list of material phase dictionaries and a total seed volume using the `SeedList.from_info()` class method. The default seed volume is the volume of the domain. For more information on how to format the phase information, see the *Phase Dictionaries* below.

One convenience function is `Seed.factory()`, which takes in a geometry name and keyword arguments and returns a Seed with that geometry.

3.4.2 1. Pack Geometries into Domain

The standard domain is a geometry from the *microstructpy.geometry package*. To pack the geometries into the domain, the centers of the seeds are specified such that there is a tolerable about of overlap with other seeds, if any.

The standard method for positioning seeds in a domain is `SeedList.position()`. This function updates the `Seed.position` property of each Seed in the SeedList. The centers of all the seeds are within the domain geometry.

3.4.3 2. Tessellate the Domain

A tessellation of the domain divides its interior into polygonal/polyhedral cells with no overlap or gaps between them. This tessellation is stored in a `PolyMesh` class. The default method for creating a PolyMesh from a positioned list

of seeds and a domain is *PolyMesh.from_seeds()*. This method creates a Voronoi-Laguerre diagram using the Voro++ package. Note that the only supported 3D domains are cubes and boxes.

3.4.4 3. Unstructured Meshing

Unstructured (triangular or tetrahedral) meshes can be used in finite element software to analyze the behavior of the microstructure. Their data are contained in the *TriMesh* class. This mesh can be created from a polygonal tessellation using the *TriMesh.from_polymesh()* method. The mesh can be output to several different file formats.

The unstructured meshes are generated using Triangle in 2D, TetGen in 3D, and MeshPy is the wrapper.

3.4.5 File I/O

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.

3.4.6 Plotting

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

- *SeedList.plot()*
- *SeedList.plot_breakdown()*
- *PolyMesh.plot()*
- `PolyMesh.plot_facet()`
- *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.

3.4.7 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 SciPy statistical distribution. 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.

3.5 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.

3.5.1 Installation

These are problems encountered when installing MicroStructPy.

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.

3.5.2 Command Line Interface

These are problems encountered when running `microstructpy input_file.xml`.

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`.

Could not connect to display

Problem Description

The program crashes while trying to plot and there is an error message that says:

```
QXcbConnection: Could not connect to display
```

The `show_plots` setting in the input file is set to False, so a display should not be necessary.

Problem Solution

The default behavior for matplotlib is to use an interactive backend. If MicroStructPy is running in an environment that cannot create windows, then matplotlib will crash.

This problem can be solved in two way, 1) run MicrostructPy in an environment with windows or 2) set the default behavior of matplotlib to use a non-interactive backend. For option 2, add the following to your `matplotlibrc` file:

```
backend : agg
```

The path to `matplotlibrc` file is OS-dependent and explained on the matplotlib website. For Linux, the path is `~/.config/matplotlib/matplotlibrc`.

