ARL-TR-8914 • MAR 2020

Computational Model Builder and Analysis Toolkit (COMBAT) Demonstrating Capabilities through Practical Examples

by Dylan M Anstine, Chi-Chin Wu, James P Larentzos, and John K Brennan

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-TR-8914 ● MAR 2020

Computational Model Builder and Analysis Toolkit (COMBAT) Demonstrating Capabilities through Practical Examples

Dylan M Anstine
Department of Materials Science and Engineering, University of Florida

Chi-Chin Wu, James P Larentzos, and John K Brennan
Weapons and Materials Research Directorate, CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
March 2020	Technical Report	13 May – 31 August 2019

4. TITLE AND SUBTITLE
Computational Model Builder and Analysis Toolkit (COMBAT) Demonstrating Capabilities through Practical Examples

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Dylan M Anstine, Chi-Chin Wu, James P Larentzos, and John K Brennan

5d. PROJECT NUMBER
HIP-19-021

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLW-LB
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8914

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Department of Defense (DOD) High Performance Computing Modernization Program (HPCMP)
Vicksburg, MS

10. SPONSOR/MONITOR'S ACRONYM(S)
HPCMP

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

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

13. SUPPLEMENTARY NOTES
ORCID ID(s): Chi-Chin Wu, 0000-0002-6036-3271; James P Larentzos, 0000-0002-9873-4349; John K Brennan, 0000-0001-9573-5082

14. ABSTRACT
Particle-based simulations that access nanometer-to-micrometer length and nanosecond-to-microsecond timescales are becoming an increasingly common research practice. This report highlights the use of the Computational Model Builder and Analysis Toolkit (*combat*) to accomplish routine atomistic or coarse-grained modeling tasks for these types of simulations by using simple python programming. Several practical examples are presented that demonstrate both newly developed and pre-existing functionality of the *combat* software package. An additional module, *combat_analysis*, has been added that can leverage the parallel computational efficiency of *combat* to perform a number of in silico analytical characterization techniques. Each example section is clearly organized with a defined modeling task, a description of the code that uses *combat* to accomplish the task, and an accompanying graphical representation for the process. Although the demonstrations provided highlight a small fraction of *combat*'s functionalities, ideally the examples discussed will inspire creative use of the software's capabilities for a range of material systems at scales up to, or beyond, the micro regime.

15. SUBJECT TERMS
analysis toolkit, molecular modeling, simulation, data processing, system preparation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Chi-Chin Wu
Unclassified	Unclassified	Unclassified	UU	33	19b. TELEPHONE NUMBER (Include area code) (410) 306-1905

Contents

List of Figures

List of Tables

Acknowledgment

The authors acknowledge the Department of Defense (DOD) High Performance Computing Modernization program (HPCMP) for providing funding to Dylan M Anstine through the HPCMP internship program (HIP-19-021). Additionally, the authors express their gratitude to the support staff associated with the DOD Supercomputing Resource Center. Finally, the authors are grateful for helpful discussions with Dr Brian Barnes (US Army Combat Capabilities Development Command Army Research Laboratory [ARL]), Dr Betsy Rice (ARL), and Kelsea Miller (Texas Tech University).

1. Introduction

The evolving capabilities of computing hardware and the development of efficient, often parallel algorithms implemented in particle-based simulation software have enabled simulations that now extend to the microscale and beyond. These large-scale simulations consisting of millions to billions of particles over nanosecond-to-microsecond timescales present enormous challenges in "big data" analysis, where representative examples are Mattox et al.[1] and Jaramillo et al.[2] A python-based toolkit for particle simulations, Computational Model Builder and Analysis Toolkit (COMBAT), hereafter referred to as *combat* (available at https://github.com/USArmyResearchLab/ARL-COMBAT), was recently developed and described in Fortunato et al.[3] That report highlighted the use of organized Pandas DataFrames[4] to describe model systems and parallel computations for analysis and processing. In the current report, we present a number of examples that demonstrate new functionality of *combat* and highlight the use of simple python programming to perform a range of pre- and postprocessing tasks. While the chosen examples demonstrate only a small fraction of *combat*'s capabilities, ideally they create perspective insights into the potential applicability to a broad range of material systems modeled at atomistic and coarse-grain scales. In addition to the pre-existing *combat* functionalities, a new module was added, *combat_analysis*, which consists of a continuously expanding set of computational analytical tools. Currently, *combat_analysis* houses common methods, such as histogram generators, designed to leverage the computational efficiency of *combat* to readily perform in silico characterization and analysis.

2. The *combat* Software Package

A significant amount of the pre- and postprocessing analysis performed on large-scale simulations across different research groups is accomplished by "in-house/in-group" code. This has been in large part based on a lack of accessibility to the necessary computing power to perform such simulations for most computational researchers. However, combining the fact that large-scale simulations are becoming more attainable with the rapid growth of data informatics, a need for a large-scale pre- and postprocessing software is steadily growing. The original intent of *combat* was to provide a platform for addressing some of the large-scale computing needs by exploiting the efficiency of parallelizable python-based scientific computing libraries to provide readily available functionality for processing large-scale particle simulations. The scope of *combat*'s development has been focused particularly on systems for simulations performed with the Large-Scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) software.[5]

Essentially, large quantities of particle data can be processed by performing calculations on an entire data frame of attributes, which can be easily partitioned across multiple central processing units, as opposed to much slower "for loops" and object-by-object processing. Additionally, *combat* is flexible by design because of its potential to use existing python computing libraries.

The overarching goal of the *combat* software package is to provide a set of tools written in python language to enable the pre- and postprocessing of model systems for molecular dynamic (MD) simulations. The software is centered on using efficient vectorized calculations, which makes *combat* particularly well-suited for large systems [$O(10^6–10^8)$ particles]. An overview of *combat*'s structure and workflow is given in Fig. 1. The main unit to the *combat* software package is the system class object, which houses information about the simulation cell (e.g., box dimensions), a data frame of particle attributes, and a collection of data frames containing molecular connectivity definitions (in the case of bonded systems). Additionally, the *combat* system is accompanied by many class functions that perform routine pre- and postprocessing methods. For example, bond_lengths() is a function that when called will calculate and append the lengths of all bonds contained in the bonds data frame. Furthermore, the design of *combat* allows for simple application of available SciPy,[6] Numpy,[7] and Pandas functionality to calculate system properties or perform operations that are not directly supplied by the functions in the system class object. As a demonstration, the example in the following makes use of linear algebra functions available in Numpy to calculate the distances among particles from a defined point. To highlight some of the capabilities that are readily available in *combat*, and to potentially excite creative usage of the software package, example applications for a variety of chemical systems are provided in the remainder of this report. For the seven examples presented, the first five are for preprocessing functionality, while the last two demonstrate postprocessing and analysis. The structure of each example contains the following:

- Description of the objective of the example with relevant background information
- Elaboration of important steps taken in the python script that use *combat* to accomplish the defined objective
- A line-numbered table containing the python code and annotated comments to illustrate the python syntax required to accomplish the defined objective
- A figure to provide a visual representation of the pre- and postprocessing or analysis performed in the example

