ARL-TN-0806 • Dec 2016

US Army Research Laboratory

Voronoi-Based Nanocrystalline Generation Algorithm for Atomistic Simulations

by Daniel Foley, Shawn P Coleman, Garritt Tucker, and Mark A Tschopp

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-0806 ● DEC 2016

US Army Research Laboratory

Voronoi-Based Nanocrystalline Generation Algorithm for Atomistic Simulations

by Daniel Foley
Oak Ridge Institute for Science and Engineering (ORISE), Belcamp, MD

Shawn P Coleman and Mark A Tschopp
Weapons and Materials Research Directorate, ARL

Garritt Tucker
Drexel University, Philadelphia, PA

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2016	Technical Note	May 2015–August 2016

4. TITLE AND SUBTITLE
Voronoi-Based Nanocrystalline Generation Algorithm for Atomistic Simulations

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Daniel Foley, Shawn P Coleman, Garritt Tucker, and Mark A Tschopp

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Research Laboratory
ATTN: RDRL-WMM-F
Aberdeen Proving Ground, MD 21005-5069

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-0806

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Email: mark.a.tschopp.civ@mail.mil

14. ABSTRACT

The objective herein is to discuss an algorithm for generating nanocrystalline structures for use in molecular dynamics simulations. This algorithm employs Voronoi tessellations to populate grains using atomic coordinates obtained from a set of reference structures. To ensure a realistic grain morphology, a shell-based, geometric packing algorithm is used to assign grain centers. Optimized placement of centers represents an improvement over traditional Voronoi methods that rely on random placement by ensuring a consistent grain size and realistic grain geometry. Careful tracking of the grain centers and the atomic structure within grains allows the user to rescale a previously built nanostructure, resulting in larger or smaller grains with consistent grain morphology. This algorithm can be used to generate a wide variety of granular nanostructures including nano-twinned metals, alloyed nanocrystals, and multiphase nanocrystals.

15. SUBJECT TERMS
US Army Research Laboratory, ARL, nanocrystalline materials; molecular dynamics; twinning; grain size

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Mark A Tschopp
Unclassified	Unclassified	Unclassified	UU	44	19b. TELEPHONE NUMBER (Include area code) 410-306-0855

Contents

List of Figures

Acknowledgments

The authors would like to acknowledge funding from High Performance Computing Modernization Program DOD Next Generation Workforce Development, HIP-15-020 and HIP-16-005. This research was supported in part by an appointment to the Graduate Research Participation Program at the US Army Research Laboratory (ARL) administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the US Department of Energy and ARL.

1. Introduction

One of the major challenges facing researchers studying nanoscale systems is identifying the dynamic mechanisms controlling material properties and performance. This is especially apparent in nanocrystalline metals where mechanical properties are largely governed by atomic scale defects and interfaces between crystalline regions. Molecular dynamics simulations provide researchers with a tool to accurately simulate such small systems with atomic resolution, but these simulations lack an efficient means of producing the complex structures associated with nanograined systems. These limitations mean that researchers must generate nanocrystalline structures manually with an external algorithm.

In this technical note, an algorithm (*nanocrystal_builder.py*) implemented in Python is used to efficiently generate highly tailored nanocrystalline microstructures based on user input. The Python script is attached as the Appendix and a description of its capabilities and execution is described within this document.

2. Background

2.1 Voronoi Tessellation

The nanocrystalline builder algorithm is based on a Voronoi tessellation method that implements packing rules to create optimal grain morphologies. Voronoi tessellations, also called Voronoi diagrams, are powerful space-filling geometric structures that have many applications in modern science. In this work, center of mass Voronoi tessellations are used to construct 3-dimensional polyhedra, which define the limits of individual grains in a polycrystal. In center of mass Voronoi tessellations, a set of points, $\mathbf{p}$, are defined as local sources of influence (centers of mass) for a spherical field $\mathbf{q}$. Individual points $\mathbf{q}_j$ are assigned to centers of influence $\mathbf{p}_i$ based on their proximity. Voronoi regions are then defined as the collection of all points that are assigned to a single region of influence. Figure 1 showcases a simple 2-dimensional case with 25 centers of mass, where the Voronoi regions are bounded by the center lines separating Voronoi centers.

Traditionally, Voronoi tessellation algorithms are initialized with either randomly placed center of masses or centers packed using a regular grid; however, these are not ideal for constructing nanocrystals. The disadvantage with implementing randomly dispersed Voronoi tessellation algorithms for nanocrystalline construction is

that if the Voronoi (grain) centers are not properly packed, the resulting structure will likely be unrealistic. Purely random placement of centers can result in acute grains and needle-like grain structures that are rarely, if ever, observed in physical systems. Similarly, grid packing can result in highly cubic grains or idealized grains that are equally unlikely. Therefore, in this work, a spherically optimized packing routine was implemented to overcome the hurdles presented by these traditional packing routines to create more realistic nanocrystalline starting structures.

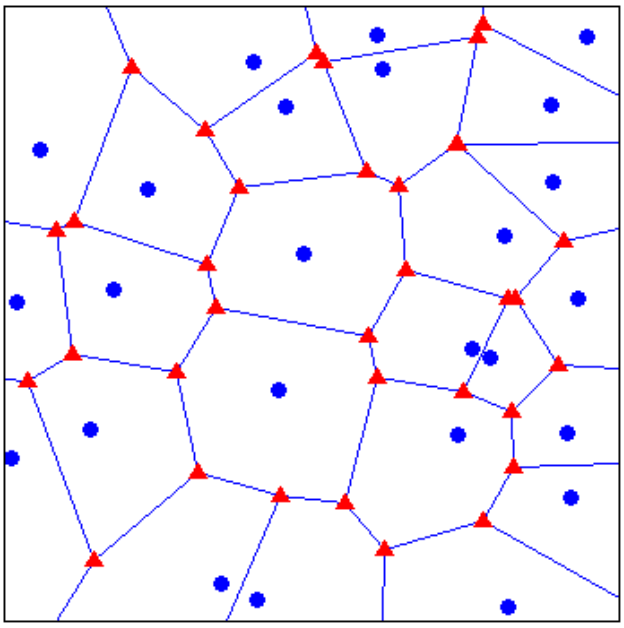

Fig. 1 A Voronoi diagram showing the tessellation for 25 randomly generated centers of mass. The centers are blue circles. The blue lines are equal distance to the 2 closest center points. The red triangles are vertices points at equal distance to 3 center points.

2.2 Nearest-Neighbor Analysis

One obstacle to using Voronoi tessellations for creating atomic nanocrystalline structures is the computationally inefficient process of testing for nearest neighbors when assigning grain centers and populating/trimming atoms. The brute force method loops over each point and repeatedly calculates the distances to all other points (incorporating periodic boundaries) to define nearest neighbors. While this method is easy to implement, such brute force methods waste computational resources by running calculations on nonlocal points that could be ignored. It is possible to include rules for ignoring points far away, which can help this brute force approach to be slightly more efficient. However, since *nanocrystal_builder.py*

generates potentially millions of data points (atoms), a more efficient method for proximity testing was necessary. Typically, high (computational) efficiency nearest-neighbor analysis is performed by breaking the data set down into a tiered data structure that spatially sorts the atom coordinates into bins. This allows these high-efficient algorithms to limit the number of points queried to just those belonging to the same tier (or neighboring tiers) as the point of interest. An example of such a data structure, the KDTree from SciPy stack in Python, is leveraged in this work to improve the computational efficiency of the nearest-neighbor testing routines.

