**LUNAR User Manual**

Josh Kemppainen ([jdkemppa@mtu.edu](mailto:jdkemppa@mtu.edu))

4/25/2024

Version 1.0

Contents

[Introduction 4](#_Toc162450710)

[Setup and Installation 5](#_Toc162450711)

[General LUNAR usage options 5](#_Toc162450712)

[Workflows and Use Cases 6](#_Toc162450713)

[Bond order force fields 6](#_Toc162450714)

[Fix-bond force fields 7](#_Toc162450715)

[Fix bond/react “REACTER” 9](#_Toc162450716)

[IFF/IFF-R π-electrons and graphitic systems 9](#_Toc162450717)

[IFF-R Morse bond update 10](#_Toc162450718)

[Molecular Dynamics-based Process Modeling 11](#_Toc162450719)

[Packmol interfacing 11](#_Toc162450720)

[VMD interfacing with CNT and graphite builders 14](#_Toc162450721)

[Code: atom\_typing.py 15](#_Toc162450722)

[Purpose 15](#_Toc162450723)

[Code Variables and Execution 16](#_Toc162450724)

[Outputs 25](#_Toc162450725)

[Code: all2lmp.py 25](#_Toc162450726)

[Purpose 25](#_Toc162450727)

[Code Variables and Execution 26](#_Toc162450728)

[Outputs 44](#_Toc162450729)

[Reproducibility, publication, and citing LUNAR 44](#_Toc162450730)

[Code: bond\_react\_merge.py 45](#_Toc162450731)

[Purpose 45](#_Toc162450732)

[Code Variables and Execution 46](#_Toc162450733)

[Outputs 65](#_Toc162450734)

[Code: auto\_morse\_bond\_update.py 66](#_Toc162450735)

[Purpose 66](#_Toc162450736)

[Code Variables and Execution 67](#_Toc162450737)

[Outputs 77](#_Toc162450738)

[Code: cell\_builder.py 77](#_Toc162450739)

[Purpose 77](#_Toc162450740)

[Code Variables and Execution 78](#_Toc162450741)

[Outputs 88](#_Toc162450742)

[Code: bond\_react\_merge\_prep.py 89](#_Toc162450743)

[Purpose 89](#_Toc162450744)

[Code Variables and Execution 89](#_Toc162450745)

[Outputs 94](#_Toc162450746)

[Code: add\_pi\_electrons.py 94](#_Toc162450747)

[Purpose 94](#_Toc162450748)

[Code Variables and Execution 95](#_Toc162450749)

[Outputs 99](#_Toc162450750)

[Code: atom\_removal.py 100](#_Toc162450751)

[Purpose 100](#_Toc162450752)

[Code Variables and Execution 100](#_Toc162450753)

[Outputs 105](#_Toc162450754)

[Code: lmp2SYBYLmol2.py 105](#_Toc162450755)

[Purpose 105](#_Toc162450756)

[Code Variables and Execution 105](#_Toc162450757)

[Outputs 109](#_Toc162450758)

[Code: cluster\_analysis.py 109](#_Toc162450759)

[Purpose 109](#_Toc162450760)

[Code Variables and Execution 110](#_Toc162450761)

[Outputs 113](#_Toc162450762)

[Code: auto\_cluster\_analysis.py 113](#_Toc162450763)

[Purpose 113](#_Toc162450764)

[Code Variables and Execution 114](#_Toc162450765)

[Outputs 119](#_Toc162450766)

[Code: free\_volume.py 119](#_Toc162450767)

[Purpose 119](#_Toc162450768)

[Code Variables and Execution 119](#_Toc162450769)

[Outputs 127](#_Toc162450770)

[Code: log\_analysis.py 127](#_Toc162450771)

[Purpose 127](#_Toc162450772)

[Code Variables and Execution 127](#_Toc162450773)

[Outputs 134](#_Toc162450774)

[References 134](#_Toc162450775)

# Introduction

LUNAR (LAMMPS Utility for Network Analysis and Reactivity) is a set of codes developed by Josh Kemppainen during his Ph.D. at Michigan Technological University. The purpose of LUNAR is to serve as a general molecule builder and force field application tool to prepare inputs to LAMMPS simulation software and provide some post-processing and property predictions using the LAMMPS datafile [1]. LAMMPS is an open-source flexible and well-developed molecular dynamics (MD) simulation software package with a large variety of possible applications. In general, LAMMPS requires two inputs: an “input script”, which defines the MD simulation settings, and a “datafile” that defines the initial atomic positions, topological specification (bonds, angles, dihedrals, impropers), simulation cell dimensions, and force field parameters for the molecules that wish to be simulated. LUNAR aims to help automate building the LAMMPS “datafile” input, provide the ability of medial and post-processing of the LAMMPS datafile and assume general knowledge of the LAMMPS “input script” and MD theory. Some codes within LUNAR will also auto-generate LAMMPS “input scripts” to help with using LAMMPS.

