8/30/2015 4:34:00 AM

Manual for Programmable Polymer Bonding 2

Software by

Zachary Kraus

Manual by

Zachary Kraus and Mari Mo

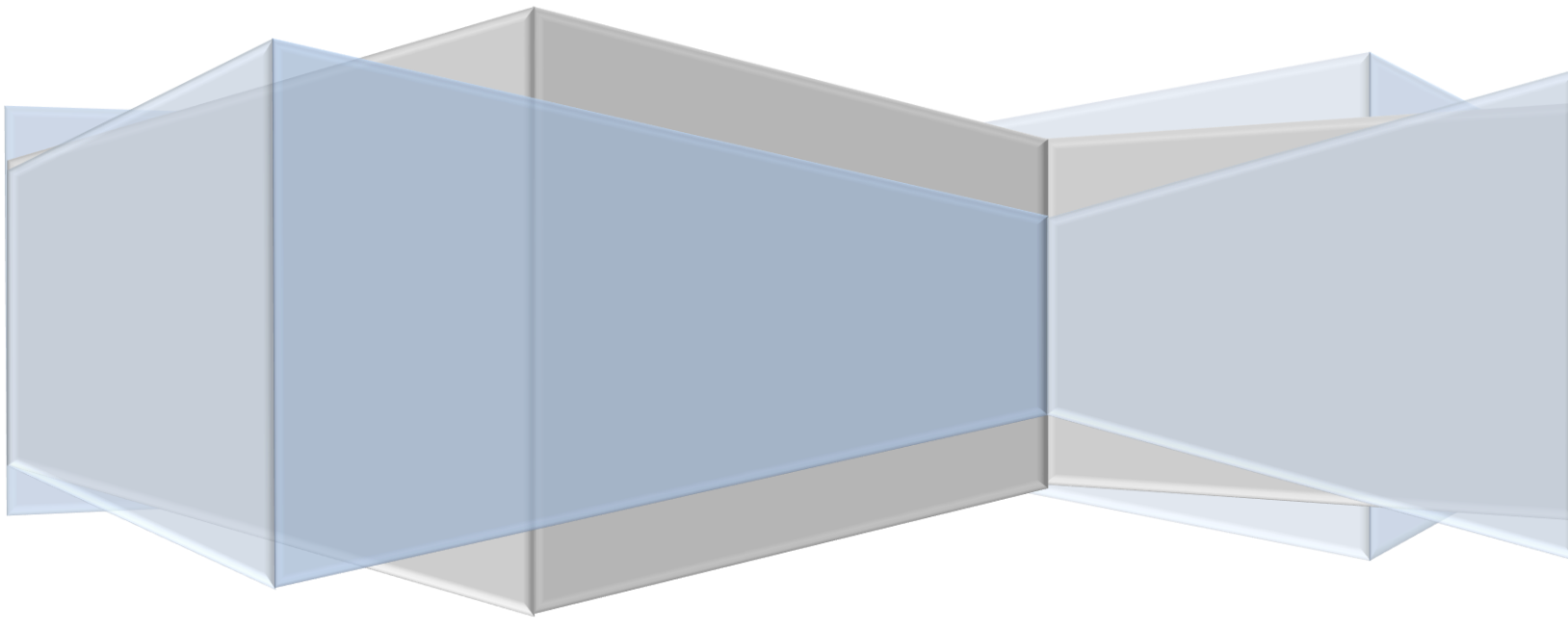

Table of Contents

Contents

1 Introduction

This software was developed to allow controlled bonding of polymers to ceramic nano particles for LAMMPS. The first version of the software used an object oriented approach where the key classes were “Lmpsdata”, “Lmpsmolecule”, and “particlesurface”. The Lmpsdata class reads, writes, extracts, and stores all of the data required for LAMMPS to run simulations. The Lmpsmolecule class only stores molecular style data which is used for manipulating molecules for bonding. The “particlesurface” class stores the data of the nanoparticle surface that is interacting/bonding with the polymer. These classes interact in such a way to allow the user to read and write data files, and implement reactions between the particle surface and the polymer chains.

The second version expands on the object oriented nature of the first version by using a dictionary of objects to store the data from the LAMMPS data files. Previously this information was stored in a list of lists. The difference in speed can be seen in Figure 1 below.