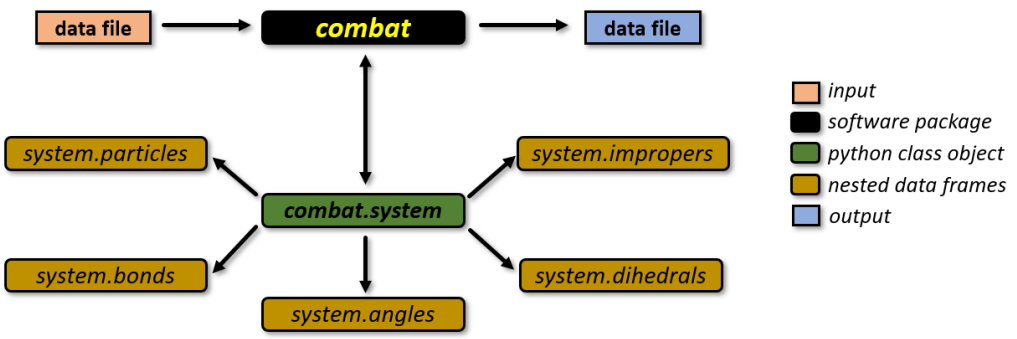


Fig. 1 Schematic representation of the *combat* software package structure and workflow. An input file is used to create a system class object. Operations are performed on the data frames nested in the *combat* system class object to produce an output data file.

2.1 Example 1: Creating a Face-Centered Cubic (FCC) Surface

A common use of the *combat* software package is to read in a molecular or material system from a configuration file (e.g., a LAMMPS-formatted configuration file [*.lmps*]), perform a modification on the system, and then write a new *.lmps* configuration file that can be used in a molecular dynamics simulation. As a demonstration, this example will read in a bulk FCC material, then perform the necessary operations to modify the system and output an *.lmps* configuration file of an FCC structure with a new surface in perpendicular to the z dimension.

To begin, an FCC crystal structure is generated using LAMMPS functionality (see Table A-1 in the Appendix for a sample LAMMPS input script that can be used to produce an FCC crystal structure). The FCC system is then modified through the set of python commands shown in Table 1 to generate an FCC surface. The system class object function *from_data()* is called to read in an *.lmps* configuration file to *combat*, where the *.lmps* configuration file name is provided as an argument. An additional argument (atom_style) is specified to describe the attributes associated with the atoms. For the python code given in the following, the atom style is specified to be "charge", indicating that all atoms contained within the *.lmps* configuration file are associated with the following attributes: the atom type, the partial charge, and the x, y, and z atomic coordinates. These properties can be displayed by using the python print function on the particles data frame, as demonstrated in Table 1.

Once the *.lmps* configuration file is used to generate a *combat* system, one can easily access and manipulate the properties of the system to produce a material with a surface perpendicular to the z dimension. The simulation cell dimensions of the system are defined through the lower (xlo, ylo, zlo) and upper (xhi, yhi, zhi) boundary system class attributes, which can be altered to create a material surface.

For the following example, an FCC surface is arbitrarily chosen to be produced that is oriented along the z direction by defining a variable that is equal to 75% of the z-dimension box length (see the surface_z variable in Table 1). The python command given at Line 8 is used to update the particles data frame such that the data frame includes only those particles that have a z coordinate smaller than surface_z. By deleting those particles that reside above surface_z, the resulting atom indices are now discontinuous and must be re-indexed to run an LAMMPS MD simulation. To accomplish the indexing, a Pandas function (i.e., the *reset_index()* function given on Line 10) operates on the particles data frame. Note that the Pandas function is not provided directly in the *combat* source code and is included as part of a separate library that is applicable to *combat.* This emphasizes the aforementioned flexibility to use both the functionality implemented in *combat* as well as the continuously increasing functionality of its dependencies (e.g., Pandas or SciPy). The final step is to use the system class object function *write_data()* to write a new *.lmps* configuration file that can be used for the MD simulations of an FCC surface. The example python script follows, alongside the initial and final structures (Fig. 2).

Table 1 Python script that uses *combat* to convert a bulk FCC single crystal to a material with an FCC structure and a surface at 75% of z box dimension by deleting particles with z positions greater than 75% of the z box length

Line #	
1	import combat
2	
3	s = combat.System.from_data('example_input.lmps', atom_style = 'charge')
4	print(s.particles)
5	print(s.xlo, s.ylo, s.zlo, s.xhi, s.yhi, s.zhi)
6	# delete particles to form the surface
7	surface_z = s.zhi * 0.75
8	s.particles = s.particles[s.particles.z < surface_z]
9	# reset particle indicies
10	s.particles.reset_index(drop=True, inplace=True)
11	s.particles.index += 1
12	# write new system to be used in MD simulations
13	s.write_data('example_output.lmps', atom_style='charge')

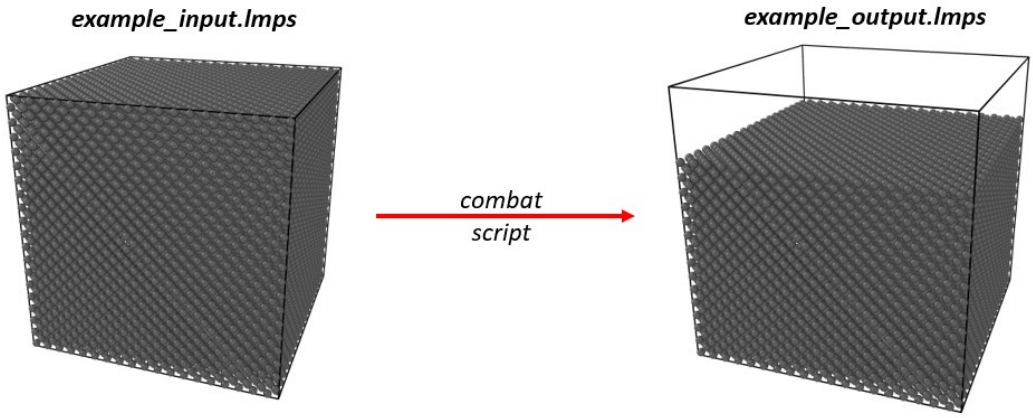


Fig. 2 Visualization of the conversion of a bulk FCC crystal to an FCC material with a surface at 75% of the z box length oriented perpendicular to the z dimension

2.2 Example 2: Production of a Random Ternary Alloy

The objective of this example is to convert an *.lmps* configuration file that contains a one-component single crystal with the diamond lattice structure to a randomly distributed ternary alloy with the same crystallographic structure. The initial structure can be easily generated in LAMMPS with a script similar to that given in Table A-1 of the Appendix. To avoid unnecessary complexity of the demonstration presented here, the three components (referred to arbitrarily as A, B, and C) are assumed to have the same lattice constant. A random number generator can be used to select atoms and then convert the atom types until a desired composition is reached. For this example, the initial diamond structure is defined to be composed purely of Component A, and the desired system is chosen to be a ternary alloy with the composition $A_{0.4}B_{0.2}C_{0.4}$.