3. Methodology

The following sections provide a detailed description of how *nanocrystal_builder.py* works. The source code has been broken apart into several subfunctions to help improve readability and to establish a modular flow control. Each function in the code has been designed to perform a specific set of tasks that will be described separately.

In general, there are 2 ways to construct nanocrystals with *nanocrystal_builder.py*. The first method uses an optimized packing algorithm to generate a list of grain centers that are populated with seeds—spherical groups of atoms extracted from a reference file. This method uses a single reference file, and is therefore referred to as Single Mode. The second method, referred to as Config Mode, uses data from a configuration file to generate nanocrystalline structures thus providing additional control over the structure. Configuration files contain information pertaining to the simulation cell size, average grain size, as well as the position, orientation and reference data for each grain. Configuration files are automatically generated by *nanocrystal_builder.py* as a standard output to ensure the resulting structure is easily reproducible. Users can modify configuration files to obtain a high degree of control over the resulting structure, allowing for the construction of networked grains and rescaled grain sizes.

3.1 Required Software

The *nanocrystal_builder.py* algorithm detailed in this document is written in Python and makes use of the SciPy stack. Development and testing were performed using Python 2.6.6 and the SciPy 0.15.0 with its associated libraries. These version numbers should be considered the minimum required for use. The standard output for the atomic data generated by *nanocrystal_builder.py* is formatted to match the data file format associated with the molecular dynamics code LAMMPS.[1,2] Therefore, structures can be read into LAMMPS using the *read_data* command (i.e., http://lammps.sandia.gov/doc/read_data.html).

3.2 Single Mode (Generating from a LAMMPS Dump File)

This section discusses the methods and code used to generate a nanocrystalline structure with a single reference file for seed extraction. Some of the code segments detailed here are also used in Config Mode as discussed in Section 3.3. It is recommended that the reference structures used for Single Mode be of single crystalline or bicrystalline nature as more complex structures may result in unrealistic structures. Additional care must be taken when using a bicrystalline reference structure as the existence of a grain boundary within the seed effectively reduces the grain size and may create potentially unrealistic nanocrystalline structures.

3.2.1 Wrapping Periodic Bounds

The *nanocrystal_builder.py* script accommodates user-defined periodic boundaries. This task is performed by testing if a point, which can be an atom or a grain center, is positioned beyond the upper and lower bounds of the simulation cell for each dimension. If one of the boundaries is exceeded, the point is conditionally moved by adding (or subtracting) a periodic length if the point lies outside of the lower (or upper) bound. Performing this process in all 3 axes results in all points being contained within the simulation cell.

3.2.2 Optimized Packing of Grain Centers

The *nanocrystal_builder.py* script uses Voronoi tessellations to form 3-dimensional, grain-like structures. In this routine an initial center is placed at the corner of the simulation cell lying on the origin. A cloud of allowed points is then defined at a minimum and maximum radius around this center; these radii are defined to be 75% and 125% of the requested grain size, respectively. The next grain center is chosen

at random from these points and a new cloud of allowed points is defined such that none of the new points fall within the minimum radius of any existing grain center. This method is repeated until an iteration is reached in which the number of allowed points becomes zero, thereby optimally filling the requested simulation cell size. To account for periodic boundary conditions, "ghost" grain centers are placed in the area outside the simulation cell which represent the reflections, across the periodic bounds, of previously defined centers. This ensures that centers near the simulation edge are not within the minimum radius of each other when wrapped around the periodic boundaries.

3.2.3 Generating and Scaling Reference Files

The *nanocrystal_builder.py* script uses reference files constructed in the LAMMPS dump file format. In the event that the reference file provided by the user is smaller than the desired grain size, the *nanocrystal_builder.py* code enters into a dynamic rescaling function. This function checks the dimensions of the reference file and replicates the reference atoms across the periodic edge. This is repeated until all of the reference file dimensions are at least 150% of the requested grain size. In Single Mode this is done once and the newly replicated reference is held in memory until the code exits. In Config Mode the reference structure for each grain is individually rescaled and dropped from memory at the end of each iteration of the code.

3.2.4 Populating Atoms

The *nanocrystal_builder.py* script then populates each Voronoi grain with atoms from the rescaled reference file. Figure 2 highlights the steps that are used. Images generated using OVITO.[3] The first step in this process is to isolate a sphere of atoms from the reference file. This is done by searching for all atoms within a cutoff radius of the reference structure center point using a KDTree query. By default the sphere of atoms is rotated randomly using the axis-angle method. However, if a texturing axis has been specified, the rotation axis is fixed to the texturing axis. The rotated sphere of atoms is then assigned to a grain center by placing the center point of the sphere on the grain center. This process is repeated until all of the grain centers are populated.

At this point, the structure is populated with overlapping spheres of atoms. These spheres are trimmed according to the Voronoi tessellation of the simulation cell (i.e., the boundaries between grains are defined by the midlines between grain cen-

ters [see Fig. 1]). To determine if an atom lies within the boundaries of its parent grain, KDTree queries are used to test if the nearest grain center to the atom is the grain center it was assigned (its parent). If the atom is closest to a center that is not its parent center, it is deleted. To account for the periodic edges "ghost" centers are placed around the defined simulation cell. After overlapping atoms are removed, the structure is wrapped across the periodic bounds to ensure that all atoms are contained within the user-defined simulation cell as shown in Fig. 2.

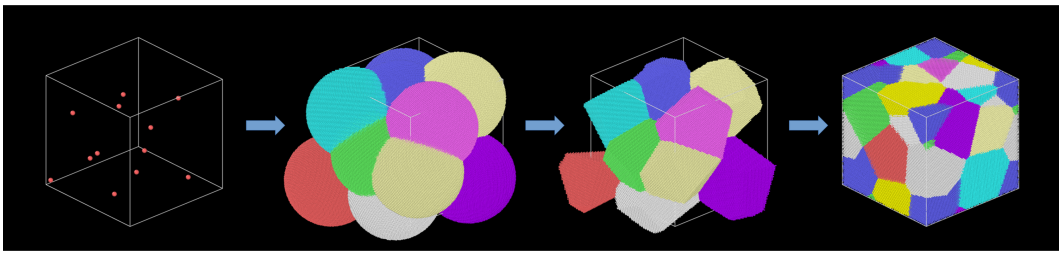

Fig. 2 Images showing steps taken when generating nanocrystals (left to right): populating cell with grain centers, sphere of atoms with defined crystal structure centered at each grain center, identifying atoms belonging to each grain center, removing overlapping atoms, and wrapping atoms through periodic boundaries back into the simulation cell

3.3 Config Mode: Generating from a Configuration File

The *nanocrystal_builder.py* script can also operate in Config Mode to generate nanocrystalline structures. In this mode, the nanocrystalline structure is generated using configuration files that contain the simulation cell dimensions, average grain size, grain positions, grain orientations, and reference data. Config Mode enables easy reproduction of previously built nanostructures and provides capabilities of rapidly modifying existing nanostructures. For example, Config Mode allows the user to automatically replace a percentage of the grain reference files, reorient grains to add texturing, and to rescale the entire system. These options are enabled using a series of questions posed to the user at the command prompt.