The general philosophy of LUNAR is to build the minimal amount of code needed to generate LAMMPS datafile and/or LAMMPS molecule files while using other existing software such as ChemDraw, Avogadro, BIOVIA Materials Studio, VMD, MolView.org, etc, … to draw the starting molecules. Some of the chemical and molecule drawing tools listed previously are open-source or have free versions, while others are subscription-based. Since LAMMPS is open-source, it would be ideal to have a workflow to generate inputs to LAMMPS that is also open-source and free of charge. LUNAR is the bridge between software like Chemdraw, Avogadro, BIOVIA Materials Studio, and VMD to LAMMPS and LUNAR is offered as an open-source project. There have been some previous tools like LUNAR, but they are either very limited in the wide variety of available force field options, difficult to get up and running, or difficult to use, or require a subscription to use [2-11]. It is worth looking into the tools listed [2-11], but from the writer of LUNAR’s view, LUNAR is likely the easiest open-source option to use for building LAMMPS datafiles that are fit for novice to advanced MD modelers. LUNAR also has limitations but tries to be as easy to use as possible while maintaining a certain level of flexibility.

LUNAR is coded in Python v3 and assumes v3.7 or higher. Most of LUNAR is object-oriented programming (OOP) and has been extensively optimized for quick run time. The programmer of LUNAR is not a trained software developer, but rather a coding hobbyist. This means that while the best coding practices may have been attempted, they may not be up to rigorous software development standards. The source code can be modified by users as they wish or added onto to further develop the scope of LUNAR. The writer of LATs requests that no LUNAR code should be used on the backend of commercial software to push their subscription-based software forward. The manual is organized as follows:

* Recommended workflows (broad overview of LUNAR codes and how they complement one another)
* CODE.py (name of code)
  + Purpose of code
  + Code Execution
  + Code Variables
  + Code Outputs

This manual is quite long and comprehensive and is not meant to be read in its entirety before using any of the codes. The reading of this manual should be treated like reading the LAMMPS manual, where you only read the sections that you currently need and slowly work your way through the manual as you progress in your project. Also note that many of the code variables and execution methods are identical, meaning that once you are familiar with one of the codes the rest of the codes are similar to use. Running the codes and looking at the outputs also serves as a “hands-on” learning experience. The codes, run methods, and outputs are meant to be as intuitive and comprehensive as possible, thus it should be easy to learn how to use LUNAR and all its possibilities.

## Setup and Installation

LUNAR is coded in Python v3 and assumes features that only v3.7 or higher offers, therefore Python v3.7 or higher must be installed to run the LUNAR codes. There are a variety of different ways to build a Python environment if it is not already installed on your machine. Installing Python with Anaconda (<https://www.anaconda.com/>) is recommended if this is your first time using Python. Alternatively installing Python directly from Python’s website is a good option (<https://www.python.org/>), ensuring to check the box to include Python.exe to the path and making sure to check the box to install pip to allow the ability to add other Python packages easily. You may also install Python using both methods because each install method gives some unique abilities to run the LUNAR codes. LUNAR uses the standard Python library with minimal usage of secondary modules of **matplotlib**, **numpy**, **numba**, **tqdm**, **rdkit,** and **natsort**. To utilize some analysis methods found in **log\_analysis.py** the **scipy** module and/or the **pwlf** module will also need to be installed. Most of the code in LUNAR will run on the Python standard library with **auto\_cluster\_analysis.py** requiring **natsort**, **free\_volume.py** requiring **tqdm**, **numpy,** and **numba** (if you want to compile the code down to machine code), **auto\_morse\_bond\_update.py** requiring **matplotlib**, and **atom\_typing.py** requiring **rdkit** (if the users desire the ability to use SMILES strings to generate molecular structures). It is advised to install **matplotlib**, **numpy**, **numba**, **tqdm**, **natsort,** and **rdkit** using the pip manager. Depending on your installation method you will have to use pip in different ways, below is a few methods that depend on the installation method of python:

* Installed via <https://www.python.org/>
  + python -m pip install PACKAGE
* Installed via <https://www.anaconda.com/>
  + pip install PACKAGE
* Installed via: sudo apt-get install python3.11 (Linux Ubuntu example)
  + pip install PACKAGE
* Please resort to your installation method documentation for further details. You may also create virtual environments to run LUNAR, but that is up to the user to figure out how to setup virtual environments depending on their python installation method. Note if installed via <https://www.anaconda.com/> conda, can be used to install the required third-party packages as well.
* Alternativity LUNAR comes with a python script that can be run to automatically install all required dependencies. Here is how to run the automatic dependencies install script depending on your python install method:
  + Installed via <https://www.python.org/>
    - python install\_required\_packages.py
  + Installed via <https://www.anaconda.com/>
    - python3 install\_required\_packages.py
  + Installed via: sudo apt-get install python3.11 (Linux Ubuntu example)
    - python3 install\_required\_packages.py

## General LUNAR usage options

In general, LUNAR offers three different ways to interact with the codes through an IDE or command line override, or GUI. LUNAR codes were initially developed with the idea of using an IDE to run them which means it is best to get comfortable with the idea of opening each code in an IDE to manipulate the variables within the code file itself. Once you are familiar with running LUNAR via an IDE and can manipulate the variables in the IDE, you can switch to using the command line overrides or the GUI. Both the command line overrides, and the GUI are initialized from the variables within each code file which is why it is best to be familiar with the IDE usage before moving into the command line override or GUI usage. Changing the variables in the code file will then allow the user to control what the default settings are and can be used to speed up usage of both the command line override and the GUI, by minimizing what settings need to be manually changed each time you run the codes.

Please note that throughout this manual the command line notation to run the codes is “python3 CODENAME.py”, where in some cases depending on your Python installation method you may have to run the codes using “python CODENAME.py” (removing the “3” from “python3”). In most Unix environments “python” or “python2” are used for all variants of Python V2 and “python3” is used for all variants of Python V3. In Windows however, if you installed Python through the binaries provided on the official Python website (<https://www.python.org/>) you will be running Python through Windows “cmd” using “python CODENAME.py”, this is due to how Python was installed. Finally, aliases may be assigned such that you can run Python with your call to the interpreter. However, if you have done this, I trust you know how to run your Python interpreter via your aliases.

If Python was installed from the Python website directly (<https://www.python.org/>) you may open the LUNAR distribution and “double click” on **LUNAR.py** (or any of the CODENAME.py files if use\_GUI = True in the code file)which will open the master LUNAR GUI (or CODENAME GUI), which will then allow you to open the rest of the LUNAR GUIs. This method is nice to run LUNAR because the **LUNAR.py** GUI (or CODENAME GUI) loads very quickly. It gives the “codes” the feel of a “full piece of software” or “app-like feel” while still maintaining the OS-agnostic nature of a simple Python script (i.e., if LUNAR was compiled there would have to be different compilations based on the different OS architecture, but keeping everything as a Python interpreted script the OS architecture issue is dealt with during the initial install of Python).

# Workflows and Use Cases

The molecular structure and force field (if applicable) are read into LAMMPS via the LAMMPS datafile. Most of the codes in LUNAR are to generate or manipulate the LAMMPS datafile. The information present in the LAMMPS datafile is dependent on the force field that will be used to model atomic interactions and some style-based information that is dependent on how the user wants certain sections of the datafile to be organized. There are two main categories of force fields in LAMMPS 1.) bond order force fields and 2.) fix-bond force fields, each of which require different information in the LAMMPS datafile. The workflows and use cases chapter are split up based on this broad categorization of force fields.

## Bond order force fields

For most (if not all) LAMMPS datafiles that are for bond order force fields like ReaxFF, Tersoff, AIREBA, and REBO, … the only information that is required is simulation cell dimensions, Masses, and Atoms. This is a relatively small number of requirements needed especially in comparison to the number of requirements that exist for fix-bond force fields, which makes the generation of these LAMMPS datafiles much simpler. An example workflow is shown in Figure 1 for generating a LAMMPS datafile for a bond order force field (note that not all settings in **all2lmp.py** are used for this datafile generation).