The python random module, imported in Line 2 of Table 2, provides a basic random number generator that can be used repeatedly to progress toward the final ternary alloy until the target composition is attained. First, two variables are defined (Lines 6 and 7) containing the number of atoms that need to be converted to achieve the desired ternary alloy composition. To convert Type-A particles to Type-B particles, the random number generator produces values corresponding to particle indices that will be converted by adjusting the associated particle ["type"] of the data frame from a value of 1 (indicating Type-A) to a value of 2 (indicating Type-B). This conversion process is performed within a python "while loop" until 20% of all of the particles in the simulation cell have been converted to Type-B particles. Similarly, the process is repeated to convert Type-A particles to Type-C particles using a new python "while loop", which terminates when 40% of the Type-A particles are modified and change to Type-C particles. Note that the conversion nested within both "while loops" only happens when the random number generator

provides an index value for a particle that is Type-A (i.e., the particle ["type"] is a value of 1). If, by chance, the random number generator provides a value for the index of particle as Type-B or Type-C, a new random number will be generated, and the process will be repeated. An updated *.lmps* configuration file is then written upon reaching the target composition of $A_{0.4}B_{0.2}C_{0.4.}$ Figure 3 is a visualization of this process.

Table 2 Python script that uses *combat* to convert a bulk-diamond single crystal to a random ternary alloy of the user-defined composition

Line #	
1	import combat
2	import random
3	# read in system containing the diamond lattice structure
4	s = combat.System.from_data('example_input.lmps', atom_style = 'charge')
5	# use the total number of particles to determine the number of converstions to be made
6	conv_a_to_b = int(0.2*len(s.particles))
7	conv_a_to_c = int(0.4*len(s.particles))
8	# use a while loop to convert a (type== 1) particles to b (type==2) particles
9	while len(s.particles[s.particles.type == 2]) < conv_a_to_b:
10	# generate a random number for the attempted index and store it in a variable
11	hold_index = random.randint(1, len(s.particles))
12	# only attempt the conversion if the attempt atom is of type A (1)
13	if s.particles.loc[hold_index]['type'] == 1:
14	s.particles.at[hold_index,'type'] = 2
15	# The same process can now be repeated with conv_a_to_c
16	while len(s.particles[s.particles.type == 3]) < conv_a_to_c:
17	# generate a random number for the attempted index and store it in a variable
18	hold_index = random.randint(1, len(s.particles))
19	# only attempt the conversion if the attempt atom is of type A (1)
20	if s.particles.loc[hold_index]['type'] == 1:
21	s.particles.at[hold_index,'type'] = 3
22	
23	s.write_data('example_output.lmps', atom_style='charge')

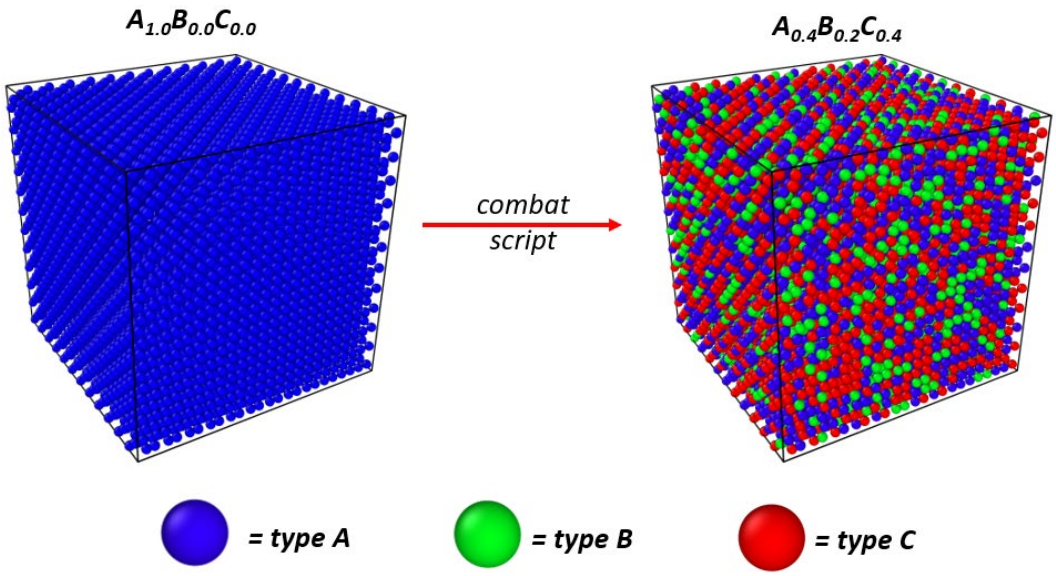
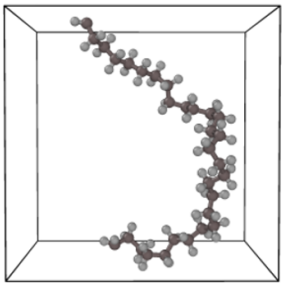


Fig. 3 Visualization of the conversion of an initial single-component (Type-A) diamond crystal converted to a randomly distributed ternary diamond alloy with a user-defined composition

2.3 Example 3: Bond Energy Calculation

For the purposes of pre- and postprocessing and system–system comparison, the magnitude of energy-related terms needs to be considered in the applied interatomic potential. This example focuses on calculating the total bond energy of a simple molecular system (i.e., a short polymer chain). As shown in Fig. 4, the system contains carbon-carbon (C-C) and carbon-hydrogen (C-H) bonds that need to be separately accounted for in the energy calculations. The sum of the total bond energy for each bond type is calculated and expressed as the total conformation bond energy in the *.lmps* configuration file. The example assumes that the particles are bonded with simple harmonic springs, which is one of the most common bond-stretching potentials. The equilibrium bond positions are taken to be 1.5 and 1.1 Å for the C-C and C-H bonds, respectively, chosen for demonstration purposes and not from any particular source, which are reasonable values for a C-C or C-H bond. In the interest of simplicity, both bond types will be assumed to have force constants of 250 kJ/mol. For illustration purposes for users who are less familiar with bonded system models, two sample calculations are provided: 1) a C-C bond with a 1.6-Å bond length, which results in an increased energy of 2.5 kJ/mol, and 2) a C-H bond with a 1.3-Å bond length, which produces an increased energy of 10 kJ/mol.



$$U_{total\ bond\ stretch} = \sum U_{C-C\ bond\ stretch} + \sum U_{C-H\ bond\ stretch}$$

$$U_{bond\ stretch} = k(\vec{r} - \vec{r}_o)^2$$

$U = energy$
$k = force\ constant$
$\vec{r} = bond\ length$
$\vec{r}_o = equilibrium\ bond\ length$

Fig. 4 Simulation cell containing a single polymer chain alongside the equations used to calculate the bond-stretching energy for the conformation presented

Similar to Examples 1 and 2, the *combat* python module is imported and an *.lmps* configuration file is read to create a system class object (see Table 3). Note that the *atom_style* keyword passed to the *from_data()* function is *'full'*, as opposed to *'charge'* in the previous examples, because the short polymer chain, being a bonded system, has additional atom attributes (e.g., the molecule identifier). The user is referred to the LAMMPS documentation for a list of possible atom style formats that *.lmps* data files can assume.