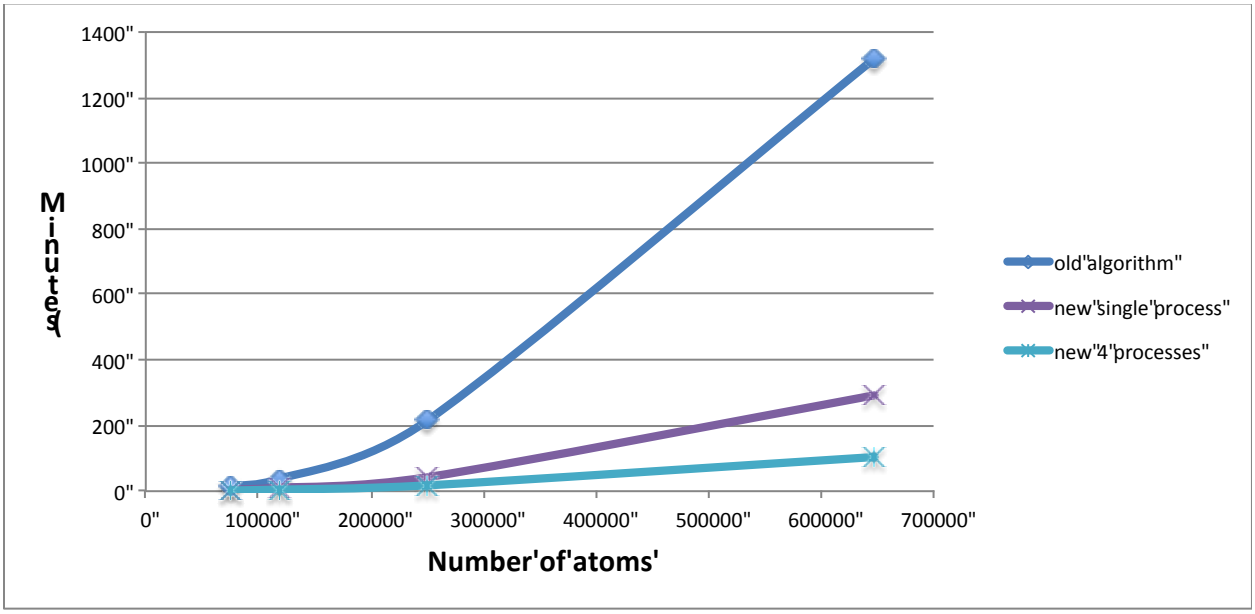

Figure 1.

Figure 1: shows how the old algorithm using four processes compares to the new algorithm with either a single process or four processes. The new algorithm and old algorithm at small number of atoms run in approximately

the same time. At larger number of atoms, the new algorithm is way more efficient and scales almost linearly compared to the old algorithm. Even though there are now more classes, only 5 classes will the user usually interact with. These are Lmpsdata, Molecule (used to be Lmpsmolecule), Nanoparticle (used to be particle surface), Density, and Reactor. These are the classes that will be demonstrated in the examples and explained in greater detail later in this manual.

2 Current Version: Version 2.0

2.1 Changes

1. Removed creation of XYZ files from the software
2. Requirement of having the nano particle at the origin has been removed
3. Allow multiple nano particles to be created.
4. Particle surface is now included in the Nano particle
5. Nano particles can now be either cubes or spheres
6. Density has been separated into its own object
7. Density can be calculated as shells, cubes or around a nano particle
8. Reactions are more generic compared to previous version

3 Previous Versions: *Version 1.0

3.1 Capabilities

1. Bonding ceramic nano particles to the polymer matrix
2. Inserting nano particles into openings in the polymer matrix
3. Calculating the density of spherical shells centered around the origin
4. Creating xyz files for VMD (REMOVED)
5. Create a polymer system with a specific molecular weight distribution (UNTESTED)

3.2 Limitations

*For large systems, code can run extremely slowly
*Python's multiprocessing has not been tested on distributed memory machines
*Designed to only handle one nano particle which is in the center of the simulation box

4 Terminology

Most of the terminology used below is explained in the LAMMPS manual or doc folder under the "read_data" command. The rest of the terminology comes from object oriented coding descriptions and procedural coding descriptions.

5 Sample Scripts

5.1 Example 1: Inserting a Nano Particle

In this example, a nano particle is inserted into a spherical whole in the polymer box. Also, the density is calculated, which illustrates the nano particle was successfully inserted. The velocities were deleted because the nano particle has no velocity and LAMMPS wont let you have a data file where only some of the atoms have velocities. In this case, the pair coefficients were also deleted because LAMMPS will not let you read in pair coefficients if the pair style is set to hybrid or hybrid/overlay.

```
from sys import path
path.append('/Users/zacharykraus/lmpsdata2')
del path[0]
import lmpsdata
from pylab import *

# Read in the alumina nanoparticle
f=open('alumina.pmma85','r')
nanoparticle=[]
for line in f:
        row=line.split()
        nanoparticle.append(row)

# Read in the polymer datafile
data=lmpsdata.Lmpsdata('full', 'pmma85data.finaleq3')
```

```
# Add in the mass information for the atom type 7, 8, and 9
# 7 is aluminum cation and 8 is oxgen anion and 9 is the surface aluminum cation
massinfo=[['7','26.981539'],['8','15.9994']] #this is not working
data.add_data(massinfo,'Masses')

#print data.masses

# Add in the alumina nanopartice
data.add_atoms(nanoparticle)

# Delete the data for the velocities and pair coefs
data.delete_body_data('Velocities')
data.delete_body_data('Pair Coeffs')

# Delete Velocities and Pair Coeffs from the keywords
data.body_keywords.remove('Velocities')
data.body_keywords.remove('Pair Coeffs')

# Write the defnsity calculations to the file testden.txt
r,rho=lmpsdata.Density.shells(data,.5,0,32.5)
plot(r,rho)
xlabel('distance (angstrom)')
ylabel('density (amu/angstrom^3)')
#legend(loc=1)
show()


# Write the new nanocomposite into a new datafile
data.write('pmma85_nano_composite_data.initial')
```