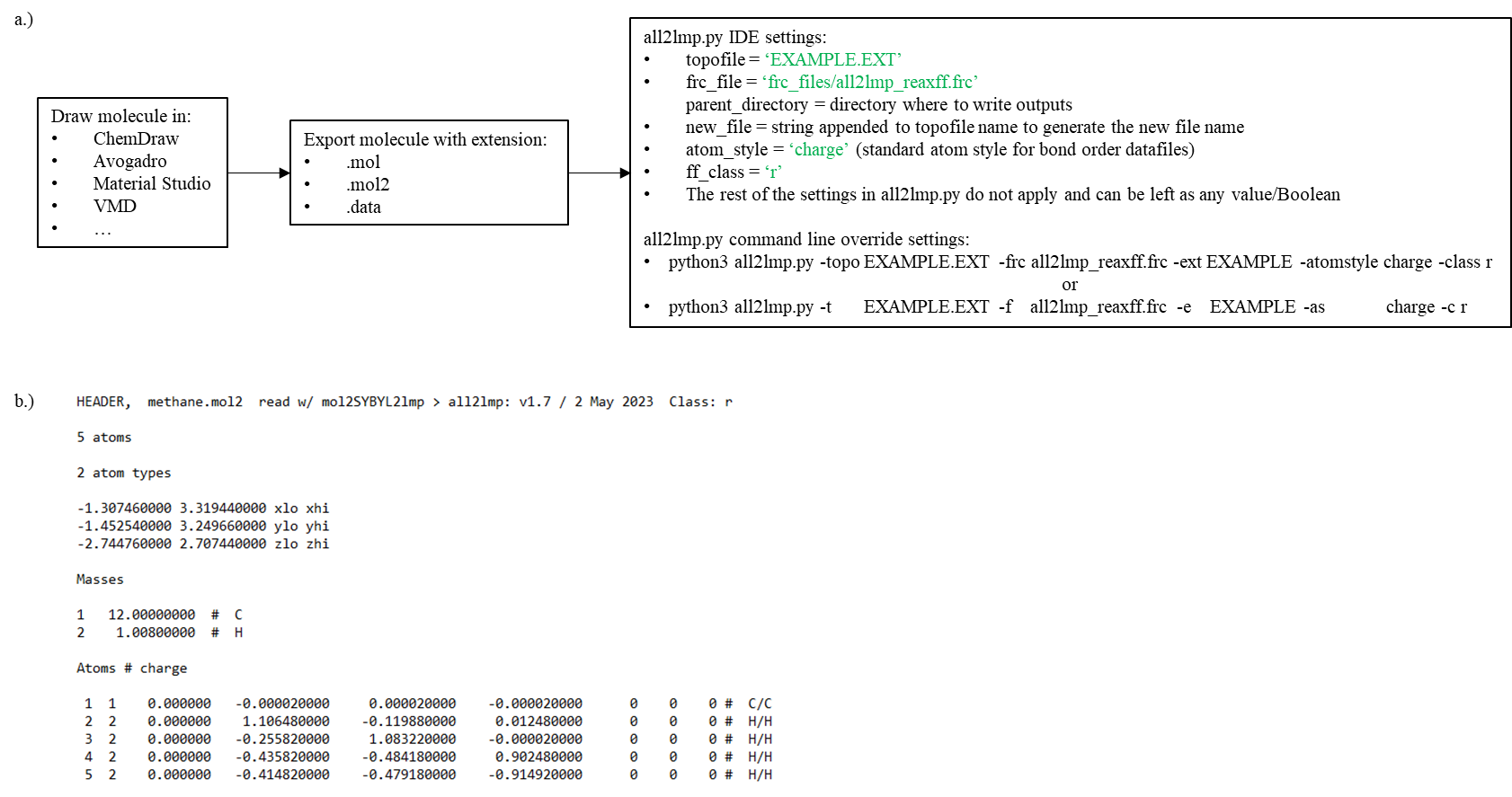


Figure 1: LAMMPS datafile generation workflow for bond order force fields: a.) workflow to generate LAMMPS datafile b.) example of LAMMPS datafile written by all2lmp.py.