3.3.1 Reading Grain Information

The *nanocrystal_builder.py* script reads grain center information from a configuration file, prompts the user for modifications, and then builds the structure according. An example configuration file is shown in Fig. 3. The first line of the configuration file is for user comments about the structure, which is ignored by the code. The second line consists of 7 entries pertaining to the average grain size and simulation cell limits, specifically grain size (Å), X_{min}, X_{max}, Y_{min}, Y_{max}, Z_{min}, and Z_{max}. The

remaining lines contain information regarding individual grains in the following order: grain number, center x, y, z coordinate, rotation-axis x, y, z index, rotation angle, reference file. To improve efficiency, the grain data are stored as Python dictionary objects.

```
#data for centroids
50.000000 0.000000 100.000000 0.000000 100.000000 0.000000 100.000000
0 25.000000 25.000000 25.000000 0.577350 0.577350 0.577350 1.019442 out.crystcufix.txt
1 15.653842 14.213959 73.605896 0.577350 0.577350 0.577350 2.977697 out.twin1c.txt
2 10.845016 80.767845 21.027548 0.577350 0.577350 0.577350 0.997454 out.twin1c.txt
3 35.462112 77.228439 75.485779 0.577350 0.577350 0.577350 1.959592 out.crystcufix.txt
4 72.080017 5.680593 99.983498 0.577350 0.577350 0.577350 2.679047 out.twin1c.txt
5 48.061413 66.311081 38.980812 0.577350 0.577350 0.577350 1.239753 out.crystcufix.txt
6 77.514809 2.187867 55.050816 0.577350 0.577350 0.577350 2.508741 out.twin1c.txt
7 77.514809 45.743797 67.839989 0.577350 0.577350 0.577350 1.082618 out.crystcufix.txt
8 39.543736 37.302536 91.937027 0.577350 0.577350 0.577350 1.483817 out.crystcufix.txt
9 77.514809 45.743797 15.866305 0.577350 0.577350 0.577350 1.644024 out.twin1c.txt
10 90.813972 77.228439 83.444176 0.577350 0.577350 0.577350 1.879738 out.twin1c.txt
11 11.178849 49.204056 51.654900 0.577350 0.577350 0.577350 1.983781 out.crystcufix.txt
```

Fig. 3 An example of the contents of a config file: the first line is a comment line and is ignored by the algorithm, the second line contains information about the grain size and the simulation cell size (listed as grain size (Å), X_{min}, X_{max}, Y_{min}, Y_{max}, Z_{min}, and Z_{max}), and the remaining lines contain the grain center ID, atomic Cartesian coordinates, rotation axis, rotation angle (radians), and reference file

3.3.2 Repopulating Atoms

In Config Mode, the *nanocrystal_builder.py* script populates atoms using the information contained within the configuration file as well as any modifications specified by the user in the command prompt. First, the *nanocrystal_builder.py* script assesses any user-specified modifications. This starts by multiplying all lengths and center coordinates by the user-specified rescale factor. Once the data have been rescaled the code determines how many, if any, of the grains need to have their reference file replaced to satisfy the user-specified secondary reference composition. Finally, if the user has specified a texturing direction, the code will replace the rotation axis indices for all entries with those requested by the user.

Once the grain center data have been read and processed, the atom population subfunction commences in the same manner as Single Mode. However, as explained earlier, while in Config Mode the code reads and rescales the reference file at each step of the algorithm to accommodate the use of multiple references. This additional processing means that Config Mode runs slower than Single Mode, which performs the reference rescaling only once.

4. Usage

The following sections describe how to invoke and interact with the *nanocrystal_builder.py* script. These sections act as a general user guide to the *nanocrystal_builder.py* script.

4.1 Invoking the User Interface

The *nanocrystal_builder.py* script is written as a Python run script and can therefore be invoked through either a bash terminal or a Python interactive environment. Prior to invoking the script, the user must change the present working directory to the directory containing the *nanocrystal_builder.py* script. Once this is done the script is invoked by entering `python nanocrystal_builder.py` in terminal or `import nanocrystal_builder.py` in the Python interactive environment. Once invoked, the algorithm begins querying the user for necessary information.

The *nanocrystal_builder.py* script relies on user interaction to establish key initialization parameters within the algorithm. This interaction takes place through a series of simple text queries to the user within the run environment. The first query "single or config?" sets the variable environment and determines what set of queries the user is subsequently prompted to answer. This prompt only accepts user feedback of "single" or "config" (no quotes). Any other input will result in an error prompting the user to enter a valid response.

Single Mode. If the user responds to the first query with "single" the following prompts will be provided:

- "Input reference file path":

  The answer to this query tells the algorithm where the crystal reference file is located. The script expects the user to return a valid file path.

- "Input desired grain size in angstroms":

  User should respond with the desired average grain size, in angstroms, for their structure. Response should be an integer or float value.

- "Input desired box dimensions in angstroms.

  Use the following convention (xlo xhi ylo yhi zlo zhi)":

The user should provide the desired simulation cell bounds. Individual values should be in integer or float format with a single space in between values. Note that no parenthesis should be present in the response.

- "Input crystalline texturing direction

  Reply "none" (quotations excluded) if no texturing desired

  Use following convention "index_x, index_y, index_z":

  If the user desires their structure to possess crystalline texturing, they should respond to this query with the desired texturing direction using the Miller index convention. Individual indices should be integers with commas separating indices. If texturing is not desired, the user should respond with "none" (no quotes).

This mode outputs a configuration file, which can be used to recreate and manipulate the structure through Config Mode, and a LAMMPS-style data file that contains the atom types and positions.

Config Mode. If the user responds to the first query with "config" (no quotes) the following prompts will be provided:

- "Input path of configuration file":

  The user should provide a valid path to a file containing the desired reference data. Information on how this file is created and structured is provided in the methods section.

- "Input a scaling factor (1 if no change desired, must be greater than 0)":

  This query is used to linearly rescale the reference file. Values larger than 1 will result in an increase in size while values less than one will result in a decrease in size. A value of 1 will result in no change in size. User should provide a response in either integer or float format.

- "Percent of grains to be reassigned (decimal from 0–1)":

  User will specify what percentage of the grains in their structure will have their reference file replaced. Input should be a float between 0 and 1 corresponding to the decimal form of the desired percentage.

- "Input path of new reference file":

  Conditional query, only requested if previous is non-zero. User must specify a valid path to the desired secondary reference file.

- "Input crystalline texturing direction

  Reply "none" (quotations excluded) if no texturing desired

  Use following convention "index_x, index_y, index_z":

  If the user desires their structure to possess crystalline texturing, they should respond to this query with the desired texturing direction using the Miller index convention. Individual indices should be integers with commas separating indices. If texturing is not desired, the user should respond with "none" (no quotes).

Config Mode will produce a new configuration file containing all information pertinent to the newly created structure along with a LAMMPS-style data file containing atom information.

