Programmable Polymer Bonding

Table of Contents

Introduction

This software was developed to allow controlled bonding of polymers to ceramic nano particles for LAMMPS. The software uses an object oriented approach where the key classes are “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.

Current Version: Version 1.0

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
5. Create a polymer system with a specific molecular weight distribution (untested)

Limitations:

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

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.

Sample Scripts

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.

```
import lmpsdata
from pylab import *

# Read in the alumina nanoparticle
f=open('alumina.pmma80','r')
nanoparticle=[]
for line in f:
    row=line.split()
    nanoparticle.append(row)

# Read in the polymer datafile
data=lmpsdata.Lmpsdata('pmma80data.finaleq3','full')

# Add in the mass information for the atom type 7, and 8
# 7 is aluminum cation and 8 is oxygen anion
massinfo=[['7','26.981539'],['8','15.9994']]
data.adddata(massinfo,'Masses')

# Add in the alumina nanopartice
data.addatoms(nanoparticle,False)

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

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

# Write the density calculations to the file testden.txt
r,rho=data.density(.5,0,20)
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('pmma80_nano_composite_data.initial',1)
```

Example 2: Bonding of PMMA to a 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.

```
import lmpsdata, copy
data=lmpsdata.Lmpsdata('pmma80compositedata.initeq','full')

#copies pmma80compositedata.initeq to a new folder
directory='./bulk/'
data.write(directory+'pmma80_composite_data.initial', 0)



#orders atomdata before doing bonding procedure.
data.atomorder()

#seperate nanocomposite atoms into polymer and nanoparticle portion
# Note:Only the atom structures will be present in these variables.
polymer=lmpsdata.molecules(data,1,5)
nanoparticle=lmpsdata.molecules(data,6,6,'atom')

#seperate nanoparticle into its particle surface
surface=lmpsdata.particlesurface(nanoparticle[0], 1.94, 8, 'full')
surface.createxyz('alumina_surface_initeq.xyz',data)

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

#setup copies of surface for different reaction processes
surface_crosslink=copy.deepcopy(surface)
surface_weakbond=copy.deepcopy(surface)
surface_strongbond=copy.deepcopy(surface)

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

#setup copies of data for different reaction processes
```

```
data_crosslink=copy.deepcopy(data)
data_weakbond=copy.deepcopy(data)
data_strongbond=copy.deepcopy(data)


#find possible bonding between crosslink(polymer) and
surface_crosslink
bondinglen=0
for molecule in crosslink:
    molecule.findparticlebondingpoints(surface_crosslink,6,3.2,2)
    #need to ensure the number of bonds are in multiples of 2
    if len(molecule.bondinginformation)!
=int(len(molecule.bondinginformation)/2.0)*2:
           i=len(molecule.bondinginformation)-1
           del molecule.bondinginformation[i]
    print 'the length of the bonding information is',
len(molecule.bondinginformation)
    bondinglen+=len(molecule.bondinginformation)
bondinglen=bondinglen/float(len(crosslink))


#Bond crosslink(polymer) to surface_crosslink
count=1
for molecule in crosslink:
    print 'the iteration is', count
    molecule.bondtoparticle(surface_crosslink,1,'9','-.7825')
    #add 1 atom to surface_crosslink per 2 atoms bonded
    for j in range(len(molecule.bondinginformation)/2):
           surface_crosslink.addatom(8,-.945,5)
    count+=1

# extract the crosslink surface
surface_crosslink.extractparticle()

# extract molecule informtaion to data_crosslink
data_crosslink.extractmolecules(crosslink)

# add in the crosslinked nanoparticle to data_crosslink
data_crosslink.addatoms(surface_crosslink.particle)

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

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

# write the crosslink bonded nanocomposite into a new data file
```

```
directory='./crosslink/'
data_crosslink.write(directory+'pmma80_composite_data.initial',1)
# write the crosslink bondinglen into a file in the crosslink
directory
f=open(directory+'bondinglen.info','w')
f.write('{0}'.format(bondinglen))
f.close()
```

Methods by class

Notes: Not all of the methods are included in this guide. The methods I have left out are meant to be private methods, but as all methods in python are public, they can still be accessed. Therefore, to avoid people including them in Python scripts and possibly causing unintended behavior, I will just avoid placing them in the guide.

All methods using atom information are written assuming the image flags are included. If you don't use the image flags, this software may crash.