The atomtypeID numbers are set based on the alphabetical ordering of element symbols. The charges will be set as zeros since most bond order force fields have a dynamic method to update charges as the bonding connectivity changes throughout the simulations. All atoms will also be geometrically centered about zero in the x, y, and z directions. The simulation cell dimensions will be set at the extent of the atoms ± 3 angstroms on all 6-faces of the simulation cell and the image flags are set to zero, due to the non-periodic setting of the simulation cell being larger than the molecule itself. The 3 angstroms large cell in each direction means any atom is minimally 6 angstroms away from the periodic atom, which is important, so the bond order force field doesn’t try modeling bonds that don’t make sense during the first simulation of the molecule(s).

## Fix-bond force fields

Generating a LAMMPS datafile for a fix-bond force field is more involved in comparison to a LAMMPS datafile for bond-order force fields. The general steps for using a fix-bond force field are:

1. Draw the molecule which you wish to simulate.
2. Set atom types based on the desired force field you want to use.
3. Find bonds, angles, dihedrals, and impropers that the molecule has and set the simulation cell domain.
4. Find the unique bond types, angle types, dihedral types, and improper types.
5. Then apply the force field parameters to the unique topology types and set per atom charges.
6. Re-order the bonds, angles, dihedrals, and improper atomID lists to be consistent with the permutation ordering of the found parameter types.

There are different codes in LUNAR to perform the steps listed above. The general sequence will be to use **atom\_typing.py** to set the atom types of the molecule and take the outputs of **atom\_typing.py** to use as inputs to **all2lmp.py**. Both **atom\_typing.py** and **all2lmp.py** have a variety of different settings that work together to apply the force field and charges to the molecular system. Specifically, how charges get set for different force fields and the **.nta** (**n**ew atom **t**ype **a**ssignment file) that **atom\_typing.py** generates. Both **atom\_typing.py** and **all2lmp.py** have different charging methods available via the **reset\_charge**s Boolean variable; **atom\_typing.py** **reset\_charge**s Boolean variable sets charge via the Gasteiger method [12] and **all2lmp.py** uses the bond-increments method and applies the bond-increments that are found in the read in force field file to **all2lmp.py**.Also, note that all the codes in LUNAR maintain the per atom charge such that you may set the charges by using the **.mol2** or **.data** or **.pdb** or **.car/.mdf** file format (since there is a charge column in these file formats) and then turn-off any charging methods found in **atom\_typing.py** or **all2lmp.py** to maintain the charge that is set via the file. The tool linked below seems to be a very powerful and useful tool for adding charges to a **.mol2** via a variety of different charging methods available:

* <https://github.com/sb-ncbr/eem_parameters>
* <https://acc2.ncbr.muni.cz/>

**all2lmp.py** supports many different force fields (some force fields have different dihedral or improper definitions) that have a parameter file in the Material Studio **.frc** file format. **atom\_typing.py** currently supports automatic atom typing for the following force fields:

* PCFF-IFF
* PCFF
* Compass
* CVFF-IFF
* CVFF
* Clay-FF
* DRIEDING
* OPLS-AA

Note that if **atom\_typing.py** does not yet support the force field you wish to work with, but you have a valid force field file in **.frc** format, you may manually generate a **.nta** file of atom types to read into **all2lmp.py** to apply the force field parameters and generate a LAMMPS datafile. On the other hand, if **atom\_typing.py** supports your force field, but you wish to use different atom types you may manually adjust the atom types in the **.nta** file before passing the files into **all2lmp.py**. Each force field requires different methods of assigning charges to the system, which requires adjusting the settings in **atom\_typing.py** and **all2lmp.py** accordingly for each force field (Table 1). Note that both the PCFF-IFF and CVFF-IFF charging method in Table 1 is listed as bond-increment, which is correct if you are using the native PCFF or CVFF atom types. However, if you are using the newer IFF atom types in either force field you must manually set the charge for each atom in the system as per the guidelines set out by the developers of IFF (<https://bionanostructures.com/interface-md/>). There are additional options that can be added to the **.nta** file that **all2lmp.py** reads that can help in setting the charge for the newer IFF atom types which will be discussed in the **all2lmp.py** chapter.