Final Input and Runtime. Once the user has defined the algorithm-critical variables the final prompt asks the user for an output name. This output should be the general name as the appropriate file extensions and flags are then automatically added. At this point the algorithm executes and generates the nanocrystalline structure. At various stages during runtime the algorithm prints status messages to the screen to verify its progress. Since default Python runtimes do not take advantage of parallel computing, large structures can take some time to generate and may require significant memory. In testing, structures containing up to 5 million atoms could be generated with fewer than 4 GB of RAM (random access memory).

4.2 Output

Two files are output by the *nanocrystal_builder.py* script upon completion. The first file is the structure configuration file. As previously detailed, this file contains information that can be used to reproduce and modify the structure while retaining the overall grain morphology. This file will be named *filename.config* where *filename* is the user-specified base file name. The second file, named *filename.data*, is a LAMMPS data file containing the ID, type, and position of all of the atoms in the structure. By default, the atom types will be based on which grain the atom

belongs to. The preamble in this file is standard to the LAMMPS data file format and contains information about the simulation cell dimensions.

5. Examples

In this section, both Single Mode and Config Mode for generating nanoscale structures using *nanocrystal_builder.py* are demonstrated. Example command prompt options for these modes are shown by the screen shot in Fig. 4. First, a 10-nm grain structure is created in a 15- $\times$ 15- $\times$ 15-nm simulation cell. Here, each grain contains an ideal face-centered cubic structure as described by the *dump.Cu* LAMMPS file (that is located in the working directory). No texture direction is specified, thus each grain has a random orientation. Assuming 100% packing efficiency, the algorithm estimates that six 10-nm grains need to be created within the 15-nm cube; however, the optimized spherical packing algorithm is only able to successfully place 5 grain centers within the volume. After determining the grain center locations, the nanostructure is built and the resulting outputs are a LAMMPS data file called *data.Cu_NC* and a configuration file saved as *Cu_NC_centroids.config*.

The *nanocrystal_builder.py* script is invoked a second time to demonstrate the Config Mode in the lower half of Fig. 4. Here, the *Cu_NC_centroids.config* file generated from the previous example is used to preserve the grain morphology (grain center locations). However, in this example the simulation dimensions are rescaled 1.5 times to create approximately 15-nm grains. Using the command prompt, 75% of the grains are randomly chosen to be reassigned to a secondary atomic structure. The secondary atomic structure, called *dump.Cu_twin* (located in the working directory), contains approximately 5-nm spaced Cu twins. Through command prompt interaction, all grain orientations are reoriented about the $[111]$ direction to observe crystallographic texturing. As previously shown, the resulting outputs from the *nanocrystal_builder.py* code are a structure and a configuration file. Both structures created in this section are shown in Fig. 5.

```
bash-4.1$ python nanocrystal_builder.py
Do you want to generate from a single reference or a
configuration file (respond single or config):
single
Input the reference file path:
./dump.Cu
Input desired grain size in angstroms:
100
Input the box dimensions in angstroms
Use the following convention lowerx, upperx, lowery,
uppery, lowerz, upperz:
0, 150, 0, 150, 0, 150
Input crystalline texturing direction
Reply "none" (quotations excluded) if no texturing desired
Use following convention index_x, index_y, index_z:
none
Input desired output basename:
Cu_NC
asigning grain centers
approximate number of grains = 6
asigning grain 1
asigning grain 2
asigning grain 3
asigning grain 4
asigning grain 5
populating atoms
populating grain 0
populating grain 1
populating grain 2
populating grain 3
populating grain 4
populating grain 5
trimming excess atoms
wrapping periodic bounds
outputting to data file
```

```
bash-4.1$ python nanocrystal_builder.py
Do you want to generate from a single reference or a
configuration file (respond single or config):
config
Input path of configuration file:
Cu_NC_centroids.config
Input a scaling factor (1 if no change desired, must be
greater than 0):
1.5
Percent of grains to be reassigned (decimal from 0-1):
0.75
Input path of new reference file:
./dump.Cu_twin
Input crystalline texturing direction
Reply "none" (quotations excluded) if no texturing desired
Use following convention index_x,index_y,index_z:
1,1,1
Input desired output basename:
Cu_NC_1.5scale_0.75twin_111
reading configuration data
reassigning 4 reference files
populating atoms
populating grain 0
populating grain 1
populating grain 2
populating grain 3
populating grain 4
populating grain 5
trimming excess atoms
wrapping periodic bounds
outputting to data file
```

Fig. 4 User feedback flow for Single Mode without texturing (top), and Config Mode with texturing, rescaling, and reassigning (bottom)

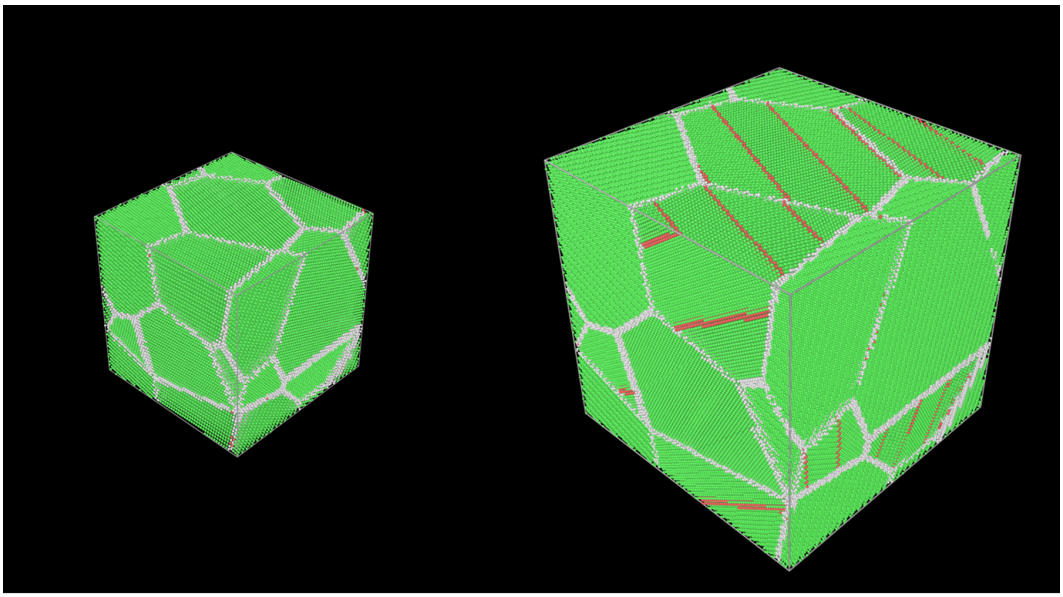

Fig. 5 Image demonstrating the outputs from the examples in Fig. 4. The nanocrystalline structure on the left is generated from Single Mode, the structure on the right is generated from Config Mode, where it has been rescaled to 150% original size and 75% of the grains have been reassigned to have twins.

6. Extensibility

6.1 Single Mode

In the current version, Single Mode allows control over grain size, simulation cell size, and crystallographic texturing. These capabilities are well suited for producing homogeneous structures with equiaxed grains but are not well suited for producing grain size gradients or non-equiaxed grain shapes. It is possible to add these capabilities by manipulating the centroid placement algorithm to use a gradient radius function or an elliptical region of allowed space instead of the homogeneous spherical space. With the current distribution this extension should be considered as future work.