5.2 Example 2: Bonding of PMMA to an alumina nano particle.

In this example PMMA is bonded to an alumina nano particle with this

reaction:

$Al_2O_3 + O + 2[-CC(C)(COOC)-] \rightarrow 2AlO^+(CC(C)(COO))^- + 2O^- + 2C^+$

Before this script can be run, the polymer needs to be near or at equilibrium. The starting bonding distance used in this example was the cation-cation distance for alpha aluminum. This value can be increased by an angstrom or two to form the number of bonds required. In this script the value is increased by about an angstrom. For brevity purposes, I only included one of the three bonding cases shown in the top portion of this example.

Version A multiprocessing:

```
from sys import path
path.append('/Users/zacharykraus/lmpsdata2')
del path[0]
import lmpsdata, copy
data = lmpsdata.Lmpsdata('full', 'pmma90compositedata.initeq')

#copies pmma80compositedata.initeq to a new folder
directory='./bulk/'
data.write(directory+'pmma90_composite_data.initial', 0)

#add mass data for the bonded carbonyl type
massinfo=[['9','15.9994']]
data.add_data(massinfo,'Masses')

from multiprocessing import Pool
p = Pool(4)

#seperate nanocomposite data into polymer and nanoparticle portion
polymer = []
for i in range(1,31):
        polymer.append(p.apply_async(lmpsdata.Molecule, (data, i,)))
for i in range(len(polymer)):
        print 'creating molecule ', i
        polymer[i] = polymer[i].get()
        print 'finished producing molecule ', i
```

```
print 'creating nanoparticle'
nanoparticle = lmpsdata.Molecule(data, 65)
print 'finished with nanoparticle'

#add surface to nanoparticle using Nanoparticle class
particle = lmpsdata.Nanoparticle(nanoparticle, 'input', 'sphere', [0.0, 0.0, 0.0], 15, 1.94, ['type'],
[8])
print 'finished inputing the nanoparticle and producing a surface'

#setup copies of molecules for different reaction processes
polymer_crosslink = copy.deepcopy(polymer) #crosslink case is 2 bonds formed
polymer_weakbond = copy.deepcopy(polymer)  #weakbond case is 4 bonds formed
polymer_strongbond = copy.deepcopy(polymer) #strongbonds case is 8 bonds formed

#setup copies of particle for different reaction processes
particle_crosslink = copy.deepcopy(particle)
particle_weakbond = copy.deepcopy(particle)
particle_strongbond = copy.deepcopy(particle)

#setup different reactors
crosslink_reactors = []
weakbond_reactors = []
strongbond_reactors = []

#insert polymers into appropriate reactor
for i in range(len(polymer)):
        crosslink_reactors.append(lmpsdata.Reactor(polymer_crosslink[i]))
        weakbond_reactors.append(lmpsdata.Reactor(polymer_weakbond[i]))
        strongbond_reactors.append(lmpsdata.Reactor(polymer_strongbond[i]))

#find possible bonding between crosslink(polymer) and surface_crosslink
bondinglen=0
for reactor in crosslink_reactors:
        reactor.find_particle_bonding_locations(particle_crosslink, 'type', 6, 3.2, 2)
        if len(reactor._particle_bonding_information) !=
int(len(reactor._particle_bonding_information)/2.0)*2:
```

```
                i = len(reactor._particle_bonding_information) - 1
                del reactor._particle_bonding_information[i]
        print 'the length of the bonding information is ',
len(reactor._particle_bonding_information)
        bondinglen += len(reactor._particle_bonding_information)
bondinglen = bondinglen/float(len(crosslink_reactors))

#Bond crosslink(polymer) to particle_crosslink
count=1
for reactor in crosslink_reactors:
        print 'the bonding iteration is', count
        reactor.bond_to_ceramic_particle(particle_crosslink, [['type', 9], ['charge', -.7825]],
[['type', 1]])
        for j in range(len(reactor._particle_bonding_information)/2):
                print 'adding atom', j
                particle_crosslink.add_atom(['65', '8', '-.945'])
        count += 1

# join molecules to data
data.join(polymer_crosslink)

# add in the crosslinked nanoparticle to data_crosslink
data.add_atoms(particle_crosslink.atoms) #cannot use add_body_data since velocity structure
doesn't include the added atoms from the reaction

# delete the data for the velocities and pair coeffs from data_crosslink
data.delete_body_data('Velocities')

# delete the Velocities and Pair Coeffs from the keywords from data_crosslink
data.body_keywords.remove('Velocities')

# write the crosslink bonded nanocomposite into a new data file
directory='./crosslink/'
data.write(directory+'pmma90_composite_data.initial','crosslink test')
# write the crosslink bondinglen into a file in the crosslink directory
f=open(directory+'bondinglen.info','w')
```

```
f.write('{0}'.format(bondinglen))
f.close()
```