Table 3 Python script that uses *combat* to calculate the bond-stretching energy of a single polymer chain containing C-C and C-H bonds

Line #	
1	import combat
2	
3	# read in system containing the short polymer chain
4	s = combat.System.from_data('example_input.lmps', atom_style = 'full')
5	# calculate all bond lengths internally with combat
6	s.bond_lengths()
7	# create two different temporary data frames containing only particles of each type
8	temp_df1 = s.bonds[s.bonds.type == 1]
9	temp_df2 = s.bonds[s.bonds.type == 2]
10	# create new columns in each data frame with the squared distance from equilibrium
11	temp_df1['delta_length'] = (temp_df1['length'].values-1.5)**2
12	temp_df2['delta_length'] = (temp_df2['length'].values-1.1)**2
13	# create two variables that store the energy for each bond type
14	bond_stretching_energy_1 = 250.0*temp_df1['delta_length'].sum()
15	bond_stretching_energy_2 = 250.0*temp_df2['delta_length'].sum()
16	# sum the two energies and print the final bond stretching energy
17	final_stretching_energy = bond_stretching_energy_1 + bond_stretching_energy_2
18	print(final_stretching_energy)

After reading the *.lmps* configuration file, the *bond_lengths()* function is executed (Line 6 of Table 3) to compute the bond distances. This action requires no keyword arguments and internally calculates the bond lengths for all bonds stored in the bonds data frame. The values are then appended to a new column in the bonds data frame with a header "length" to be used for further calculations.

To simplify the complexity of the python code required for this example, two temporary data frames are created (temp_df1 and temp_df2) to exclusively contain bonds of a specific type (Lines 8 and 9). New columns corresponding to each bond type (in this case, the C-C and C-H bond types) are appended to each data frame and labeled with the column header "delta_length". These new columns contain the value of the displacement from the equilibrium position (see Lines 11–12). The sum of this column multiplied by the force constant is used to compute the total bond-stretching energy for all bonds of the same given type (Lines 14 and 15). Summing these C-C and C-H bond stretching energies gives the total bond-stretching energy of the entire system (Line 17) that is displayed to the user with the simple python print function (Line 18).

2.4 Example 4: Generating a Core-Shell Nanoparticle

Similar to the designs of Examples 1 and 2, this example begins with a single-phase bulk crystal and transforms it into a user-defined nanostructure with the use of the python commands given in Table 4. In particular, a 10-nm simulation cell containing a single-phase crystal with a body-centered-cubic (BCC) lattice is used to produce a spherical nanoparticle with a 5-nm diameter. The outer layer of this nanoparticle (1 nm) is then converted to a new atom type to form a concentric core-shell nanostructure. The key method employed to accomplish this task is to create a new particle attribute, r, for all of the particles in the system and append it to a new column in the particles DataFrame (Fig. 5a). The r attribute describes the radial distance of particles away from a chosen point, which is taken to be the center of the simulation cell for the sake of convenience.

Table 4 Python script that uses *combat* in conjunction with Numpy linear algebra functions to produce a concentric core-shell nanoparticle structure from a bulk BCC single crystal

Line #	
1	import combat
2	import numpy as np
3	# read in system containing the single crystal BCC lattice structure
4	s = combat.System.from_data('example_input.lmps', atom_style = 'charge')
5	# create a list of the geometric center of the simulation cell
6	geo_center = [(s.xhi-s.xlo)/2, (s.yhi-s.ylo)/2, (s.zhi-s.zlo)/2]
7	# replicate this list multiple times for computational efficiency
8	r = np.array([geo_center]*len(s.particles))
9	# create a new s.particles column with the distance from geo_center
10	s.particles['r'] = np.linalg.norm(s.particles[:,2:5].values - r, axis=1)
11	# remove all particles that are beyond 2.5 nm
12	s.particles = s.particles[s.particles.r <= 25]
13	# store the indices of all particles between 2 nm and 2.5 nm
14	hold = np.array(s.particles[(s.particles['r'] <= 25]) & (s.particles['r'] >= 15)].index)
15	# convert all particles with the stored indices to type 2
16	s.particles.at[hold, 'type'] = 2
17	# adjust particle indices to be continuous and starting at 1
18	s.particles.reset_index(drop=True, inplace=True)
19	s.particles.index += 1
20	
21	s.write_data('example_output.lmps', atom_style='charge')

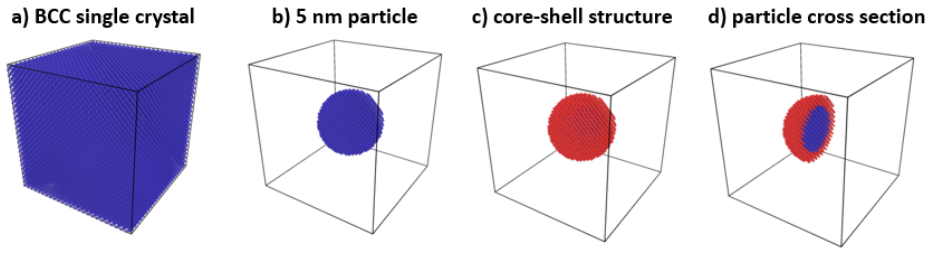


Fig. 5 Process visualization for the formation of a concentric core-shell nanoparticle: a) the initial BCC single crystal structure, b) 5-nm particle carved out of the bulk phase, c) core-shell spherical nanoparticle with a 1.0-nm-thick outer layer, and d) cross-sectional view of the core shell spherical nanoparticle

After reading the *.lmps* configuration file into *combat*, a list is created containing the x, y, and z coordinates for the geometric center of the simulation cell (Line 6 of Table 4). Their values are then replicated multiple times and inserted into a Numpy array that is the same length as the number of particles present (Line 8). This step is performed for the purpose of computational efficiency because it allows for the

distance between r and every particle to be calculated in a single vectorized command (Line 10 in Table 4). The particles data frame is then reduced to contain only those particles that have an r value equal to or less than 25 Å (Line 12), thus forming the 5-nm-sized core, as shown in Fig. 5b. The outer 10-Å layer of the atoms is then converted by selecting all particles with an r value greater than or equal to 15 Å (Line 14) and setting their atom type to 2 in order to form the concentric core-shell nanoparticle (Line 16). As a final step before writing a new *.lmps* configuration file, because particles were removed from the simulation cell, the particle indices are adjusted to be continuous and starting at 1 to meet the requirements of the LAMMPS software package (Line 18). The final core-shell nanoparticle structure is shown in Fig. 5c, with a cross-section image shown in Fig. 5d.