6.2 Config Mode

The current version is configured to automatically use 2 separate reference files when rebuilding a structure. This is very useful for adding twinned grains to an existing granular structure, as was the intention of this project. However, should the user desire to produce a structure using additional reference files, they would have to either manually manipulate the configuration file or run Config Mode multiple

times. As both of these options are suboptimal for large structures or many references, future versions of this algorithm can include capabilities to automatically process structures with arbitrary numbers of reference files.

7. Conclusions

The *nanocrystal_builder.py* script was developed to generate nanocrystalline structures with flexibility to alter texture, reference structures, and grain sizes. The algorithm uses an optimized packing routine to produce realistic grain shapes and size distributions. An efficient nearest-neighbor matching algorithm is leveraged to eliminate excess atoms. To increase functionality, the algorithm is able to read in previously built structures and modify them at user request. This functionality allows the user to resize, twin, or alloy a structure while retaining grain geometry. In testing, the algorithm successfully produced a series of structures in which the grain size, twin density, and texturing axis were varied while the grain geometry remained consistent.

The algorithm produced from this research is currently being used to study the effect of nanotwins on the mechanical response during deformation. This study is being complemented by the use of virtual diffraction techniques to directly compare results to experiments. Ongoing efforts are being made to expand the algorithm to accommodate more complex structures, such as size distributions and irregular grain sizes.

The *nanocrystal_builder.py* code can be downloaded by clicking here.

8. References

1. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. J Comput Phys. 1995;117:1–19.

2. LAMMPS molecular dynamics simulator. Albuquerque (NM): Sandia National Laboratories; 2016 [accessed 2016 July]. http://lammps.sandia.gov/index.html.

3. Stukowski A. Visualization and analysis of atomistic simulation data with OVITO – the open visualization tool. Modelling Simul Mater Sci Eng. 2010;18:015012. http://ovito.org/.

INTENTIONALLY LEFT BLANK.