Version B single process:

```
from sys import path
path.append('/Users/zacharykraus/lmpsdata2')
del path[0]
import lmpsdata, copy
data=lmpsdata.Lmpsdata('full', 'pmma85compositedata.initeq')

#copies pmma80compositedata.initeq to a new folder
directory='./bulk/'
data.write(directory+'pmma85_composite_data.initial', 0)

#add mass data for the bonded carbonyl type
massinfo=[['9','15.9994']]
data.add_data(massinfo,'Masses')

#seperate nanocomposite data into polymer and nanoparticle portion
polymer = []
for i in range(1, 20): #change back to 1, 20 later
        print 'creating molecule ', i
        polymer.append(lmpsdata.Molecule(data, i))
        print 'finished producing molecule ', i
print 'creating nanoparticle'
nanoparticle=lmpsdata.Molecule(data, 20)
print 'finished with nanoparticle'

#Add surface to nanoparticle using Nanoparticle class
particle = lmpsdata.Nanoparticle(nanoparticle, 'input', 'sphere', [0.0, 0.0, 0.0] , 15, 1.94, ['type'],
[8])
print 'finished inputing the nanoparticle and producing a surface'

#setup copies of molecules for different reaction processes
polymer_crosslink = copy.deepcopy(polymer) #crosslink case is 2 bonds formed
```

```
polymer_weakbond = copy.deepcopy(polymer)  #weakbond case is 4 bonds formed
polymer_strongbond = copy.deepcopy(polymer) #strongbonds case is 8 bonds formed

#setup copies of particle for different reaction processes
particle_crosslink = copy.deepcopy(particle)
particle_weakbond = copy.deepcopy(particle)
particle_strongbond = copy.deepcopy(particle)

#set up different reactors
crosslink_reactors = []
weakbond_reactors = []
strongbond_reactors = []

#insert polymers into appropriate reactor
for i in range(len(polymer)):
        crosslink_reactors.append(lmpsdata.Reactor(polymer_crosslink[i]))
        weakbond_reactors.append(lmpsdata.Reactor(polymer_weakbond[i]))
        strongbond_reactors.append(lmpsdata.Reactor(polymer_strongbond[i]))

#find possible bonding between crosslink(polymer) and particle_crosslink
bondinglen=0 #bug in find_particle_bonding_locations where nano particle keys can be
duplicated even though they shouldn't be
for reactor in crosslink_reactors:
        reactor.find_particle_bonding_locations(particle_crosslink, 'type', 6, 3.2, 2)
        #need to ensure the number of bonds are in multiples of 2
        if len(reactor._particle_bonding_information) !=
int(len(reactor._particle_bonding_information)/2.0)*2:
                i=len(reactor._particle_bonding_information)-1
                del reactor._particle_bonding_information[i]
        print 'the length of the bonding information is', len(reactor._particle_bonding_information)
        print 'the final bonding information is', reactor._particle_bonding_information
        bondinglen += len(reactor._particle_bonding_information)
bondinglen = bondinglen/float(len(crosslink_reactors))
print 'the bonding length is ', bondinglen

#Bond crosslink(polymer) to particle_crosslink
```

```
count=1
for reactor in crosslink_reactors:
        print 'the bonding iteration is', count
        reactor.bond_to_ceramic_particle(particle_crosslink, [['type', 9], ['charge', -.7825]],
[['type', 1]])
        for j in range(len(reactor._particle_bonding_information)/2):
                print 'adding atom', j
                particle_crosslink.add_atom(['20', '8', '-.945'])
        count+=1

#join mololecules to data
data.join(polymer_crosslink)

# add in the crosslinked nanoparticle to data
data.add_atoms(particle_crosslink.atoms) #cannot use add_body_data since velocity structure
doesn't include the added atoms from the reaction

# delete the data for the velocities and pair coeffs from data_crosslink
data.delete_body_data('Velocities')

# delete the Velocities and Pair Coeffs from the keywords from data_crosslink
data.body_keywords.remove('Velocities')

# write the crosslink bonded nanocomposite into a new data file
directory='./crosslink/'
data.write(directory+'pmma85_composite_data.initial','crosslink test')
# write the crosslink bondinglen into a file in the crosslink directory
f=open(directory+'bondinglen.info','w')
f.write('{0}'.format(bondinglen))
f.close()
```