'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
```

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.

3.6 microstructpy package

3.6.1 Subpackages

microstructpy.geometry package

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.

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.

n_dim
number of dimensions, 3

Type int

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

Type float

class microstructpy.geometry.**Cube**(***kwargs*)
Bases: *microstructpy.geometry.box.Box*

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.

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.

side_length
length of the side of the cube.

Type float

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.

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** – Alias for 2x *r*.
- **radius** – Alias for *r*.
- **d** – Alias for 2x *r*.
- **size** – Alias for 2x *r*.
- **position** – Alias for *center*.

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 scipy.stats distributions.

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 *microstructpy.geometry.Circle*.

Returns Expected value of the area of the circle.

Return type float

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.

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

Type float

n_dim
number of dimensions, 2

Type int

volume
alias for area

Type float

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.

Parameters

- **center** (*list, optional*) – The ellipse center. Defaults to (0, 0).
- **axes** (*list, optional*) – A 2-element list of semi-axes. 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 rotation angle, in degrees.
- **angle_deg** (*float, optional*) – The rotation angle, in degrees.
- **angle_rad** (*float, optional*) – The rotation angle, in radians.
- **matrix** (*2x2 array, optional*) – The rotation matrix.
- **a** – Alias for *axes[0]*.
- **b** – Alias for *axes[1]*.
- **angle_deg** – Alias for *angle*.
- **angel_rad** – Alias for *pi * angle / 180*.
- **orientation** – 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.[1]

Parameters **x1** (*float*) – Center position of first circle.

Returns An Nx3 list of the (x, y, r) data of each circle approximating the ellipse.

[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.

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 scipy.stats distributions.

If an ellipse is specified by size, the expected value is computed as follows.

$$\begin{aligned}\mathbb{E}[A] &= \frac{\pi}{4}[S^2] \\ &= \frac{\pi}{4}(\mu_S^2 + \sigma_S^2)\end{aligned} \tag{3.1}$$

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:

$$\begin{aligned}\mathbb{E}[A] &= \pi\mathbb{E}[KB^2] \\ &= \pi\mu_K(\mu_B^2 + \sigma_B^2)\end{aligned} \tag{3.3}$$

Finally, if the ellipse is specified by the first semi-axis and the aspect ratio, the expected value is computed by Monte Carlo:

$$\begin{aligned}\mathbb{E}[A] &= \pi\mathbb{E}\left[\frac{A^2}{K}\right] \\ &\approx \frac{\pi}{n}\sum_{i=1}^{n}\frac{A_i^2}{K_i}\end{aligned} \tag{3.5}$$

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 is heavily adapted from the least-squares-ellipse-fitting repository on GitHub. This repository 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*) – List of points to fit.

Returns An instance of the class that best fits the points.

[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)

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.

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*) – 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*) – Point or list of points.

Returns Set to True for points in ellipse.

Return type bool or numpy.ndarray

angle_deg
rotation angle, in degrees

Type float

angle_rad
rotation angle, in radians

Type float

area
area of ellipse, $A = \pi ab$

Type float

aspect_ratio
ratio of x-axis length to y-axis length

Type float

axes
list of semi-axes.

Type tuple

bound_max
maximum bounding circle of ellipse, (x, y, r)

Type tuple

bound_min
minimum interior circle of ellipse, (x, y, r)

Type tuple

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

Type list

matrix
rotation matrix

Type numpy.ndarray

n_dim
number of dimensions, 2

Type int

orientation
rotation matrix

Type numpy.ndarray

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

Type list

size
diameter of equivalent area circle

Type float

volume
alias for area

Type float

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

- **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** (*3x3 numpy.ndarray, optional*) – A 3x3 rotation matrix expressing the orientation of the ellipsoid. Defaults to the identity.
- **position** – Alias for *center*.
- **a** – Alias for *axes[0]*.
- **b** – Alias for *axes[1]*.
- **c** – Alias for *axes[2]*.
- **orientation** – 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*) – Center position of the first sphere.

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.[3]

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 Axes3D.plot_surface method to add an ellipsoid to the current axes. The keyword arguments are passed 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.

[3] 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)

Parameters **points** (*list*) – 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 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 scipy.stats distributions.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi ABC] \\ &= \frac{4}{3}\pi\mathbb{E}[A]\mathbb{E}[B]\mathbb{E}[C] \\ &= \frac{4}{3}\pi\mu_A\mu_B\mu_C\end{aligned} \tag{3.7}$$

If the ellisoid is specified by size and aspect ratios, then the expected volume is computed by:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{\pi}{6}S^3] \\ &= \frac{\pi}{6}(\mu_S^3 + 3\mu_S\sigma_S^2 + \gamma_1\sigma_S^3)\end{aligned} \tag{3.10}$$