2.5 Example 5: Nanoparticle in a Gaseous Environment

Considering the diversity and complexity of possible chemical systems, it is often desired to conduct simulations for multi-component material systems; for instance, liquid mixtures, particles in gaseous environments, or the interface between two materials. This example highlights the use of the *populate()* function to combine multiple *.lmps* configuration files into a single system and assist in the preprocessing for different simulation setups. The employment of this function is similar to the popular software package PACKMOL.[8] The goal for this example is to place the core-shell nanoparticle generated in Example 3 into an environment containing oxygen and nitrogen gases. Three independent combat systems are created from *.lmps* files with labels s1, s2, and s3 (Lines 4–6 in Table 5) corresponding to the nanoparticle, a single oxygen molecule, and a single nitrogen molecule, respectively.

Each *combat* system has a *populate()* function that takes another *combat* system object as an argument and attempts to insert a specified number of copies of that system. For instance, if *s1.populate(s2)* is called, it would result in a single nitrogen molecule being added to the nanoparticle system. The insertion can be replicated multiple times as defined by the user using the *total_add* argument. For this example, the values of the *total_add* argument will arbitrarily be chosen to be 50 and 200 for nitrogen and oxygen, respectively. Unless otherwise specified, the particles that are being inserted are randomly rotated to minimize configurational bias within the system. Following rotation, the attempted insertion is made, and the new particle is tested for spatial overlaps with the pre-existing system. Overlapping criteria are user-defined through the *cutoff* argument of the *populate()* command, which sets the lower bound for a specific allowable interparticle spacing. If particles are found to be overlapping with any part of the pre-existing system, the attempted

insertion fails, and the process is repeated until the number of successful insertions is equal to the value of *total_add*. This example uses a default value of 1.0 Å cutoff, which is explicitly defined in the sample python code in Table 5 for clarity. (Interested readers are encouraged to refer to Appendix Section A.2 for additional information on the *populate()* function.) Figure 6 shows the end result after the two *populate()* functions are called. The *populate()* function, in principle, is capable of inserting a system of any size given that there is an appropriate amount of available space (see Appendix Fig. A-3 for an example involving polymer chains).

Table 5 Python script demonstrating use of *populate()* function to create a gaseous environment containing molecular oxygen and nitrogen gases around a spherical nanoparticle

Line #	
1	import combat
2	
3	# read in systems containing the particle and gas molecules
4	s1 = combat.System.from_data('example_input.lmps', atom_style = 'charge')
5	s2 = combat.System.from_data('oxygen.lmps', atom_style = 'charge')
6	s3 = combat.System.from_data('nitrogen.lmps', atom_style = 'charge')
7	# Add 200 oxygen molecules
8	s1.populate(s2, total_add = 200, cutoff = 1.0)
9	# Add 50 nitrogen molecules
10	s1.populate(s3, total_add = 50, cutoff = 1.0)
11	# Add an additional 50 oxygen molecules
12	
13	s.write_data('example_output.lmps', atom_style='charge')

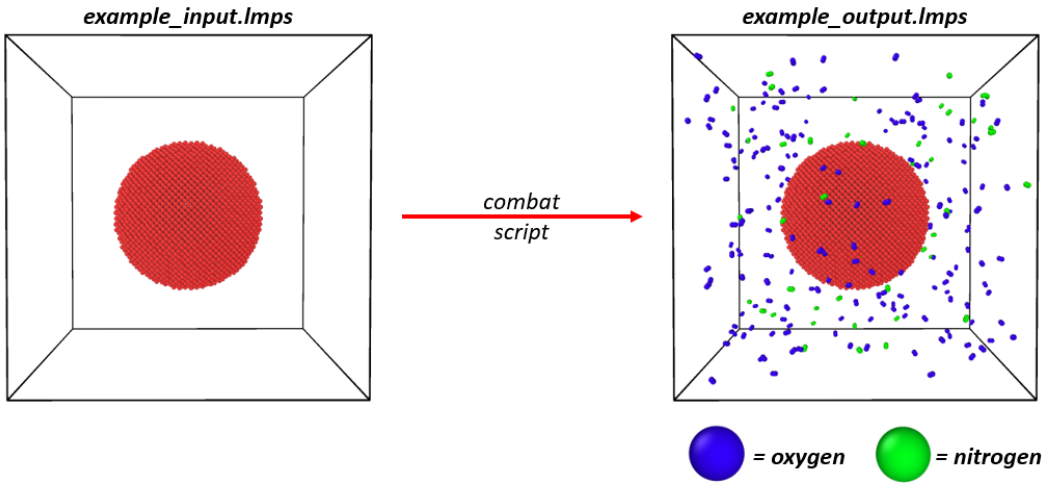


Fig. 6 Visualization of the *combat populate()* function with the addition of molecular oxygen and nitrogen gases to a system containing a spherical nanoparticle

2.6 Example 6: Creating a Spatial Composition Histogram

The *combat* toolkit contains functions for routine postprocessing analyses. Many of these postprocessing functions are housed in the *combat_analysis* module to be imported separately alongside *combat*. The functions housed within *combat_analysis* take the form of static functions (similar to Numpy), meaning *combat* systems are passed to the function alongside other keyword arguments. The purpose of this example is to use the *combat_analysis spatial_histogram()* function to analyze the spatial variation in the composition for the randomly distributed ternary alloy produced in Example 2. The procedure shown in Table 6 includes importing *combat*, *combat_analysis*, and reading in the *.lmps* configuration file that was output in Example 2 (Lines 1–4).

Table 6 Python script using the *combat* and *combat_analysis* modules to produce a spatial histogram describing the relatively uniform alloy composition along the z direction

Line #	
1	import combat
2	import combat_analysis
3	# read in the output system from example 3
4	s = combat.System.from_data('example_3_output.lmps', atom_style = 'charge')
5	# call the spatial histogram function, the \ denotes a continuation to the next line of code
6	combat_analysis.spatial_histogram(s, identifier='type', dimension='z',
7	\ binstart=s.zlo, binstop=s.zhi, binsize=4.0, file_name='output.csv')

The purpose of a spatial histogram is to create bins along a dimension of the simulation cell and count the number of particles having a certain attribute. Passing the *identifier* argument to the *spatial_histogram()* function is a requirement because it specifies the particle attribute that is considered for bin incrementing. Because this example aims to assess spatial variation in the composition, the *'type'* keyword will be passed as the identifier. In practice, the identifier argument can take the form of any column header present in the particles data frame. The desired dimension for the spatial histogram is then specified using the *dimension* argument. Here, this example is passing *'z'* to indicate the z dimension. The binning arguments are also passed to inform the beginning (*binstart*) and end (*binstop*) locations of the histogram, and the width of each bin (*binsize*). The *binstart* and *binstop* arguments are set to be the lower and upper box dimensions, and the *binsize* argument is chosen to be 2.0 Å. Finally, the *file_name* argument is passed to indicate the name of the output *.csv* file, which is a convenient file format that can be readily graphed in any common graphing software. Figure 7 shows the composition spatial histogram graph for the ternary alloy from Example 2. The approximate randomly distributed composition is clearly exhibited in the figure through the relatively comparable numbers of particles along the z coordinate axis.