5.3 Example 3: Examining Density

In these examples the density is calculated using three different methods. These methods are shells, nanoparticle, and cubes. The first method calculates the density of the material in shells centered around the origin of the simulation box. The nanoparticle method calculates the density of the material in shells centered around the center of the nano particle. The cubes method focuses on a small section of the simulation box and breaks that area into cubes of a certain length. The density in these cubes is than calculated.

Version A shell and nano particle:

```
from sys import path
path.append('/Users/zacharykraus/lmpsdata2')
del path[0]
import lmpsdata
from pylab import *
def converttodata(restartfile,datafile):
        import os
        os.system('../../lammps/tools/restart2data ' + restartfile + ' ' + datafile)

# Convert restart file to datafile
converttodata('pmma85composite.initeq', 'pmma85compositedata.initeq')


# Read in the polymer datafile
data=lmpsdata.Lmpsdata('full', 'pmma85compositedata.initeq')

# Write the density calculations to the file testden.txt
r,rho=lmpsdata.Density.shells(data,.5,0,32.5)
plot(r,rho)
xlabel('distance (angstrom)')
ylabel('density (amu/angstrom^3)')
#xticks(arange(0,21,1))
#legend(loc=1)
```

```
#need to test nanoparticle density method
figure()
test_particle = lmpsdata.Nanoparticle(data, 'extract', 'sphere', [0.0, 0.0, 0.0], 15, 1.94, ['type'],
[8])
#for i in test_particle.surface:
#	print i , test_particle.atoms[i].write()
#f = open('test_particle.txt', 'w')
#for i, val in test_particle.atoms.items():
#	f.write('{0} {1}\n'.format(i, val.write()))
#print test_particle.shape
#print dir(test_particle)
r_new, r_rho = lmpsdata.Density.nanoparticle(test_particle, data, .5, 0, 17.5)
plot(r_new, r_rho)
lmpsdata.Density.write(r_new, r_rho, 'particle_test.txt')
xlabel('distance (angstrom)')
ylabel('density (amu/angstrom^3)')
show()
```

Version B shell and cube:

```
from sys import path
path.append('/Users/zacharykraus/lmpsdata2')
del path[0]
import lmpsdata
from pylab import *
def converttodata(restartfile,datafile):
	import os
	os.system('../../lammps/tools/restart2data ' + restartfile + ' ' + datafile)

# Convert restart file to datafile
converttodata('pmma85.finaleq3', 'pmma85data.finaleq3')


# Read in the polymer datafile
```

```
data=lmpsdata.Lmpsdata('full', 'pmma85data.finaleq3')
#data=lmpsdata.Lmpsdata('data.pmma100','full')

# Write the density calculations to the file testden.txt
r,rho=lmpsdata.Density.shells(data,.5,17.5,31)
lmpsdata.Density.write(r, rho, 'shell_test.txt')
r,rho = lmpsdata.Density.cubes(data, .5 , [18, 18, 18], [31, 31, 31], [22, 22, 22])
lmpsdata.Density.write(r, rho, 'cube_test.txt')
```

6 Classes and their methods

Classes			
lmpsdata	Coeffs	Coeffs_dihedral	Header_data
angle	Coeffs_improper	Bond	molecule
Coeffs_angle	Improper	Coeffs_bond	Body_data
Density	Nanoparticle	Pair_coeffs	reactor
atom	dihedral	Mass	velocity

Lmpsdata : stores the information associated with body data and header data keywords. Allows access to both body data and header data methods. Allows reading and writing of LAMMPS data files

Lmpsdata	
FUNCTION	DESCRIPTION
__init__(atom_style, file = '')	initializes body_data, and header_data. than attempts to read the file. atom_style is a string. file is a string that defaults to empty
read(file)	read LAMMPS data file. file is a string. if the string is empty this method does nothing. if the string does not match an existing file, an error occurs
write(file, comment_line = '')	write LAMMPS data file. file is a string. comment_line is a string that is written as the first line of the LAMMPS data file. The comment_line defaults to an empty string
join(data_list)	first deletes current data stored in the object. Than adds each body_data object stored in the list. Finally updates all body and header keywords
add_body_data(data)	adds a body data object. than updates all body and header keywords
add_atoms(atoms)	adds a dictionary of atoms or a list of strings. Than updates the keywords associated with atoms
add_data(data, keyword)	adds a dictionary, list of list of strings or a list. Than updates the correct keywords.