Lmpsdata(file,atomtype)

Initiates the class and reads from "file" which is a LAMMPS data file. All information from the read data file is stored in this class. In order to utilize the atom information later, the atom style must be assigned from "atomtype".

write(file, modflag)

Writes the information stored in this class out to a LAMMPS data file. The modflag allows the user to control whether certain header data will be calculated or written to the data file as is. The modflag set to "0" accesses the portion of the method which writes out all information to the data file unchanged, but the modflag set to "1" accesses the portion of the method which calculates certain header data. The header data that can be calculated are the number of atoms, bonds, angles, dihedrals, impropers and their number of types.

atomorder()

This method organizes the atoms by atom id from least to greatest and allows the bonding algorithm to always attempt to bond the first and last bondable atom on the molecule.

addatoms(atoms,retlist=false)

This method adds "atoms" to the atoms already stored in the "Lmpsdata" class. If "retlist" is set to "false" the method returns nothing. If "retlist" is set to "true" the method returns a list of the modified atom-ids of the added atoms.

adddata(data, keyword)

This method adds "data" to the information already stored in the "Lmpsdata" class. The specific structure the data is added to is determined by the "keyword" given. Only a body keyword input will result in data being added.

deletebodydata(keyword)

This method empties the corresponding structure of the “keyword”. The “keyword” must be a body keyword for the method to work.

extractmolecules(molecule)

Extracts the information contained in “molecule” to this class. The input, “molecule”, is a list of “Lmpsmolecule” objects. This method should be used with care, because the method empties the “Lmpsdata” class’s structures which correspond to the keywords stored in the first “Lmpsmolecule” object. Note, if the keywords in the “Lmpsmolecule” objects are different, this may cause this method to not function properly. If the program requires multiple lists of “Lmpsmolecule” objects, only the first list being added to this class can use this method. The rest of the lists must be added manually using the “addatoms” and “adddata” methods. Otherwise, important information will be lost from the class’s structures.

density(ringsize, init, final, file=’ ‘)

Creates spherical shells from the “init” radius to the “final” radius. The thickness of these shells are defined by the “ringsize”. The method than calculates the density in each shell. The resulting list of radii and densities are either returned to the calling script or printed to a file. The default option is to return the values to the calling script. To print the values to a file, set the desired destination with the “file” variable.

createxyz(file, routine=‘mass’, values=None)

Stores the atom information from the class in a xyz formatted file specified by “file”. This format is readable by VMD. There are two different algorithms used by “createxyz” for this conversion. One uses the masses of the atoms and the other uses the atomtype of the atoms. The mass method is the default method. In this method the mass of the atom is converted to its corresponding element number in the code. Currently, the code only contains this conversion for carbon, oxygen and aluminum. The masses used for these materials are formatted to four decimal points and the atom’s mass must exactly match in order for a conversion to occur. The atomtype method can be used by setting routine to ‘atomtype’ and values to a list of element numbers. The atomtype is used to access the list at the index atomtype-1.

particlesurface (particle, cutoff, atomid, atomtype, shape=‘sphere’)

Initiates the class and stores the “particle”’s atoms. Then produces the particle’s surface using the “cutoff”, “atomid”, “atomtype”, and “shape”. Currently, only one shape of the nano particle is supported, a sphere. The current implementation assumes the nano particle is centered around the origin and finds the atom with atomtype matching “atomid” which is the maximum distance away from the center. The algorithm than fills all atoms with atomtype matching “atomid” which are the “cutoff” distance away from the maximum distance. The “atomtype” input corresponds with the atomstyle which is required for all distance calculations and atom manipulations.

addatom(atomtype, charge=None, moleculenum=None)

This method adds an atom to the surface of the nano particle between the cutoff distance and the maximum distance. This also adds an atom to the particle's atoms stored in this class. The "atomtype" corresponds to the atomtype of the added atom. The "charge" and "moleculenum" have the default setting of None. The "moleculenum" corresponds with the molecule number. If the atom style of this class requires either a charge or a molecule number and the default settings are used, undefined behavior may occur because the current code does not check if these variables are using the default. This method also adds image flags automatically at the end of the atom information. These flags are all set to zero.

extractparticle()

This method removes atoms from the particle surface which have been marked as being replaced during the bonding process. These atoms are also removed from the particle's atoms stored in the class.