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.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*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

axes

the 3 semi-axes of the ellipsoid

Type tuple

bound_max

maximum bounding sphere, (x, y, z, r)

Type tuple

bound_min
minimum interior sphere, (x, y, z, r)

Type tuple

coefficients
coeffificients 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

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

Type list

matrix
A 3x3 rotation matrix

Type numpy.ndarray

matrix_quadeq
Matrix of the quadratic equation

Type numpy.ndarray

matrix_quadform
Matrix of the quadratic form

Type numpy.ndarray

n_dim
number of dimensions, 3

Type int

orientation
A 3x3 rotation matrix

Type numpy.ndarray

ratio_ab
ratio of x-axis length to y-axis length

Type float

ratio_ac
ratio of x-axis length to z-axis length

Type float

ratio_ba
ratio of y-axis length to x-axis length

Type float

ratio_bc
ratio of y-axis length to z-axis length

Type float

ratio_ca
ratio of z-axis length to x-axis length

Type float

ratio_cb
ratio of z-axis length to y-axis length

Type float

rot_seq_deg
rotation sequence, with angles in degrees

Type list

rot_seq_rad
rotation sequence, with angles in radiands

Type list

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

Type list

size
diameter of equivalent volume sphere

Type float

volume
volume of ellipsoid, $V = \frac{4}{3}\pi abc$

Type float

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.

Parameters

- **length** (*float, optional*) – Length of the rectangle.
- **width** (*float, optional*) – Width of the rectangle.
- **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** (*2x2 array, optional*) – The rotation matrix.

approximate(*x1=None*)

classmethod area_expectation(***kwargs*)

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 *microstructpy.geometry.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.

angle

angle_deg

angle_rad

area
area of rectangle

Type float

length

n_dim
number of dimensions, 2

Type int

width

class microstructpy.geometry.**Square**(**kwargs)
Bases: *microstructpy.geometry.rectangle.Rectangle*

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.

classmethod area_expectation(**kwargs)

side_length
length of the side of the square.

Type float

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.

Parameters

- **r** (*float, optional*) – The radius of the n-sphere.
- **radius** (*float, optional*) – The radius of the n-sphere.
- **d** (*float, optional*) – The diameter of the n-sphere.
- **diameter** (*float, optional*) – The diameter of the n-sphere.
- **size** (*float, optional*) – The size of the n-sphere.
- **center** (*list, float, numpy.ndarray*) – The coordinates of the center.
- **position** (*list, float, numpy.ndarray*) – The coordinates of the center.

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

This function uses the 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.

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 __init__ function. The values for these keywords can be either constants or scip.stats distributions.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi R^3] \\ &= \frac{4}{3}\pi\mathbb{E}[R^3] \\ &= \frac{4}{3}\pi(\mu_R^3 + 3\mu_R\sigma_R^2 + \gamma_1\sigma_R^3)\end{aligned} \tag{3.12}$$

Parameters ****kwargs** – Keyword arguments, see *microstructpy.geometry.Sphere*.

Returns Expected value of the volume of the sphere.

Return type float

n_dim
number of dimensions, 3

Type int

volume
volume of sphere

Type float

Submodules

microstructpy.geometry.box module

Box

This module contains the Box class.

class microstructpy.geometry.box.**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.

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.

n_dim
number of dimensions, 3

Type int

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

Type float

class microstructpy.geometry.box.**Cube**(***kwargs*)
Bases: *microstructpy.geometry.box.Box*

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.

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.

side_length
length of the side of the cube.

Type float

microstructpy.geometry.circle module

class microstructpy.geometry.circle.**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.

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** – Alias for 2x *r*.
- **radius** – Alias for *r*.
- **d** – Alias for 2x *r*.
- **size** – Alias for 2x *r*.
- **position** – Alias for *center*.

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 scipy.stats distributions.

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 *microstructpy.geometry.Circle*.

Returns Expected value of the area of the circle.

Return type float

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.

area

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

Type float

n_dim

number of dimensions, 2

Type int

volume

alias for area

Type float

microstructpy.geometry.ellipse module

class microstructpy.geometry.ellipse.**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.

Parameters

- **center** (*list, optional*) – The ellipse center. Defaults to (0, 0).
- **axes** (*list, optional*) – A 2-element list of semi-axes. 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 rotation angle, in degrees.
- **angle_deg** (*float, optional*) – The rotation angle, in degrees.
- **angle_rad** (*float, optional*) – The rotation angle, in radians.
- **matrix** (*2x2 array, optional*) – The rotation matrix.
- **a** – Alias for *axes[0]*.
- **b** – Alias for *axes[1]*.
- **angle_deg** – Alias for *angle*.
- **angel_rad** – Alias for *pi * angle / 180*.