CLASSES

Angle :stores, reads and writes a LAMMPS angle

Angle	
FUNCTION	DESCRIPTION
__init__()	Initializes a LAMMPS angle
read(input, index)	converts a list of strings into the information stored in Angle input is a list of strings
write()	converts the information stored in Angle to a string returns the string

CLASSES

Coeffs : Stores reads and writes the coefficients

Coeffs	
FUNCTION	DESCRIPTION
__init__()	initializes LAMMPS coefficients
read(input, index)	copies the list of strings into the class input is a list of strings
write()	converts the list of strings stored in this class into one string returns the string

CLASSES

Coeffs_improper :stores, reads, and writes the coeffs for improper.additionally, can be used to match an improper_type object to a improper object the matching functionality will be implemented later

Coeffs_improper	
FUNCTION	DESCRIPTION
__init()	initializes the parent class Coeffs

CLASSES

Improper :stores, reads and writes a LAMMPS improper

Improper	
FUNCTION	DESCRIPTION
__init__()	initializes a LAMMPS improper
read(input, index)	converts a list of strings into the information stored in Improper input is a list of strings
write()	converts the information stored in Improper to a string returns the string

CLASSES

Nanoparticle: stores lammps body data, shape information and the surface of a nanoparticle

Nanoparticle	
FUNCTION	DESCRIPTION
__init__(data, method, shape, position, size, cutoff_distance, info=[], value=[], rotation=None)	Initializes nanoparticle by inserting lammps body data and building the nanoparticle surface and the shape. This class has three different methods of initialization: extract, input, and join. Extract only inserts the atoms into the nanoparticle that are inside the nanoparticle's shape. Input adds all atoms from data to the nanoparticle. Shape has two types: sphere and cube. Info is a list of atom attributes, and value is a list of those attribute's values Info and value is used to build the nanoparticle's surface Only atoms that meet one of the requirements in info and value are added to the nanoparticle's surface.
nearest_neighbor(atom, cutoff_distance)	Returns true if the atom is the cutoff distance from the surface of the shape.
delete_atoms(atom_keys)	Delete a list of atoms from the nano particle's surface and body data object contained in the nano particle
add_atom(info, method='random_position', pbi = [0, 0, 0])	Add an atom to the body data object contained in the nano particle and the surface if applicable. The random position method guarantees an atom will be added to the surface but the atom will be random unlike the known position method, which cannot guarantee the atom will be added to the surface.
delete_body_data(keyword)	Reimplementation of delete body data method which adds the capability to delete the nano particle surface
extract(data)	Reimplementation of extract method, which adds atoms that are inside the shape of the nano particle.
add_body_data(data)	Reimplementation of add body data which handles creating the nano particle surface
add_atoms(atoms)	Reimplementation of add atoms which handles creating the nano particle surface

CLASSES

Atom : stores, reads and writes the different types of atom in LAMMPS. Current valid types are: angle, atomic, bond, charge, colloid, full, and molecular. If your type is not listed here you can add the functionality to this class easily.

Note: the current design in this class will not work for hybrid atoms. If you need hybrid functionality, you will have to alter this class's design

Atom	
FUNCTION	DESCRIPTION
__init__(atom_style)	initializes a LAMMPS atom and sets the atom_style to be used. not all of the information stored in this class will be used by every atom_style
read(input, index)	converts a list of strings into the information stored in Atom input is a list of strings
write()	converts the information stored in atom to a string. uses atom_style to produce the correct string. returns the string. if the string contains <type 'int'> or <type 'float'>, either atom_style has accidentally been changed or a member used by atom_style was not assigned in the read method of this class

CLASSES

Density : contains methods for calculating the density from information stored in the body_data class and additionally in the header_data class if needed. also contains a method to write the calculated values to a file

Density	
FUNCTION	DESCRIPTION
shells(data, ringsize, init, final)	calculates the density by creating spherical shells from the initial radius to the final radius. the spherical shell's thickness is defined by ringsize. data must either be a body_data class or a class that inherited body_data. calculates the mass of particles within each spherical shell. calculates the volume of each spherical shell. calculates the density in each spherical shell. shells are centered around the origin of the simulation box
cubes(data, edge_length, init, final, origin = [0, 0 , 0])	Splits the simulation box into cubes where the density is calculated for each one.
nanoparticle(particle, data, spacing, init, final)	Splits the space around the nanoparticle into nanoparticle shaped shells. The density is calculated in each shell.
write(r, rho, file)	writes the information returned from density calculation methods to a file. r is the position information. rho is the density information. file is the location to write the informatin stored in r and rho to

CLASSES

Velocity : stores, reads and writes the different types of velocities in LAMMPS. Current valid types are: angle, atomic, bond, charge, colloid, full, and molecular. If your type is not listed here you can add the functionality to this class easily