createxyz(file, data, routine='mass', values=None)

Stores the surface atom information from the class in a xyz formatted file specified by "file". This format is readable by VMD. There are two different algorithms used by "createxyz" for this conversion. One uses the masses of the atoms and the other uses the atomtype of the atoms. The mass method is the default method. In this method the mass of the atom is converted to its corresponding element number in the code. Currently, the code only contains this conversion for carbon, oxygen and aluminum. The masses used for these materials go out to four decimal points and the atom's mass must exactly match in order for a conversion to occur. The atomtype method can be used by setting routine to 'atomtype' and values to a list of element numbers. The atomtype is used to access the list at the index atomtype-1. The "data" input is a "Lmpsdata" class which is required for the mass algorithm to work. But even if your not using that algorithm the input is still required. Note: if you are trying to create a xyz file after the surface has bonded, the particle will need to be extracted than reinserted into the "particlesurface" class. Afterwards, the "createxyz" method can be run.

Lmpsmolecule (moleculenum, data, method)

Initiates the class and stores the molecular information from "data". The input "data" is a "Lmpsdata" class. The molecular information consists of atoms, angles, bonds, dihedrals, impropers and velocities. There are two "methods"; the default method incorporates all the molecular data. The other method incorporates only the atom data and can be accessed by setting method to 'atom'.

deleteatoms(atomnumbers, atomid)

Algorithm finds all atoms in the "Lmpsmolecule" class bonded to "atomnumbers" in the direction of "atomid". The "atomnumbers" is a list of atom ids in the "Lmpsmolecule" class where the nano particle surface is forming bonds. The "atomid" can either be a list of atom id values or a single atom type. The "atomid" is used to find the direction of deletion for the algorithm. This algorithm can be used to create a polymer system with a

specific molecular weight distribution; though, this use for the algorithm has not been tested.

findparticlebondingpoints(particle, atomid, cutoffdistance, bondnumber)

This method finds and stores the bonding points between the “Lmpsmolecule” object and the “particle”, a “particlesurface” object. Around the particle is a bonding region where atoms in the molecule with the correct atom type, “atomid”, can form bonds with the nano particle surface. The thickness of this region is defined by the “cutoffdistance”. The max number of possible bonds formed is defined by the “bondnumber”; though, this number may not be reached.

bondtoparticle(particle, atomid, newid, newcharge)

This method bonds the molecule to the particle surface using the previously found bonding points. The “particle” is a “particlesurface” object. The “atomid” can either be a list of atom id values or a single atom type. The method changes the bonding atom in the molecule to have “newid” as the atomtype and “newcharge” as the charge. The method than uses the “atomid” to run the “deleteatoms” method. Finally, the method communicates with the “particle” which atoms on the particle surface need removing.

createxyz(file, data, routine=‘mass’, values=None)

Stores the surface atom information from the class in a xyz formatted file specified by “file”. This format is readable by VMD. There are two different algorithms used by “createxyz” for this conversion. One uses the masses of the atoms and the other uses the atomtype of the atoms. The mass method is the default method. In this method the mass of the atom is converted to its corresponding element number in the code. Currently, the code only contains this conversion for carbon, oxygen and aluminum. The masses used for these materials go out to four decimal points and the atom’s mass must exactly match in order for a conversion to occur. The atomtype method can be used by setting routine to ‘atomtype’ and values to a list of element numbers. The atomtype is used to access the list at the index atomtype-1. The “data” input is a “Lmpsdata” class which is required for the mass algorithm to work. But even if your not using that algorithm the input is still required.

Functions

molecules(data, init, final, processors, method=‘all’)

Builds a list of “Lmpsmolecule” classes using Python’s multiprocessing module. The “Lmpsmolecule” classes begin with the molecular number “init” and end with the molecular number “final”. The “Lmpsdata” which is passed to the “Lmpsmolecule” classes come from the input “data”. The “processors” tells the function how many processes to devote to building the list. The multiprocessing used here is an embarrassingly parallel algorithm. There are two “methods”; the default method incorporates all the molecular data. The other method incorporates only the atom data and can be accessed by setting method to ‘atom’.

Future Versions

Updates to the software will add bonding of metallic and polymer nano particles to the polymer matrix. This expansion of the software capabilities may change which language the code is written in, but the current functionality and method calls should change little.