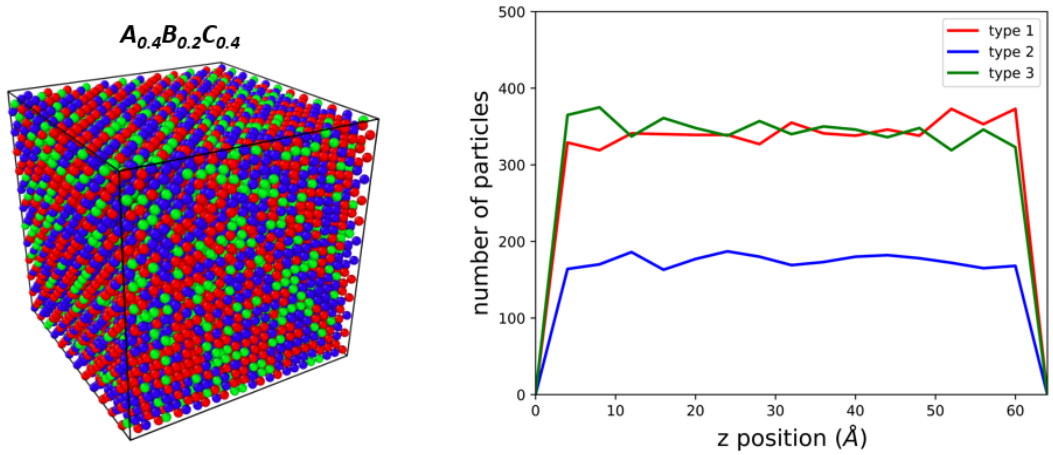


Fig. 7 Visualization of a random ternary alloy system and the graphed results from the *.csv* file generated by the *spatial_histogram()* function

2.7 Example 7: Distributions of a Bonded Molecular System

The ability to quickly calculate the bond distances and/or bond angles in a particle-based molecular system and determine their distributions is often sought in the modeling of molecular species. As an example, it would be beneficial to determine the C-C bond-length distribution in a simulation cell containing a polymer system for several reasons. From a preprocessing standpoint, this type of analysis allows modelers to judge whether the initial system preparation is present in an energetically unfavorable configuration. From a postprocessing perspective, the calculation of bond-length distributions enables the analysis of dynamic effects or the impact of external interactions on the conformational state of a molecule. With this in mind, the purpose of this example is to demonstrate that this type of analysis can be accomplished with *combat/combat_analysis* functions using minimal lines of a python code (Table 7). The system presented here is a relatively simple molecular model (i.e., a short polymer chain) consisting of carbon and hydrogen atoms only (Fig. 8a).

Table 7 Python script using the *combat* and *combat_analysis* modules to produce histograms for routine bond-length distribution analysis

```
Line #
1  import combat
2  import combat_analysis
3  # read in the molecular hydrocarbon system (polymer chain)
4  s = combat.System.from_data('example_7.lmps', atom_style = 'full')
5  # call function to calculate bond lengths, angle thetas, and dihedral phis
6  s.bond_lengths()
7  s.angle_thetas()
8  s.dihedral_phis()
9  # histogram the bond lengths from 1 to 1.6 with a bin width of 0.01
10 combat_analysis.bond_histogram(s, binstart=1.0, binstop=1.6, binsize=0.01, file_name='bond.csv'
11 # histogram the angle thetas from 0 to 140 with a bin width of 20
12 combat_analysis.angle_histogram(s, binstart=0.0, binstop=140, binsize=20, file_name='angle.csv'
13 # histogram the bond lengths from 1 to 1.6 with a bin width of 0.01
14 combat_analysis.dihedral_histogram(s, binstart=0.0, binstop=180, binsize=20, file_name='dihedral.csv'
```

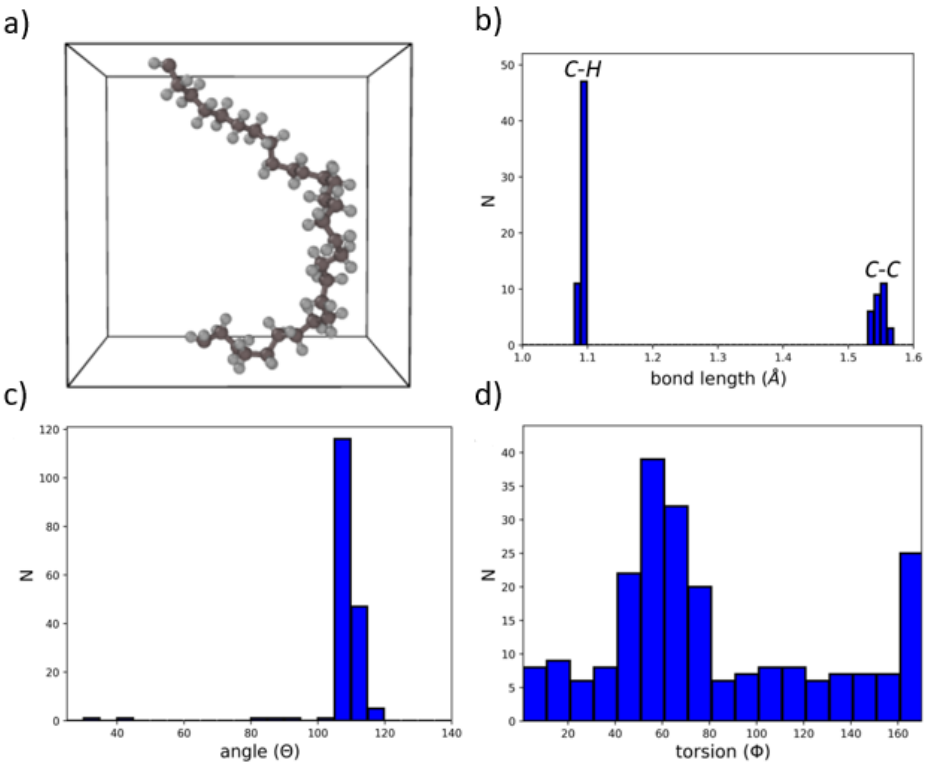


Fig. 8 Bond, angle, and torsional distributions of a model hydrocarbon molecular system. The y-axis (N) is the total number of bond-relevant molecular connectivity values found to fit in a bin: a) visualization of the model molecular system, b) 2-body bond distribution with C-H and C-C bonds labeled, c) 3-body angle distribution, and d) 4-body torsional distribution.