Appendix. Python Script: nanocrystal_builder.py Function

This appendix appears in its original form, without editorial change.

```
# File: nanocrystal_builder.py
# Script builds nanocrystalline centers using optimized grain center placement
# Daniel Foley
# Army Research Laboratory
# Summer 2015/2016

#----------Initialize----------
import os
import sys
import numpy as np
from scipy import spatial as sp
import random
import linecache
import math as mt

reference_mode = raw_input('Do you want to generate from a single reference or a configuration file ' \
                    '(respond single or config):\n')

while reference_mode != 'single' and reference_mode != 'config':
    print ("I could not interpret your choice, please check your spelling and try again.")
    reference_mode = raw_input('single or config? ')

if reference_mode=='single':
    crref = raw_input('Input the reference file path:\n')

    avg_grain_size = float(raw_input('Input desired grain size in angstroms:\n'))
```

```
    box_bounds = raw_input('Input the box dimensions in angstroms\nUse the following convention' \
                           ' lowerx, upperx, lowery, uppery, lowerz, upperz:\n')
    direction = raw_input('Input crystalline texturing direction\nReply "none" (quotations excluded)' \
                          ' if no texturing desired\nUse following convention ' \
                          'index_x, index_y, index_z:\n')
    if direction != 'none':
        direction = np.fromstring(direction,sep=',')
        direcnorm = np.linalg.norm(direction)
        direction = direction/direcnorm

elif reference_mode=='config':
    ref_list=raw_input('Input path of configuration file:\n')
    rescale_factor = float(raw_input('Input a scaling factor (1 if no change desired, must be ' \
                                     'greater than 0):\n'))
    contwin = float(raw_input('Percent of grains to be reassigned (decimal from 0-1):\n'))
    if contwin != 0.:
        twinr = raw_input('Input path of new reference file:\n')
    else:
        twinr = 'foo'
    direction = raw_input('Input crystalline texturing direction\nReply "none" (quotations ' \
                          'excluded) if no texturing desired\nUse following convention ' \
                          'index_x,index_y,index_z:\n')
    if direction != 'none':
        direction = np.fromstring(direction,sep=',')
        direcnorm = np.linalg.norm(direction)
        direction = direction/direcnorm
```

```
output_name = raw_input('Input desired output basename:\n')

#----------Define Functions----------

def efficient_packing(center,grain_size):
     #defines a point space in which the next grain center may be placed
    rot = np.linspace(0.,360*np.pi/180.,45).reshape(45,1)
    r = np.linspace(.75*grain_size,1.25*grain_size,20).reshape(20,1)
    zero = np.zeros((20,2))
    r = np.concatenate((r,zero),axis = 1)
    point_space = np.empty([20*45,3])
    n = 0
    for j in range(0,20): #this can probably be vectorized... work in progress
        for i in range(0,45):
            R_y = np.array([[np.cos(rot[i,0]),0,np.sin(rot[i,0])],[0,1,0], \
                            [-np.sin(rot[i,0]),0,np.cos(rot[i,0])]])
            point_space[n,0:3] = np.dot(r[j,0:3],R_y)
            n = n + 1
    point_space_2 = np.empty([20*45*45,3])
    n = 0
    for j in range(0,len(point_space)): #this can probably be vectorized... work in progress
        for i in range(0,45):
            R_z = np.array([[np.cos(rot[i,0]),-np.sin(rot[i,0]),0],[np.sin(rot[i,0]), \
                            np.cos(rot[i,0]),0],[0,0,1]])
            point_space_2[n,0:3] = np.dot(point_space[j,0:3],R_z)
```

```
            n = n+1
    point_space_2 = point_space_2 + center
    return point_space_2

def ghost_centers(centroids,box_limits):
     #defines grain centers as seen across periodic bounds as "ghosts"
    ghost_x = np.array([-(box_limits[0,1]-box_limits[0,0]),0,(box_limits[0,1]-box_limits[0,0])])
    ghost_y = np.array([-(box_limits[1,1]-box_limits[1,0]),0,(box_limits[1,1]-box_limits[1,0])])
    ghost_z = np.array([-(box_limits[2,1]-box_limits[2,0]),0,(box_limits[2,1]-box_limits[2,0])])
        ghost = np.empty([len(ghost_x)*len(ghost_y)*len(ghost_z),3])
        roll = 0
        for i in range(0,len(ghost_x)): #this can probably be vectorized... work in progress
        for j in range(0,len(ghost_y)):
            for k in range(0,len(ghost_z)):
                            ghost[roll,0] = ghost_x[i]
                                ghost[roll,1] = ghost_y[j]
                                ghost[roll,2] = ghost_z[k]
                                roll = roll + 1
    if len(np.shape(centroids)) == 1:
        grain_centers_1 = centroids.reshape(1,3)
        grain_centers_2 = centroids.reshape(1,3)
    else:
        grain_centers_1 = centroids
        grain_centers_2 = centroids
    for i in range(0,len(ghost)):
        if ghost[i,0] == 0 and ghost[i,1] == 0 and ghost[i,2] == 0:
```

```
            continue
        else:
            hold_center = np.add(grain_centers_1[:,0:3], ghost[i,:])
            grain_centers_2 = np.concatenate((grain_centers_2,hold_center),axis=0)
    return grain_centers_2

def grain_centroid(amorphous_edge,grain_size):
     #assigns grain centers within an allowed space defined by the positions of existing grain centers
    box_vol = (amorphous_edge[0,1]-amorphous_edge[0,0])*(amorphous_edge[1,1]-amorphous_edge[1,0]) \
                *(amorphous_edge[2,1]-amorphous_edge[2,0])
    grain_volume = ((4./3.)*(grain_size / 2)**3)*np.pi
    N = int(box_vol / grain_volume)
    print 'approximate number of grains = %i' % (N)
    centroids = np.array([[(grain_size / 2.),(grain_size / 2.),(grain_size / 2.)]])
    allowed = efficient_packing(centroids[0,0:3],grain_size)
    allowed = transpose_periodic_bounds(allowed,amorphous_edge)
    ghost = ghost_centers(centroids[0,:],amorphous_edge)
    for i in range(1,N):
        print 'asigning grain %i' % (i)
        if len(allowed) == 0:
            print 'out of space'
            break
        choose = random.randint(0,len(allowed)-1)
        centroids = np.concatenate((centroids,allowed[choose,0:3].reshape(1,3)))
        allowed = np.concatenate((allowed,efficient_packing(centroids[i,0:3],grain_size)))
        centtree = sp.cKDTree(ghost_centers(centroids[0:i+1,:],amorphous_edge))
```

```
        dist, index = centtree.query(allowed)
        deny = np.where(dist < .75*grain_size,False,True)
        if np.any(deny) == 'False':
            print 'out of space, %i grains assigned' % (i)
            break
        allowed = allowed[deny,:]
        allowed = transpose_periodic_bounds(allowed,amorphous_edge)
    return centroids


def seed_rotations(refdat,grain_center,grabradius,grabedges,axis,angle):
        #function extracts and rotates crystalline seeds (updated in version 2)
    holdx1 = int(grabedges[0,0]+grabradius)
    holdx2 = int(grabedges[0,1]-grabradius)
    if holdx1 == holdx2:
        holdx1 = holdx1 - 2
        holdx2 = holdx2 + 2
    if holdx1 > holdx2:
        holdx1 = int(grabedges[0,1]-grabradius)
        holdx2 = int(grabedges[0,0]+grabradius)
    holdy1 = int(grabedges[1,0]+grabradius)
    holdy2 = int(grabedges[1,1]-grabradius)
    if holdy1 == holdy2:
        holdy1 = holdy1 - 2
        holdy2 = holdy2 + 2
    if holdy1 > holdy2:
```

```
    holdy1 = int(grabedges[1,1]-grabradius)
    holdy2 = int(grabedges[1,0]+grabradius)
holdz1 = int(grabedges[2,0]+grabradius)
holdz2 = int(grabedges[2,1]-grabradius)
if holdz1 == holdz2:
    holdz1 = holdz1 - 2
    holdz2 = holdz2 + 2
if holdz1 > holdz2:
    holdz1 = int(grabedges[2,1]-grabradius)
    holdz2 = int(grabedges[2,0]+grabradius)
    x_c = float(random.randint(holdx1,holdx2))
    y_c = float(random.randint(holdy1,holdy2))
    z_c = float(random.randint(holdz1,holdz2))