Velocity	
FUNCTION	DESCRIPTION
__init__(atom_style)	initializes a LAMMPS velocity and sets the atom_style to be used. some of the information stored in this class has been commented out because they are not used by the currently implemented atom_styles
read(input, index)	converts a list of strings into the information stored in Velocity input is a list of strings
write()	converts the information stored in velocity to a string. uses atom_style to produce the correct string. returns the string

CLASSES

Coeffs_angle : stores, reads, and writes the coeffs for angle. additionally, can be used to match an angle_type object to an angle object. the matching functionality will be implemented later

Coeffs_angle	
FUNCTION	DESCRIPTION
__init__()	initializes the parent class Coeffs

CLASSES

Body_data :stores the information associated with LAMMPS body data keywords. creates and accesses objects associated with LAMMPS body data keywords.

Body_data	
FUNCTION	DESCRIPTION
__init__(atom_style)	initializes LAMMPS body data information, sets the atom_style used, initializes dictionaries used by the class to create and access dictionaries associated with body_data
create(keyword)	create an object associated with a body data keyword using the _factory dictionary to get the correct object type. keyword is a body data keyword
check_body_keyword(input)	checks if a body data keyword is located in input. input is a list of strings. returns whether a body data keyword was found in input and the resulting body data keyword as a string. The string is empty if no body data keyword is found
get_body_data(keyword)	returns the dicitionary or list associated with keyword. keyword is a body data keyword. the association is stored in_body_keyword_map.
delete_body_data(keyword)	deletes the items stored in the dictionary or list associated with keyword. keyword is a body data keyword. if the dictionary or list stored in this class has been directly assigned to another dictionary or list. this method will clear both dictionaries or lists which have been assigned. To avoid this, use copy or deepcopy methods when assigning a list or dictionary from this class to another list or dictionary
join(data_list)	Deletes the current data stored in the object. Than adds the data stored in the list of body_data objects to the body data object.
extract(data, atom_info, value)	extracts the atoms from data that have the matching value for atom_info.than extracts the associated information from angles, bonds, dihedrals, impropers, velocities, and masses. copies all coefficient information.data is a body_data object, atom_info is a string that corresponds to the name of a variable stored in the atom class. the type for value is dependent on atom_info and will either be an int or a float
find(keyword, info, value,	finds the data corresponding with keyword that has the matching value for

Body_data	
FUNCTION	DESCRIPTION
method)	info. returns either a list of keys or a new dictionary. keyword is a string that is a body_data keyword. info is a string that corresponds to the name of a variable stored in the class that corresponds to the given keyword. the type for value is dependent on info and will either be an int, a float or a string. method is a string that is either "keys" or "dict" which correspond to the method returning a list of keys or a new dictionary. If keyword is 'PairIJ' or 'Pair Coeffs' this method returns a list of Pair_coeffs objects.
search(keyword, info, values, method)	finds the data corresponding with keyword that has the matching values for info. returns either a list of keys or a new dictionary. keyword is a string that is a body_data keyword. info is a string that corresponds to the name of a variable stored in the class that corresponds to the given keyword. the type for values is dependent on info and will either be a list of ints, a list of floats or a list of strings. method is a string that is either "keys" or "dict" which correspond to the method returning a list of keys or a new dictionary. If keyword is 'PairIJ' or 'Pair Coeffs' this method returns a list of Pair_coeffs objects
add_body_data(data)	Adds the data stored in a body data object to this body data object
add_atoms(atoms)	add a dictionary of atoms or a list of list of strings to the body data dictionary of atoms. if a list of list of strings is given its converted to a dictionary. returns a dictionary containing the old and new keys. the old keys are are the atom ids of the input atoms. the new keys are the atom ids of the atoms added to the body data dictionary of atoms. If an incompatibility is detected between the atom_style of atoms and the body data dictionary of atoms, a runtime error is raised.
add_data(data, keyword)	add a dictionary or list of list of strings or a list to the corresponding item stored in body_data. if a list of list of strings is given its converted to a dictionary or list depending on keyword. if an incompatibility is detected between the atom_style of velocities and the body data dictionary of velocities, a runtime error is raised. For coefficients runs a method to ensure duplication of data does not occur. For other data types, there is no check for duplication

CLASSES

Coeffs_bond :stores, reads, and writes the coeffs for bond. additionally, can be used to match an bond_type object to a bond object.