The *combat* toolkit is pre-equipped with functions to calculate common molecular bonding information. The system class has a function *bond_lengths()* to calculate the lengths of all bonds in the system and append them to a new column in the bonds data frame with a column header of "length". Similarly, the *angle_thetas()*,

dihedral_phis(), and *improper_chis()* functions can all be called to calculate angles, torsions, and impropers, respectively. No keyword arguments are required because these functions directly access the necessary information for calculations contained in the system class object. The hydrocarbon molecule used in this example does not have any improper angles, thus it is only necessary to call the system *bond_lengths()*, *angle_thetas()*, and *dihedral_phis()* class functions (Lines 6–8 of Table 7). Following the calculation of the relevant information for bonding in this molecular system, *combat_analysis* functions are called to create a histogram of the distribution for different terms in the molecular connectivity. These functions are shown in Lines 10–14 of Fig. 7 and require three types of arguments to be passed: a system class object, a set of binning arguments, and the output file name for saving the histogram data. The binning arguments consist of defining the values for the beginning (*binstart*) and end (*binstop*) points of the histogram, and the width of each bin (*binsize*). For this example, the histogram bounds are chosen to be 1.0–1.6 Å, 0°–140°, and 0°–180° for bonds, angles, and torsions, respectively. Histogram bin widths of 0.01 Å are used for bond lengths, and 20° for angles and torsions. Figures 8b–d show the output histogram data for the molecular system used in this example. This example demonstrates the power of *combat* for enabling routine bond, angle, and dihedral distribution analyses of a molecular system through minimal lines of python code that require only a handful of keyword arguments.

3. Conclusions

With continuous progress toward advanced computing hardware and refinements of efficient parallel algorithms, analyses through large-scale simulations have become an increasingly routine and important research practice. An evolving landscape of materials complexity also dictates that the development of in silico parallel characterization tools should have wide-reaching applicability and robust capability. The workflow and structure of *combat/combat_analysis* have been described and demonstrated in this report as a useful solution for several common molecular/material modeling processing tasks. The seven examples presented in this work highlight that common pre- and postprocessing tasks can be performed for selected diverse model systems with a minimal amount of python code. The *combat* and *combat_analysis* toolkits presented are shown to have a user-friendly platform for accomplishing the discussed tasks and are under continuous development to address the diverse set of challenges faced by researchers working on particle-based simulations at atomistic and coarse-grain scales. Currently, the flexibility of these tools and ease of implementation of high-performance python-based modules enables an enhanced ability for pre- and postprocessing of a

range of material systems across many scales. The example systems and processing tasks completed in this work provide practical demonstrations for a broad range of material models. Within our group, continuous expansion of the applicability of pre- and postprocessing tools to a broader range of system available in *combat* is ongoing.

4. References

1. Mattox TI, Larentzos JP, Moore SG, Stone CP, Ibanez DA, Thompson AP, Lísal M, Brennan JK, Plimpton SJ. Highly scalable discrete-particle simulations with novel coarse-graining: accessing the microscale. Molecular Physics. 2018;116(15–16):2061–2069.

2. Jaramillo E, Wilson N, Christensen S, Gosse J, Strachan A. Energy-based yield criterion for PMMA from large-scale molecular dynamics simulations. Physical Review B. 2012;85(2):024114.

3. Fortunato ME, Mattson J, Taylor DE, Larentzos JP, Brenna JK. Pre- and postprocessing tools to create and characterize particle-based composite model structures. Adelphi (MD): Army Research Laboratory (US); 2017 Nov. Report No.: ARL-TR-8213.

4. McKinney W. Data structures for statistical computing in Python. Proceedings of the 9th Python in Science Conference; 2010 June; Austin, TX. p. 51–56.

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

6. Jones E, Oliphant T, Peterson P. SciPy: open source scientific tools for Python. SciPy developers; 2001 [accessed 2019 Aug 16]. https://www.scipy.org.

7. van der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Computer and Information Science and Engineering. 2011;13(2):22–30.

8. Martínez, JM, Martínez, L, Packing optimization for automated generation of complex system's initial configurations for molecular dynamics and docking. J Comp Chem. 2003;24(7):819–825.

Appendix. Construction of a Single-Crystal Face-Centered Cubic (FCC) Structure

A.1 Construction of a Single-Crystal Face-Centered Cubic (FCC) Structure

Table A-1 presents an example script for generating an initial structure in Large-Scale Atomic/Molecular Massively Parallel Simulator (LAMMPS).

Table A-1 Sample LAMMPS input script used to generate an FCC crystal structure

```
units           metal
atom_style      charge
dimension       3
boundary        p p p
region          box block 0 1 0 1 0 1 units lattice
create_box      1 box
lattice         fcc 4.05 orient y 0 1 0 orient x 1 0 0 orient z 0 0 1
create_atoms    1 box
mass 1 26.98
replicate       20 20 20

write_data FCC.lmps
```

A.2 Description of the *populate()* Function

A.2.1 Randomizing Orientation of the Inserting System

For many systems, particularly simulations performed with reactive interaction potentials, there is a sensitivity to the orientation at which particles interact. To avoid orientational bias in the simulation of such systems, the *populate()* function randomly rotates the populating system to a new angle at each attempted insertion (by default), as shown in Fig. A-1. If a user would like to disable this process, they can achieve this by passing the keyword argument *"rotate = False"* when calling the *populate()* function. Otherwise, the system orientation is determined by applying a randomly determined *scipy.spatial.transfrom.Rotation* to the x, y, and z positions of the *combat.System.particles* DataFrame. In practice, the rotation transformation is achieved through matrix multiplication of three randomly constructed x, y, and z rotation matrices, all of which preserve the reference geometry of the populating system.

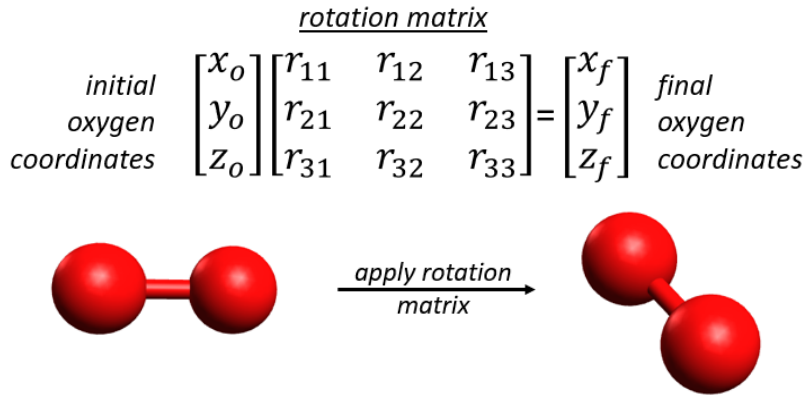


Fig. A-1 Visualization of the linear algebra operations applied to randomly orient a diatomic molecule prior to insertion, as implemented in *scipy.spatial.transform.Rotation*

A.2.2 Randomize System Position and Overlap Check

Following the randomization of the system orientation described in A.2.1, the next step of the *populate()* function is to add a random change in position $(\Delta x, \Delta y, \Delta z)$ such that the new position is within the box dimensions of the system that is being populated:

$$x_f \in [S1.xlo, S1.xhi] \qquad x_f = x_i + \Delta x$$

$$y_f \in [S1.ylo, S1.yhi] \qquad y_f = y_i + \Delta y$$

$$z_f \in [S1.zlo, S1.zhi] \qquad z_f = z_i + \Delta z$$

Following this random change in position, the system must be checked for any close overlaps (less than the defined cutoff) between the inserted molecular system and the system that is being populated. From a molecular simulation standpoint, it is desirable to avoid close overlaps because they can lead to energetically unfavorable configurations or unphysical geometries, which can either be unnecessarily difficult to equilibrate or result in unstable dynamics and invalid simulations. The overlap check is performed by comparing the number of neighbor particles of the original populating system before insertion to the number of the neighbor particles of the same system in its new environment; neighbor list details are described in Fortunato et al.[1] If the length is different, this indicates that the insertion is too close to an existing particle in the system and the attempted insertion is determined to be a failure.