    dtree = sp.cKDTree(refdat[:,2:5])
    neighpoint = dtree.query_ball_point([x_c,y_c,z_c], grabradius)
    datsphere = refdat[np.asarray(neighpoint),2:5]
    datsphere1 = datsphere[:,0:3]-[x_c,y_c,z_c]
R = np.array([[np.cos(angle)+(axis[0]**2)*(1-np.cos(angle)),\
                  axis[0]*axis[1]*(1-np.cos(angle))-axis[2]*np.sin(angle),\
                  axis[0]*axis[2]*(1-np.cos(angle))+axis[1]*np.sin(angle)],\
                  [axis[1]*axis[0]*(1-np.cos(angle))+axis[2]*np.sin(angle),\
                  np.cos(angle)+(axis[1]**2)*(1-np.cos(angle)), \
                  axis[1]*axis[2]*(1-np.cos(angle))-axis[0]*np.sin(angle)],\
                  [axis[2]*axis[0]*(1-np.cos(angle))-axis[1]*np.sin(angle),\
                  axis[2]*axis[1]*(1-np.cos(angle))+axis[0]*np.sin(angle),\
                  np.cos(angle)+(axis[2]**2)*(1-np.cos(angle))]])
```

```
        datsphererot = np.dot(datsphere1, R)
        datspherefinal = datsphererot + grain_center
        return datsphere, datspherefinal

def check_ownership(grain_centers,atoms,box_limits):
     #checks to make sure an atom's nearest center is its parent center, deletes atom if not
    grain_centers = ghost_centers(grain_centers,box_limits)
    centertree = sp.cKDTree(grain_centers)
    dist, label = centertree.query(atoms[:,0:3])
    check = np.equal(atoms[:,3],label)
    output_atoms = atoms[check,0:4]
    return output_atoms, grain_centers

def transpose_periodic_bounds(atoms,box_lims):
     #reflects atoms which exist outside the box bounds through the periodic conditions
    lower = np.empty([len(atoms),3])
    lower[:,0:3] = [box_lims[0,0],box_lims[1,0],box_lims[2,0]]
    upper = np.empty([len(atoms),3])
    upper[:,0:3] = [box_lims[0,1],box_lims[1,1],box_lims[2,1]]
    test_l = np.less_equal(atoms[:,0:3],lower[:,0:3])
    test_u = np.greater_equal(atoms[:,0:3],upper[:,0:3])
    shift_l = np.where(test_l,1,0)
    shift_u = np.where(test_u,-1,0)
    shift_L = np.multiply(shift_l[:,0:3],np.array([box_lims[0,1]-box_lims[0,0],\
                                                   box_lims[1,1]-box_lims[1,0],\
                                                   box_lims[2,1]-box_lims[2,0]]))
```

```
    shift_U = np.multiply(shift_u[:,0:3],np.array([box_lims[0,1]-box_lims[0,0],\
                                                   box_lims[1,1]-box_lims[1,0],\
                                                   box_lims[2,1]-box_lims[2,0]]))
    out_atoms = atoms
    out_atoms[:,0:3] = np.add(atoms[:,0:3],shift_L)
    out_atoms[:,0:3] = np.add(atoms[:,0:3],shift_U)
    return out_atoms

def reference_rescale(ref_atoms,box_lims,grain_size):
     #function resizes the reference structure to accomodate requested grain size
    rescale_length = np.array([(grain_size - (box_lims[0,1]-box_lims[0,0])),\
                               (grain_size - (box_lims[1,1]-box_lims[1,0])),\
                               (grain_size - (box_lims[2,1]-box_lims[2,0]))])
    if rescale_length[0] > 0.0 \
         or rescale_length[1] > 0.0\
         or rescale_length[2] > 0.0:
        n_x = int(rescale_length[0]/(box_lims[0,1]-box_lims[0,0]))+2
        n_y = int(rescale_length[1]/(box_lims[1,1]-box_lims[1,0]))+2
        n_z = int(rescale_length[2]/(box_lims[2,1]-box_lims[2,0]))+2
        new_ref = ref_atoms
        for i in range(0,n_x+1):
            for j in range(0,n_y+1):
                for k in range(0,n_z+1):
                    if i == 0 and k == 0 and j == 0:
                        continue
                    else:
```

```
                        new_ats = ref_atoms + [0,0,float(i)*(box_lims[0,1]-box_lims[0,0]),\
                                               float(j)*(box_lims[1,1]-box_lims[1,0]),\
                                               float(k)*(box_lims[2,1]-box_lims[2,0])]
                        new_ref = np.concatenate((new_ref,new_ats))
    else:
        replicate_range = np.array([[box_lims[0,0],box_lims[0,0]+2*rescale_length[0]],\
                                    [box_lims[1,0],box_lims[1,0]+2*rescale_length[1]],\
                                    [box_lims[2,0],box_lims[2,0]+2*rescale_length[2]]])
        check_x=np.where(ref_atoms[:,2] <= replicate_range[0,1],True,False)
        rep_atoms_x = ref_atoms[check_x[:],:]
        rep_atoms_x[:,2]=rep_atoms_x[:,2] + (box_lims[0,1]-box_lims[0,0])
        rep_atoms = np.concatenate((ref_atoms,rep_atoms_x),axis=0)
        check_y=np.where(rep_atoms[:,3] <= replicate_range[1,1],True,False)
        rep_atoms_y = rep_atoms[check_y[:],:]
        rep_atoms_y[:,3]=rep_atoms_y[:,3] + (box_lims[1,1]-box_lims[1,0])
        rep_atoms = np.concatenate((rep_atoms,rep_atoms_y),axis=0)
        check_z=np.where(rep_atoms[:,4] <= replicate_range[2,1],True,False)
        rep_atoms_z = rep_atoms[check_z[:],:]
        rep_atoms_z[:,4]=rep_atoms_z[:,4] + (box_lims[2,1]-box_lims[2,0])
        rep_atoms = np.concatenate((rep_atoms,rep_atoms_z),axis=0)
        new_ref = rep_atoms
    return new_ref

def multireference(ref_list,grain_size,alimits,rescale,percent_twin,twin_reference,axis,angle):
     #function will extract centroid and box data from config file and build a new structure accordingly
    c_refs = np.loadtxt('%s' % (ref_list),skiprows=2,\
```

```
                dtype={'names':('id','c_x','c_y','c_z','axisr_x','axisr_y','axisr_z','angler','file'),\
                'formats':('i4','f4','f4','f4','f4','f4','f4','f4','S32')})
n_grains = float(len(c_refs))
n_twinned = n_grains * percent_twin
check = np.empty((int(n_twinned),1),dtype=int)
print 'reassigning %i reference files' % (n_twinned)
for i in range(0,int(n_twinned)):
    itl = random.randint(0,len(c_refs)-1)
    while np.any(check[:]==itl)==True:
        itl = random.randint(0,len(c_refs)-1)
    c_refs['file'][itl] = '%s' % (twin_reference)
    check[i,0] = itl
center = np.empty((len(c_refs),3))
f=open('%s_centroids.config' % (output_name),'w')
f.write('#data for centroids\n')
f.write('%f %f %f %f %f %f %f\n' % (grain_size*rescale, alimits[0,0], alimits[0,1]*rescale,\
          alimits[1,0], alimits[1,1]*rescale, alimits[2,0], alimits[2,1]*rescale))
finaldat = np.empty([0,4])
print 'populating atoms'
for i in range(0,len(c_refs)):
    print 'populating grain %i' % (i)
    limsx = np.fromstring(linecache.getline('%s' % (c_refs['file'][i]),6),sep=' ')
    limsy = np.fromstring(linecache.getline('%s' % (c_refs['file'][i]),7),sep=' ')
    limsz = np.fromstring(linecache.getline('%s' % (c_refs['file'][i]),8),sep=' ')
    limits = np.concatenate((limsx,limsy,limsz),axis=0).reshape(3,2)
    cdat=np.loadtxt('%s' % c_refs['file'][i],skiprows=9,usecols=(0,1,2,3,4))
```

```
if 1.5*grain_size*rescale >= (limits[0,1]-limits[0,0]) \
      or 1.5*grain_size*rescale >= (limits[1,1]-limits[1,0]) \
      or 1.5*grain_size*rescale >= (limits[2,1]-limits[2,0]):
    cdat = reference_rescale(cdat,limits,1.5*grain_size*rescale)
    limits = np.array([[np.amin(cdat[:,2]),np.amax(cdat[:,2])],[np.amin(cdat[:,3]),\
                np.amax(cdat[:,3])],[np.amin(cdat[:,4]),np.amax(cdat[:,4])]])
if axis != 'none':
    c_refs['axisr_x'] = axis[0]
    c_refs['axisr_y'] = axis[1]
    c_refs['axisr_z'] = axis[2]
    haxis = axis
    f.write('%i %f %f %f %f %f %f %f %s\n' % (i, c_refs['c_x'][i]*rescale, \
                c_refs['c_y'][i]*rescale, c_refs['c_z'][i]*rescale, c_refs['axisr_x'][i],\
                c_refs['axisr_y'][i], c_refs['axisr_z'][i], c_refs['angler'][i], \
                c_refs['file'][i]))
    center[i,:]=[c_refs['c_x'][i]*rescale,c_refs['c_y'][i]*rescale,c_refs['c_z'][i]*rescale]
    [trydat, seeddat]=seed_rotations(cdat,[c_refs['c_x'][i]*rescale,\
                                           c_refs['c_y'][i]*rescale,\
                                           c_refs['c_z'][i]*rescale],\
                                           1.5*rescale*grain_size/2,limits,haxis,\