Coeffs_bond	
FUNCTION	DESCRIPTION
__init()	initializes the parent class Coeffs

CLASSES

Dihedral :stores, reads and writes a LAMMPS dihedral

Dihedral	
FUNCTION	DESCRIPTION
__init__()	initializes a LAMMPS bond
read(input, index)	converts a list of strings into the information stored in Dihedral input is a list of strings
write()	converts the information stored in Dihedral to a string returns the string

CLASSES

Mass :stores, reads and writes a LAMMPS mass

Mass	
FUNCTION	DESCRIPTION
__init__()	initializes a LAMMPS mass
read(input, index)	converts a list of strings into the information stored in Mass input is a list of strings
write()	converts the information stored in Mass to a string returns the string

CLASSES

Pair_coeffs :stores, reads, and writes the coeffs for pair. Unlike the other children classes of Coeffs, This class will not add any extra behaviour.The seperate class is just here to easily identify that the class is for pairs of atoms

Pair_coeffs	
FUNCTION	DESCRIPTION
__init__()	initializes parent class Coeffs

CLASSES

Bond :stores, reads and writes a LAMMPS bond

Bond	
FUNCTION	DESCRIPTION
__init__()	initializes a LAMMPS bond
read(input, index)	converts a list of strings into the information stored in Bond input is a list of strings
write()	converts the information stored in Bond to a string returns the string

CLASSES

Coeffs_dihedral :stores, reads, and writes the coeffs for dihedral. Additionally, can be used to match an dihedral_type object to a dihedral object.the matching functionality will be implemented later

Coeffs_dihedral	
FUNCTION	DESCRIPTION
__init()	initializes the parent class Coeffs

CLASSES

Header_data :stores, reads and writes data in header lines from LAMMPS data files

Header_data	
FUNCTION	DESCRIPTION
__init__()	initializes the data stored in header lines and creates a dictionary relating the header keywords to the header data
check_header_keyword(input)	checks if a header keyword is located in input. input is a list where the header keyword, if stored, will be contained in the last half of the list. returns true if a header keyword was found in input and the resulting header keyword as a string. The string is empty if no header keyword is found
get_header_data(keyword)	returns the list or value associated with keyword. keyword is a header keyword. the association is stored in header_keyword_map.
set_header_data(keyword, value)	sets the list or value associated with keyword. keyword is a header header keyword. the association is stored in _header_keyword_map
read(input, keyword)	converts a list of strings into information stored in header_data. the information corresponds to keyword. input is a list of strings
write(keyword)	converts the information corresponding to keyword to a string. keyword is a header keyword.

CLASSES

Molecule :stores the body_data associated with a specific molecule

Molecule	
FUNCTION	DESCRIPTION
__init__(data, molecule_id)	builds the molecule by creating a new body_data object with the atom_style from the passed in data. the new body_data object extracts the information from data that correspond to molecule_id. data is a body_data object. molecule_id is an int that corresponds to a molecule number stored in data

CLASSES

Reactor :contains methods for modifying the objects stored in body_data. Additionally,allows the storage of a body_data object that is allowed to interact with other body_data objects

Reactor	
FUNCTION	DESCRIPTION
__init__(data)	initializes the reactor object with a stored body_data object. Data is the body_data object to store in the reactor.particle_bonding_information is a list of lists where [i][0] is the atom key in the reactor and [i][1] is the atom key in a particle object
change_atom_num(data, old_new_keys)	Uses a dictionary of old new keys to convert the atom numbers stored in a body data object from the old values to the new ones.
delete_atoms(data, keys)	Deletes information in body data related to the list of atom numbers in keys
add_atom(data, info)	Adds an atom to a body data object. Info is list of strings read by the atom object.
find_connected_atoms(data, atom_keys, connection_rules)	Find all atoms in the body data object (data) that are connected to the list of atoms in the atom keys. The connection rules parameter defines the rules used for selecting a connection between atoms in data and the atoms in atom keys.
modify(data, keyword, index, info, value)	Changes a value stored in the body data object(data) dictionary that corresponds to keyword. The value changed in the dictionary is at key position and has a member name called info.
find_particle_bonding_locations(particle, info, value, cutoff_distance, bond_number)	Finds locations for the molecule to bond to the particle.
bond_to_ceramic_particle(particle, reactor_modifications, reactor_delete_rules)	Bonds molecules to the particle using atom modification rules and deletion rules.

CLASSES

7 Needed Improvements

Capability to filter all molecules in one step instead of filtering out each individual molecule separately
Capability to do additive bonding, currently can only do subtractive bonding

Developer's Note:

This is my last version I will be developing for this project. I now have too many real life responsibilities now that I am out of school and too many other projects that have already taken priority over this one. I am hoping someone will chose to pick up developing this project and continue to add features and functionality.
I would like to thank the core developers of LAMMPS for making such a great product that is used worldwide. I would also like to thank them for adding this code as a LAMMPS utility.

8 Reference

If you need a reference to explain both the theory and steps behind the current reactor in the software, use the reference below.
"Computational Tools for Preliminary Material Design of Metals and Polymer-Ceramic Nano Composites", Zachary Kraus, May 2014, Georgia Institute of Technology.