[1] Fortunato ME, Mattson J, Taylor DE, Larentzos JP, Brenna JK. Pre- and postprocessing tools to create and characterize particle-based composite model structures. Adelphi (MD): Army Research Laboratory (US); 2017 Nov. Report No.: ARL-TR-8213.

A.2.3 Region-Specific Population

For many systems, particularly large-scale systems with spatially varying density, it is desirable to only attempt system insertions within regions that have an allowable volume to accommodate the populating system. To increase computational efficiency, the *populate()* function allows for region-specific insertion of molecules by specifying the dimension bounding keyword arguments: xlo, xhi, ylo, yhi, zlo, and zhi. As an example, by setting these arguments to the edge of the metal surface shown in Fig. A-2, the probability of successfully inserting a small molecule system is higher because only overlaps with other inserted molecules need to be considered. As a result of a higher probability of successful insertions, less internal iterations of the populate function will need to be performed, and thus less wall-clock time will be required to accomplish the desired number of insertions (Fig. A-3).

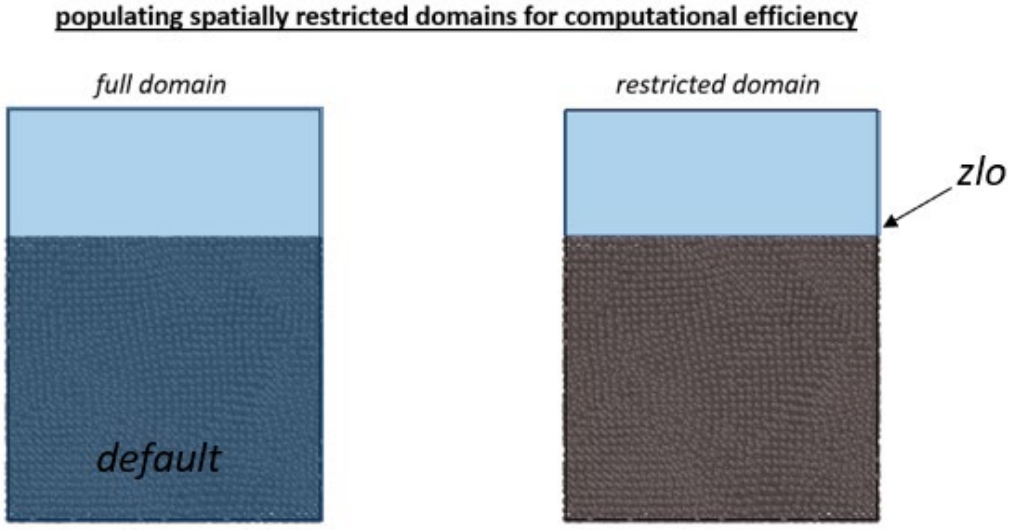


Fig. A-2 Visualization of the spatial domain within the simulation cell of the metal surface in which the *populate()* function attempts to insert molecules (shown in blue). The surface on the left represents full domain sampling (the default sampling), which results in a greater number of failed insertion attempts. The surface on the right shows reduced domain sampling, which results in failed insertions only if the attempted insertion is within the cutoff of a previously inserted molecular system.

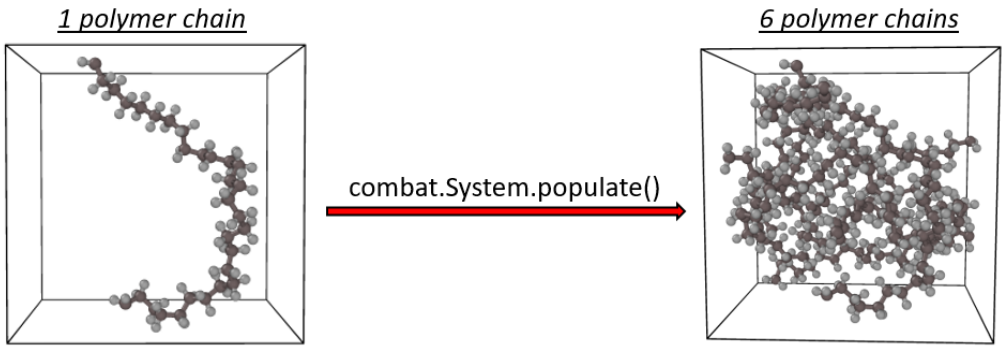


Fig. A-3 Demonstration of the ability of the *populate()* function to populate a system with molecules of significant size. In this case, a box containing a polymer chain with a degree of polymerization equaling 10 is populated with five replicas of the same system.

List of Symbols, Abbreviations, and Acronyms

.lmps	Large-Scale Atomic/Molecular Massively Parallel Simulator file format
ARL	US Army Combat Capabilities Development Command Army Research Laboratory
BCC	body-centered cubic
C	carbon
combat	Computational Model Builder and Analysis Toolkit (COMBAT)
CSV	comma-separated value
DOD	Department of Defense
FCC	face-centered cubic
H	hydrogen
HPCMP	High Performance Computing Modernization program
LAMMPS	Large-Scale Atomic/Molecular Massively Parallel Simulator
MD	molecular dynamics

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) CCDC ARL FCDD RLD CL TECH LIB

15 (PDF) DIR ARL FCDD-RLW-LB N J TRIVEDI J P LARENTZOS J K BRENNAN B RICE E F C BYRD B BARNES C WU J GOTTFRIED S IZVEKOV F DE LUCIA R PESCE-RODRIGUEZ FCDD-RLW-ME S COLEMAN FCDD-RLW-MG T SIRK J ANDZELM B RINDERSPACHER

1 (PDF) US NAVAL RSRCH LAB I SCHWEIGERT

4 (PDF) SANDIA NATIONAL LABS M WOOD S J PLIMPTON A P THOMPSON S MOORE

2 (PDF) HPCMP L DAVIS E EVANS K NEWMEYER

1 (PDF) UNIV OF FLORIDA DEPT OF CHEM C COLINA D ANSTINE

1 (PDF) MASSACHUSETTS INSTITUTE OF TECHNOLOGY M FORTUNATO

1 (PDF) PURDUE UNIVERSITY A STRACHAN

1 (PDF) UNIVERSITY OF MISSOURI T SEWELL

1 (PDF) JE PURKINJE UNIV M LISAL