- **orientation** – 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.[1]

Parameters **x1** (*float*) – Center position of first circle.

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 scipy.stats distributions.

[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.

If an ellipse is specified by size, the expected value is computed as follows.

$$\mathbb{E}[A] = \frac{\pi}{4}[S^2] \tag{3.15}$$
$$= \frac{\pi}{4}(\mu_S^2 + \sigma_S^2) \tag{3.16}$$

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:

$$\mathbb{E}[A] = \pi\mathbb{E}[KB^2] \tag{3.17}$$
$$= \pi\mu_K(\mu_B^2 + \sigma_B^2) \tag{3.18}$$

Finally, if the ellipse is specified by the first semi-axis and the aspect ratio, the expected value is computed by Monte Carlo:

$$\mathbb{E}[A] = \pi\mathbb{E}\left[\frac{A^2}{K}\right] \tag{3.19}$$
$$\approx \frac{\pi}{n}\sum_{i=1}^{n}\frac{A_i^2}{K_i} \tag{3.20}$$

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 is heavily adapted from the least-squares-ellipse-fitting repository on GitHub. This repository 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*) – 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.

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

[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)

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*) – 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*) – Point or list of points.

Returns Set to True for points in ellipse.

Return type bool or numpy.ndarray

angle_deg
rotation angle, in degrees

Type float

angle_rad
rotation angle, in radians

Type float

area
area of ellipse, $A = \pi ab$

Type float

aspect_ratio
ratio of x-axis length to y-axis length

Type float

axes
list of semi-axes.

Type tuple

bound_max
maximum bounding circle of ellipse, (x, y, r)

Type tuple

bound_min
minimum interior circle of ellipse, (x, y, r)

Type tuple

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

Type list

matrix
rotation matrix

Type numpy.ndarray

n_dim
number of dimensions, 2

Type int

orientation
rotation matrix

Type numpy.ndarray

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

Type list

size
diameter of equivalent area circle

Type float

volume
alias for area

Type float

microstructpy.geometry.ellipsoid module

class microstructpy.geometry.ellipsoid.**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

- **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** (*3x3 numpy.ndarray, optional*) – A 3x3 rotation matrix expressing the orientation of the ellipsoid. Defaults to the identity.
- **position** – Alias for *center*.
- **a** – Alias for *axes[0]*.
- **b** – Alias for *axes[1]*.
- **c** – Alias for *axes[2]*.
- **orientation** – 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*) – Center position of the first sphere.

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 Axes3D.plot_surface method to add an ellipsoid to the current axes. The keyword arguments are passed 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*) – Points to reflect.

Returns Reflected points.

Return type numpy.ndarray

[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)

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 scipy.stats distributions.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi ABC] \\ &= \frac{4}{3}\pi\mathbb{E}[A]\mathbb{E}[B]\mathbb{E}[C] \\ &= \frac{4}{3}\pi\mu_A\mu_B\mu_C\end{aligned} \tag{3.21}$$

If the ellisoid is specified by size and aspect ratios, then the expected volume is computed by:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{\pi}{6}S^3] \\ &= \frac{\pi}{6}(\mu_S^3 + 3\mu_S\sigma_S^2 + \gamma_1\sigma_S^3)\end{aligned} \tag{3.24}$$

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.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*) – Point or list of points.

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

axes

the 3 semi-axes of the ellipsoid

Type tuple

bound_max

maximum bounding sphere, (x, y, z, r)

Type tuple

bound_min

minimum interior sphere, (x, y, z, r)

Type tuple

coefficients
coefffificients 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

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

Type list

matrix
A 3x3 rotation matrix

Type numpy.ndarray

matrix_quadeq
Matrix of the quadratic equation

Type numpy.ndarray

matrix_quadform
Matrix of the quadratic form

Type numpy.ndarray

n_dim
number of dimensions, 3

Type int

orientation
A 3x3 rotation matrix

Type numpy.ndarray

ratio_ab
ratio of x-axis length to y-axis length

Type float

ratio_ac
ratio of x-axis length to z-axis length

Type float

ratio_ba
ratio of y-axis length to x-axis length

Type float

ratio_bc
ratio of y-axis length to z-axis length

Type float

ratio_ca
ratio of z-axis length to x-axis length

Type float

ratio_cb
ratio of z-axis length to y-axis length

Type float

rot_seq_deg
rotation sequence, with angles in degrees

Type list

rot_seq_rad
rotation sequence, with angles in radiands

Type list

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

Type list

size
diameter of equivalent volume sphere

Type float

volume
volume of ellipsoid, $V = \frac{4}{3}\pi abc$

Type float

microstructpy.geometry.n_box module

N-Dimensional Box

This module contains the NBox class.

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** (*nxn list, optional*) – Rotation matrix

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*) – Point or list of points.

Returns Flags set to True for points in geometry.

Return type bool or numpy.ndarray

bounds
(lower, upper) bounds of the box

Type float

corner
bottom-left corner

Type list

limits
(lower, upper) bounds of the box

Type list

n_vol
area, volume of n-box

Type float

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

Type list

microstructpy.geometry.n_sphere module