                                           c_refs['angler'][i])
else:
    f.write('%i %f %f %f %f %f %f %f %s\n' % (i, c_refs['c_x'][i]*rescale, \
                                                 c_refs['c_y'][i]*rescale, \
                                                 c_refs['c_z'][i]*rescale, \
                                                 c_refs['axisr_x'][i], \
```

```
                                                        c_refs['axisr_y'][i], \
                                                        c_refs['axisr_z'][i], \
                                                        c_refs['angler'][i], \
                                                        c_refs['file'][i]))
            center[i,:]=[c_refs['c_x'][i]*rescale,c_refs['c_y'][i]*rescale,c_refs['c_z'][i]*rescale]
            haxis = np.array([c_refs['axisr_x'][i], c_refs['axisr_y'][i], c_refs['axisr_z'][i]])
            [trydat, seeddat]=seed_rotations(cdat,[c_refs['c_x'][i]*rescale,\
                                                   c_refs['c_y'][i]*rescale,\
                                                   c_refs['c_z'][i]*rescale],\
                                                   1.5*rescale*grain_size/2,limits,haxis,\
                                                   c_refs['angler'][i])

        seeddat = np.insert(seeddat,3,i,axis=1)
        finaldat = np.concatenate((finaldat,seeddat))
        linecache.clearcache()
    f.close()
    return finaldat, center

#----------Main Body----------

if reference_mode == 'single':
    cdat = np.loadtxt('%s' % (crref),skiprows=9,usecols=(0,1,2,3,4))
    limsx = np.fromstring(linecache.getline('%s' % (crref),6),sep=' ')
    limsy = np.fromstring(linecache.getline('%s' % (crref),7),sep=' ')
    limsz = np.fromstring(linecache.getline('%s' % (crref),8),sep=' ')
    lims = np.concatenate((limsx,limsy,limsz),axis=0).reshape(3,2)
```

```
#   print lims
    # Check if the average grain size is larger than the box bounds
    if 1.5*avg_grain_size >= (lims[0,1]-lims[0,0]) or 1.5*avg_grain_size >= (lims[1,1]-lims[1,0]) \
                                    or 1.5*avg_grain_size >= (lims[2,1]-lims[2,0]):
        cdat = reference_rescale(cdat,lims,1.5*avg_grain_size)
        lims = np.array([[np.amin(cdat[:,2]),np.amax(cdat[:,2])],\
                         [np.amin(cdat[:,3]),np.amax(cdat[:,3])],\
                         [np.amin(cdat[:,4]),np.amax(cdat[:,4])]])
    lims2 = np.fromstring(box_bounds,sep=',').reshape(3,2)
    if avg_grain_size >= (lims2[0,1]-lims2[0,0]) or avg_grain_size >= (lims2[1,1]-lims2[1,0]) \
                                        or avg_grain_size >= (lims2[2,1]-lims2[2,0]):
        sys.exit("grain size larger than defined simulation box, change box sizes")
    print 'assigning grain centers'
    center = grain_centroid(lims2,avg_grain_size)
    finaldat = np.empty([0,4])
    a = np.ones((len(cdat),1)).reshape(len(cdat),1)
    b = np.arange(1,len(cdat)+1).reshape(len(cdat),1)
    cdat = np.concatenate((b,a,cdat[:,2:5]),axis=1)
    centout=np.empty((len(center),8))
    f=open('%s_centroids.config' % (output_name),'w')
    f.write('#data for centroids\n')
    f.write('%f %f %f %f %f %f %f\n' % (avg_grain_size,lims2[0,0],lims2[0,1],lims2[1,0],\
                                                    lims2[1,1],lims2[2,0],lims2[2,1]))
    print 'populating atoms'
    for i in range(0,len(center)):
        print 'populating grain %i' % (i)
```

```
        if direction == 'none':
            axis = np.array([random.uniform(0,10),random.uniform(0,10),random.uniform(0,10)])
            axnorm = np.linalg.norm(axis)
            axis = axis/axnorm
            angle = mt.radians(random.uniform(0,180))
        else:
            axis = direction
            angle = random.uniform(0,180)
        f.write('%i %f %f %f %f %f %f %f %s\n' % (i,center[i,0],center[i,1],center[i,2],axis[0],\
                                                  axis[1],axis[2],angle,crref))
        [trydat, seeddat] = seed_rotations(cdat,center[i,0:3],1.5*avg_grain_size/2,lims,axis,angle)
        seeddat = np.insert(seeddat,3,i,axis = 1)
        finaldat = np.concatenate((finaldat,seeddat))
    f.close()
    print 'finaldat=', len(finaldat)

    print 'trimming excess atoms'
    cut_atoms,check_centers = check_ownership(center,finaldat[:,0:4],lims2)
    print 'cut_atoms=', len(cut_atoms)
    print 'wrapping periodic bounds'
    fin_atoms = transpose_periodic_bounds(cut_atoms,lims2)
    idcol = np.ones([len(cut_atoms),1])
    fin_atoms = np.concatenate((idcol,cut_atoms), axis = 1)
    numcol = np.arange(1,len(fin_atoms)+1).reshape(len(fin_atoms),1)
    fin_atoms = np.concatenate((numcol,fin_atoms), axis = 1)
    output_atoms = np.column_stack((fin_atoms[:,0],fin_atoms[:,5]+1,fin_atoms[:,2],\
```

```
                                        fin_atoms[:,3],fin_atoms[:,4]))

    print 'outputting to data file'
    head = ("#data file for lammps\n%i atoms\n%s atom types\n%f %f xlo xhi\n%f %f ylo "\
             "yhi\n%f %f zlo zhi\n\nAtoms\n") % (len(output_atoms),len(center)+1,lims2[0,0],\
                                                 lims2[0,1],lims2[1,0],lims2[1,1],lims2[2,0],lims2[2,1])
    np.savetxt('data.%s' % (output_name),output_atoms, fmt='%i %i %f %f %f')
    f=open('data.%s' % (output_name),'r')
    data1 = f.read()
    f.close()

    g=open('data.%s' % (output_name),'w')
    g.write('%s\n' % (head))
    g.write('%s' % (data1))
    g.close()
else:
    print 'reading configuration data'
    conf = np.fromstring(linecache.getline('%s' % (ref_list),2),sep=' ')
    avg_grain_size = conf[0]
    lims = conf[1:].reshape(3,2)
    lims2 = lims * rescale_factor
    axis = direction
    angle = mt.radians(random.uniform(0,180))
    [finaldat, center] = multireference(ref_list,avg_grain_size,lims,rescale_factor,contwin,\
                                        twinr,axis,angle)
    if avg_grain_size*rescale_factor >= (lims2[0,1]-lims2[0,0]) \
```

```
        or avg_grain_size*rescale_factor >= (lims2[1,1]-lims2[1,0]) \
            or avg_grain_size*rescale_factor >= (lims2[2,1]-lims2[2,0]):
    sys.exit("grain size larger than defined simulation box, change box sizes")

a = np.ones((len(finaldat),1)).reshape(len(finaldat),1)
b = np.arange(1,len(finaldat)+1).reshape(len(finaldat),1)
finaldat = np.concatenate((b,a,finaldat),axis=1)

print 'trimming excess atoms'
cut_atoms,check_centers = check_ownership(center,finaldat[:,2:6],lims2)
a = np.ones((len(cut_atoms),1)).reshape(len(cut_atoms),1)
b = np.arange(1,len(cut_atoms)+1).reshape(len(cut_atoms),1)
cut_atoms = np.concatenate((b,a,cut_atoms),axis=1)

print 'wrapping periodic bounds'
fin_atoms = transpose_periodic_bounds(cut_atoms[:,2:5],lims2)
idcol = np.ones([len(cut_atoms),1])
fin_atoms = np.concatenate((idcol,cut_atoms), axis = 1)
numcol = np.arange(1,len(fin_atoms)+1).reshape(len(fin_atoms),1)
fin_atoms = np.concatenate((numcol,fin_atoms), axis = 1)

output_atoms = np.column_stack((fin_atoms[:,0],fin_atoms[:,7]+1,fin_atoms[:,4],\
                                fin_atoms[:,5],fin_atoms[:,6]))

print 'outputting to data file'
head = ("#data file for lammps\n%i atoms\n%s atom types\n%f %f xlo xhi\n%f %f ylo "\
```

```
        "yhi\n%f %f zlo zhi\n\nAtoms\n") % (len(fin_atoms),len(center)+1,lims2[0,0],\
                                            lims2[0,1],lims2[1,0],lims2[1,1],lims2[2,0],lims2[2,1])
np.savetxt('data.%s' % (output_name),output_atoms, fmt='%i %i %f %f %f')
f=open('data.%s' % (output_name),'r')
data1 = f.read()
f.close()

g=open('data.%s' % (output_name),'w')
g.write('%s\n' % (head))
g.write('%s' % (data1))
g.close()
```

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	US ARMY RESEARCH LAB RDRL CIO L IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	DIR USARL RDRL WMM F M TSCHOPP S COLEMAN