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 *2r*.
- **diameter** – Alias for *2r*.
- **size** – Alias for *2r*.
- **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]

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

Returns Set to True for points in geometry.

Return type bool or numpy.ndarray

bound_max
maximum bounding n-sphere

Type tuple

bound_min
minimum interior n-sphere

Type tuple

d
diameter of n-sphere.

Type float

diameter
diameter of n-sphere.

Type float

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

Type list

position
position of n-sphere.

Type list

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

radius
radius of n-sphere.

Type float

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

Type list

size
size (diameter) of n-sphere.

Type float

microstructpy.geometry.rectangle module

class microstructpy.geometry.rectangle.**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.

Parameters

- **length** (*float, optional*) – Length of the rectangle.
- **width** (*float, optional*) – Width of the rectangle.
- **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** (*2x2 array, optional*) – The rotation matrix.

approximate(*x1=None*)

classmethod area_expectation(***kwargs*)

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 *microstructpy.geometry.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.

angle

angle_deg

angle_rad

area
area of rectangle

Type float

length

n_dim
number of dimensions, 2

Type int

width

class microstructpy.geometry.rectangle.**Square**(***kwargs*)
Bases: *microstructpy.geometry.rectangle.Rectangle*

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.

classmethod area_expectation(***kwargs*)

side_length
length of the side of the square.

Type float

microstructpy.geometry.sphere module

Sphere

This module contains the Sphere class.

class microstructpy.geometry.sphere.**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.

Parameters

- **r** (*float, optional*) – The radius of the n-sphere.
- **radius** (*float, optional*) – The radius of the n-sphere.
- **d** (*float, optional*) – The diameter of the n-sphere.
- **diameter** (*float, optional*) – The diameter of the n-sphere.
- **size** (*float, optional*) – The size of the n-sphere.
- **center** (*list, float, numpy.ndarray*) – The coordinates of the center.
- **position** (*list, float, numpy.ndarray*) – The coordinates of the center.

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

This function uses the 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.

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 __init__ function. The values for these keywords can be either constants or scip.stats distributions.

The expected value is computed by the following formula:

$$\begin{aligned}\mathbb{E}[V] &= \mathbb{E}[\frac{4}{3}\pi R^3] \\ &= \frac{4}{3}\pi\mathbb{E}[R^3] \\ &= \frac{4}{3}\pi(\mu_R^3 + 3\mu_R\sigma_R^2 + \gamma_1\sigma_R^3)\end{aligned} \tag{3.26}$$

Parameters ****kwargs** – Keyword arguments, see *microstructpy.geometry.Sphere*.

Returns Expected value of the volume of the sphere.

Return type float

n_dim
number of dimensions, 3

Type int

volume
volume of sphere

Type float

microstructpy.meshing package

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** (*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, optional*) – The seed number associated with each region.
- **phase_numbers** (*list, optional*) – The phase number associated with each region.
- **facet_neighbors** (*list, optional*) – The region numbers on either side of each facet
- **volumes** (*list, optional*) – The area/volume of each region.

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='str'` 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*)

Create from *SeedList* and `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 by reflecting seeds across their boundaries and meshing a larger domain. The reflection process will double the number of points in the voro++ input, which may cause a noticable slow-down for high-population microstructures. This reflection process is unnecessary for rectangular and box domains.

Parameters

- **seedlist** (*SeedList*) – A list of seeds in the microstructure.
- **domain** (*Domain*) – The domain to be filled by the seeds.

Returns A polygon/polyhedron mesh.

Return type *PolyMesh*

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

This function plots the polygon mesh. The keyword arguments are passed though to matplotlib.

Parameters ****kwargs** – Keyword arguments for matplotlib.

plot_facets(***kwargs*)
Plot PolyMesh facets.

This function plots the facets of the polygon mesh, rather than the regions.

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

write(*filename*, *format='txt'*)
Write the mesh to a fileself.

This function writes the polygon/polyhedron mesh to a file. The format of the file can be specified, with the options described in the table below.

Table 1: PolyMesh Write Formats

Format	format	kD	Description
Text String	txt	ND	The Python string representation of the mesh. Human readable, but not in a standard file format.
POLY File	poly	2D	A poly file that contains a planar straight line graph (PSLG). This file can be read by the Triangle program from J. Shewchuk.
PLY File	ply	ND	A PLY file containing the mesh facets.
VTK Legacy	vtk	3D	A VTK file containing the mesh as a POLYDATA dataset. Note: seed number and phase number information are not written to the VTK file.

The text string output file is meant solely for saving the polygon/ polyhedron mesh as an intermediate step in the meshing process. The other file types lists are meant for processing and interpretation by other programs. The format for the text string file is:

```
Mesh Points: <numPoints>
    x1, y1(, z1)      <- tab character 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
```

(continues on next page)

(continued from previous page)

```
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
    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
```

Note that everything is indexed from 0 since this is produced in Python. 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.

See also:

.poly files Description and examples of poly files.

PLY - Polygon File Format Description and examples of ply files.

File Formats for VTK Version 4.2 PDF guide for VTK legacy format.

Parameters

- **filename** (*str*) – Name of the file to be written.
- **format** (*str*) – Format of the data in the file.

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 or numpy.ndarray*) – List of coordinates in the mesh.
- **elements** (*list or 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 or numpy.ndarray, optional*) – A number associated with each element.
- **facets** (*list or numy.ndarray, optional*) – A list of facets in the mesh. The shape should be Nx2 in 2D or Nx3 in 3D.
- **facet_attributes** (*list or numpy.ndarray, optional*) – A number associated with each facet.

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*, *min_angle=0*, *max_volume=inf*, *max_edge_length=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 several values, described below.

Table 2: Phase Type Options

Value	Description
crystalline, granular, solid	These options all create a mesh where cells of the same seed number are merged, but cells are not merged across seeds. (default)
amorphous, glass, matrix	These options create a mesh where cells of the same phase number are merged, creating an amorphous region in the mesh.
crack, hole, void	These options are the same as the amorphous options described above, except that these regions are treated as holes in the mesh.

Parameters

- **polymesh** (PolyMesh) – A polygon/polyhedron mesh.
- **phases** (*list*) – A list of dictionaries containing options for each phase.
- **min_angle** (*float*) – The minimum interior angle of an element.
- **max_volume** (*float*) – The default maximum cell volume, used if one is not set for each phase.
- **max_edge_length** (*float*) – The maximum edge length of elements along grain boundaries. Currently only supported in 2D.

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

This method plots the mesh using matplotlib. In 2D, the elements are plotted using a matplotlib PolyCollection. In 3D, the facets are plotted using a Poly3DCollection.

Parameters ****kwargs** – Keyword arguments that are passed through to matplotlib.

write (*filename*, *format='str'*, *seeds=None*, *polymesh=None*)
Write mesh to file.

This function writes the contents of the mesh to a file. There are some options for the format of the file, described below.

Table 3: TriMesh Write Formats

Format	format	kD	Description
Abaqus	abaqus	ND	An Abaqus input file.
Text String	str	ND	The results of __str__
TetGen / Triangle	tet / tri	ND	Node and element files formatted according to the TetGen and Triangle file standard. The filename input above should be the basename *without* extensions. The extensions .node and .ele will be added. See the links below for more information about the file formats.
VTK Legacy	vtk	3D	A VTK file

See also:

TetGen file formats

Triangle file formats

Triangle examples

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*) – The format of the output.
- **seeds** (SeedList) – List of seeds. If given, will also write phase number to VTK files. This assumes the *element_attributes* field contains the seed number of each element.
- **polymesh** (PolyMesh) – Polygonal mesh used for generating the the triangular mesh. If given, will add surface unions to Abaqus files - for easier specification of boundary conditions.

Submodules

microstructpy.meshing.polymesh module

Polygon Meshing

This module contains the class definition for the PolyMesh class.

class microstructpy.meshing.polymesh.**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** (*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, optional*) – The seed number associated with each region.
- **phase_numbers** (*list, optional*) – The phase number associated with each region.
- **facet_neighbors** (*list, optional*) – The region numbers on either side of each facet
- **volumes** (*list, optional*) – The area/volume of each region.

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='str'` 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*)
Create from *SeedList* and 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 by reflecting seeds across their boundaries and meshing a larger domain. The reflection process will double the number of points in the voro++ input, which may cause a noticable slow-down for high-population microstructures. This reflection process is unnecessary for rectangular and box domains.

Parameters

- **seedlist** (SeedList) – A list of seeds in the microstructure.
- **domain** (*Domain*) – The domain to be filled by the seeds.

Returns A polygon/polyhedron mesh.

Return type *PolyMesh*

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

This function plots the polygon mesh. The keyword arguments are passed though to matplotlib.

Parameters ****kwargs** – Keyword arguments for matplotlib.

plot_facets(***kwargs*)
Plot PolyMesh facets.

This function plots the facets of the polygon mesh, rather than the regions.

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

write(*filename*, *format='txt'*)
Write the mesh to a fileself.

This function writes the polygon/polyhedron mesh to a file. The format of the file can be specified, with the options described in the table below.

Table 4: PolyMesh Write Formats

Format	format	kD	Description
Text String	txt	ND	The Python string representation of the mesh. Human readable, but not in a standard file format.
POLY File	poly	2D	A poly file that contains a planar straight line graph (PSLG). This file can be read by the Triangle program from J. Shewchuk.
PLY File	ply	ND	A PLY file containing the mesh facets.
VTK Legacy	vtk	3D	A VTK file containing the mesh as a POLYDATA dataset. Note: seed number and phase number information are not written to the VTK file.

The text string output file is meant solely for saving the polygon/ polyhedron mesh as an intermediate step in the meshing process. The other file types lists are meant for processing and interpretation by other programs. The format for the text string file is:

```
Mesh Points: <numPoints>
    x1, y1(, z1)      <- tab character at line start
    x2, y2(, z2)
```

(continues on next page)

(continued from previous page)

```
    ...
    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
    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
```

Note that everything is indexed from 0 since this is produced in Python. 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.

See also:

.poly files Description and examples of poly files.

PLY - Polygon File Format Description and examples of ply files.

File Formats for VTK Version 4.2 PDF guide for VTK legacy format.

Parameters

- **filename** (*str*) – Name of the file to be written.
- **format** (*str*) – Format of the data in the file.

microstructpy.meshing.trimesh module

Triangle/Tetrahedron Meshing

This module contains the class definition for the TriMesh class.

class microstructpy.meshing.trimesh.**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 or numpy.ndarray*) – List of coordinates in the mesh.
- **elements** (*list or 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 or numpy.ndarray, optional*) – A number associated with each element.
- **facets** (*list or numy.ndarray, optional*) – A list of facets in the mesh. The shape should be Nx2 in 2D or Nx3 in 3D.
- **facet_attributes** (*list or numpy.ndarray, optional*) – A number associated with each facet.

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*, *min_angle=0*, *max_volume=inf*, *max_edge_length=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 several values, described below.

Table 5: Phase Type Options

Value	Description
crystalline, granular, solid	These options all create a mesh where cells of the same seed number are merged, but cells are not merged across seeds. (default)
amorphous, glass, matrix	These options create a mesh where cells of the same phase number are merged, creating an amorphous region in the mesh.
crack, hole, void	These options are the same as the amorphous options described above, except that these regions are treated as holes in the mesh.

Parameters

- **polymesh** (PolyMesh) – A polygon/polyhedron mesh.
- **phases** (*list*) – A list of dictionaries containing options for each phase.
- **min_angle** (*float*) – The minimum interior angle of an element.
- **max_volume** (*float*) – The default maximum cell volume, used if one is not set for each phase.
- **max_edge_length** (*float*) – The maximum edge length of elements along grain boundaries. Currently only supported in 2D.

plot (***kwargs*)

Plot the mesh.

This method plots the mesh using matplotlib. In 2D, the elements are plotted using a matplotlib PolyCollection. In 3D, the facets are plotted using a Poly3DCollection.

Parameters ****kwargs** – Keyword arguments that are passed through to matplotlib.

write (*filename*, *format='str'*, *seeds=None*, *polymesh=None*)

Write mesh to file.

This function writes the contents of the mesh to a file. There are some options for the format of the file, described below.

Table 6: TriMesh Write Formats

Format	format	kD	Description
Abaqus	abaqus	ND	An Abaqus input file.
Text String	str	ND	The results of __str__
TetGen / Triangle	tet / tri	ND	Node and element files formatted according to the TetGen and Triangle file standard. The filename input above should be the basename *without* extensions. The extensions .node and .ele will be added. See the links below for more information about the file formats.
VTK Legacy	vtk	3D	A VTK file

See also:

TetGen file formats

Triangle file formats

Triangle examples

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*) – The format of the output.
- **seeds** (SeedList) – List of seeds. If given, will also write phase number to VTK files. This assumes the *element_attributes* field contains the seed number of each element.
- **polymesh** (PolyMesh) – Polygonal mesh used for generating the the triangular mesh. If given, will add surface unions to Abaqus files - for easier specification of boundary conditions.

microstructpy.seeding package

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 (*microstructpy.geometry*), phase number, breakdown, and position.

Parameters

- **seed_geometry** (from *microstructpy.geometry*) – The geometry of the seed.
- **phase** (*int*) – The phase number of the seed.
- **breakdown** (*list or numpy.ndarray*) – The circle/sphere approximation of this geometry. 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*) – The coordinates of the seed. See *position* for more details.

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** (*optional, int*) – Material phase number of seed.
- **breakdown** (*optional, list*) – 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** (*optional, list*) – 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.

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 PatchCollection to the current axes.

Parameters ****kwargs** – Matplotlib keyword arguments.

limits
The (lower, upper) bounds of the seed

Type list

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.

volume
The area (2D) or volume (3D) of the seed

Type float

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*) – 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*) – Dictionary of RNG seeds for each step in the seeding process.

Returns An instance of the class containing seeds prescribed by the phase information and filling the given volume.

Return type *SeedList*

plot(***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 ****kwargs** – Keyword arguments to pass to matplotlib

plot_breakdown(***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 ****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 package*.

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 on the scipy.stats website, in the multivariate distributions section.

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 the *microstructpy.geometry package*) – The domain of the microstructure.
- **pos_dists** (*dict, optional*) – Position distributions for each phase, formatted like the example above.
- **rng_seed** (*int, optional*) – Random number generator (RNG) seed for positioning the seeds. Should be a non-negative integer.
- **hold** (*list, optional*) – List of booleans for holding the positions of seeds.
- **max_attempts** (*int, optional*) – Number of random trials before removing a seed from the list.
- **rtol** (*'fit' or float*) – The relative overlap tolerance between seeds. This parameter should be between 0 and 1. Using the 'fit' option, the function will pick a value for rtol based on the mean and standard deviation in seed volumes.
- **verbose** (*bool*) – This option will print a running counter of how many seeds have been positioned.

write(*filename*)

Write seed list to a text file

This function writes out the seed list to a file. The content of this file is human-readable and can be read by the from_file method.

Parameters **filename** (*str*) – File to write the seed list.

Submodules

microstructpy.seeding.seed module

Seed

This module contains the class definition for the Seed class.

class microstructpy.seeding.seed.**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 (*microstructpy.geometry*), phase number, breakdown, and position.

Parameters

- **seed_geometry** (from *microstructpy.geometry*) – The geometry of the seed.
- **phase** (*int*) – The phase number of the seed.
- **breakdown** (*list or numpy.ndarray*) – The circle/sphere approximation of this geometry. 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*) – The coordinates of the seed. See *position* for more details.

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** (*optional, int*) – Material phase number of seed.
- **breakdown** (*optional, list*) – 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** (*optional, list*) – 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.

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 PatchCollection to the current axes.

Parameters ****kwargs** – Matplotlib keyword arguments.

limits
The (lower, upper) bounds of the seed

Type list

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.

---

volume
The area (2D) or volume (3D) of the seed

Type float

microstructpy.seeding.seedlist module

Seed List

This module contains the class definition for the SeedList class.

class microstructpy.seeding.seedlist.**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*) – 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*) – Dictionary of RNG seeds for each step in the seeding process.

Returns An instance of the class containing seeds prescribed by the phase information and filling the given volume.

Return type *SeedList*

plot(***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 ****kwargs** – Keyword arguments to pass to matplotlib

plot_breakdown(***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 ****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 package*.

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 on the scipy.stats website, in the multivariate distributions section.

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 the *microstructpy.geometry package*) – The domain of the microstructure.
- **pos_dists** (*dict, optional*) – Position distributions for each phase, formatted like the example above.
- **rng_seed** (*int, optional*) – Random number generator (RNG) seed for positioning the seeds. Should be a non-negative integer.
- **hold** (*list, optional*) – List of booleans for holding the positions of seeds.
- **max_attempts** (*int, optional*) – Number of random trials before removing a seed from the list.

- **rtol** (*'fit' or float*) – The relative overlap tolerance between seeds. This parameter should be between 0 and 1. Using the 'fit' option, the function will pick a value for rtol based on the mean and standard deviation in seed volumes.
- **verbose** (*bool*) – This option will print a running counter of how many seeds have been positioned.

write (*filename*)
Write seed list to a text file

This function writes out the seed list to a file. The content of this file is human-readable and can be read by the from_file method.

Parameters **filename** (*str*) – File to write the seed list.

3.6.2 Submodules

microstructpy.cli module

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**(*raw_in*, *filepath='.'*)
Convert dictionary from xmltodict

This function converts the dictionary created by xmltodict. The input is an ordered dictionary, where the keys are strings and the items are either strings, lists, or ordered dictionaries. Strings occur are the "leaves" of the dictionary and are converted into values using `microstructpy._misc.from_str()`. Lists are return with each of their elements converted into values. Ordered dictionaries are converted by (recursively) calling this function.

Parameters

- **raw_in** – unconverted input- either dict, list, or str
- **filepath** (*str, optional*) – filepath of input XML, to resolve relative paths in the input file

Returns A copy of the input where the strings have been converted

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

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

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

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

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

microstructpy.cli.**run**(*phases*, *domain*, *verbose=False*, *restart=True*, *directory='.'*, *filetypes={}*, *rng_seeds={}*, *plot_axes=True*, *rtol='fit'*, *mesh_max_volume=inf*, *mesh_min_angle=0*, *mesh_max_edge_length=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** (*dict or list*) – A dictionary or list of dictionaries for each material phase. The dictionary entries depend on the shape of the seeds, but one example is:

```
phase1 = {'name': 'foam',
          'material_type': 'amorphous',
          'volume': 0.25,
          'shape': 'sphere',
          'd': scipy.stats.lognorm(s=0.4, scale=1)
          }

phase2 = {'name': 'voids',
           'material_type': 'void',
           'volume': 0.75,
           'shape': 'sphere',
           'r': 1
           }

phases = [phase1, phase2]
```

The entries can be either constants (`'r': 1`) or distributed, (`'d': scipy.stats.lognorm(s=0.4, scale=1)`).

The entries can be either constants (`'radius': 1`) or distributed, (`'diameter': scipy.stats.lognorm(s=0.4, scale=0.5)`). The following non-shape keywords can be used in each phase:

Table 7: Non-Shape Phase Keywords

Keyword	Default	Notes
color	C<n>	Can be any matplotlib color. Defaults to the standard matplotlib color cycle. More info on the matplotlib Specifying Colors page.
material_type	crystalline	Options: **crystalline**, granular, solid, **amorphous**, glass matrix, **void**, crack, hole. (Non-bolded words are alies for the bolded words.)
name	Material <n>	Can be non-string variable.
position	*uniform*	Uniform random distribution, see below for options.
fraction	1	Can be proportional volumes (such as 1:3) or fractions (such as 0.1, 0.2, and 0.7). Can also be a scipy.stats distribution. Volume fractions are normalized by their sum.

The position distribution of the phase can be customized for non-randomly sorted phases. For example:

```
# independent distributions for each axis
position = [0,
           scipy.stats.uniform(0, 1),
           scipy.stats.uniform(0.25, 0.5)]

# correlated position distributions
mu = [2, 3]
sigma = [[3, 1], [1, 4]]
position = scipy.stats.multivariate_normal(mu, sigma)
```

- **domain** (class from *microstructpy.geometry*) – The geometry of the domain.
- **verbose** (*bool*) – Option to run in verbose mode. Prints status updates to the terminal. Defaults to False.
- **restart** (*bool*) – 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*) – 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*) – 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*) – The random number generator (RNG) seeds. The dictionary values should all be non-negative integers. 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*) – 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.

- **mesh_max_volume** (*float*) – 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. Should be stritly positive.

- **mesh_min_angle** (*float*) – 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. Should be in the range 0-60.
- **mesh_max_edge_length** (*float*) – The maximum edge length of elements along grain boundaries. Currently only supported in 2D.
- **plot_axes** (*bool*) – 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*) – Option to verify the output mesh against the input phases.
- **color_by** (*str*) – Method for coloring seeds and grains in the output plots. The options are {'material', 'seed number', 'material number'}. For 'material', the color field of each phase is used. For 'seed number' and 'material number', the seeds are colored using the colormap specified in the 'colormap' keyword argument.
- **colormap** (*str*) – Name of the colormap used to color the seeds and grains if 'color_by' is set to 'seed number' or 'material number'. A full explanation of the matplotlib colormaps is availabe at Choosing Colormaps in Matplotlib.
- **seeds_kwargs** (*dict*) – Optional keyword arguments for plotting seeds. For example, the line width and color.
- **poly_kwargs** (*dict*) – Optional keyword arguments for plotting polygonal meshes. For example, the line width and color.
- **tri_kwargs** (*dict*) – Optional keyword arguments for plotting triangular meshes. For example, the line width and color.

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

microstructpy.cli.**unpositioned_seeds**(*phases*, *domain*, *rng_seeds={}*)

microstructpy.verification module

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 of dicts*) – List of input phases.
- **poly_mesh** (PolyMesh) – Polygonal/polyhedral mesh.

Returns

list, same size and dictionary keywords as phases, but with error statistics dictionaries in each entry.

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.

Parameters

- **seeds** (SeedList) – List of seeds.
- **phases** (*list of dicts*) – List of input phases.
- **poly_mesh** (PolyMesh) – Polygonal/polyhedral mesh.

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*) – Plot output directory. Defaults to ..
- **ext** (*str or list of strs*) – File extension(s) of the output plots.
- **poly_mesh** (PolyMesh) – 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 of strs*) – 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*)

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

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 of dicts*) – List of input phase dictionaries.
- **out_phases** (*list of dicts*) – List of output phase dictionaries.
- **filename** (*str*) – Filename of the output table.

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*) – Name of file to write, defaults to *volume_fractions.txt*.

Returns none, prints formatted volume fraction verification table to file

Python Module Index

m

Index

A

L

M

N

O

P

R

S

T

U

V

W