Table 1: Force Field charging method and respective settings for atom\_typing.py and all2lmp.py

|  |  |  |  |
| --- | --- | --- | --- |
| Force Field | Charge method | **atom\_typing.py** reset\_charges Boolean | **all2lmp.py** reset\_charges Boolean |
| PCFF-IFF | Bond-increment | False | True |
| PCFF | Bond-increment | False | True |
| Compass | Bond-increment | False | True |
| CVFF-IFF | Bond-increment | False | True |
| CVFF | Bond-increment | False | True |
| Clay-FF | Set manually | False | False |
| DRIEDING | Gasteiger | True | False |
| OPLS-AA | NOT SUPPORTED YET | False | False |

## Fix bond/react “REACTER”

The LAMMPS command *“fix bond/react”* also known as the REACTER package allows for a heuristic approach to modeling chemical reactions (both bond creation and bond breaking) in a classical MD simulation [13, 14]. However, the usage of the REACTER package requires multiple LAMMPS datafiles and LAMMPS molecule files that are coupled together with all the LAMMPS coeffIDs being the same amongst all files. Which makes the manual generation of all the files a complex, time-consuming, and error-ridden process. Some of the LUNAR codes can be used in a sequence to fully build all files required to run a simulation using the REACTER package. The sequence proposed by LUNAR of building all the files required for the REACTER package is as automated as the process has been able to be thus far amongst the users of the *“fix bond/react”* command and can be completed all within one consistent set of codes (other methods require using multiple different software’s and code each of which has their quirks and limitations that a modeler must learn and there is a lack of consistency which makes streamlining the process difficult). To generate a complete set of files for usage with the LAMMPS *“fix bond/react”* command the user can follow the steps below:

1. Draw your starting molecules and chemical reactions in whichever software you desire (ChemDraw, Avogadro, BIOVIA Materials Studio, VMD, etc …) and export a file in **.mol** or **.mol2** or .**pdb**, or **.mdf**/**.car** or **.data**.
2. Use **atom\_typing.py** to automatically assign atom types for a given force field.
3. Use **all2lmp.py** to generate a fully parameterized LAMMPS datafile of all molecules and reactions
4. Use **bond\_react\_merge.py** to find all the unique parameters and re-type all the starting molecules and chemical reactions. **bond\_react\_merge.py** can also be used to auto-generate a map file relating the pre-reaction template to the post-reaction template (it is up to the user to check that the auto-generated map file is correct since this is a complex task to code, however, the current algorithm has been extensively tested and has been shown to get the mapping completely correct in about 95% of the test cases). **bond\_react\_merge.py** may also be used to reduce the chemical reaction templates if the user wants to just draw the two reacting molecules and offer support for dealing with the charges of edge atoms in the modeled chemical reactions.
5. If modeling reactions for polymers: Use **cell\_builder.py** to easily mix the starting molecules in an easy-to-control stoichiometric ratio. **cell\_builder.py** allows for the complete randomization of molecules or for controlled levels of randomization to simulate the experimental mixing stage used in most thermoset polymerization processes.
6. Refer to each chapter on the codes referenced for further details on the code usage. NOTE that **all2lmp.py** can create LAMMPS datafiles with the new LAMMPS “type labels” included, where **bond\_react\_merge.py**, **cell\_builder.py**, and **auto\_morse\_bond\_update.py** can maintain the “type labels” or use the consistent commenting scheme present within the LUNAR codes to derive the “type labels” from the comments within the files.

## IFF/IFF-R π-electrons and graphitic systems

The IFF/IFF-R force field allows for the modeling of electrostatic interactions in graphitic-type systems such as CNTs and graphene/graphite. The 6-member ring present in such carbon-based structures has proven difficult to provide any partial atomic charge since electronegativity differences (and thus modeling charges) do not exist in a symmetric, carbon-pure system. To provide the ability to model charging on the carbon-based structures, IFF utilizes the idea of virtual π-electrons which are atoms that explicitly represent the pi-orbitals of aromatic compounds. A charge can be placed on the carbon atoms and an equal but opposite charge placed on the π-electrons to maintain charge neutrality, allowing for electrostatic effects to be modeled within atomically pure, aromatic systems such as CNTs and graphene/graphite MD models. The virtual π-electrons also allow for Lennard Jones parameters to be fit which, when combined with the charging, allows for the MD model to predict the π-π stacking present in graphite.

However, one difficulty of implementing an IFF/IFF-R simulation for a CNT or graphite is building the virtual π-electron model. The model building process requires the placement of the virtual π-electron atoms above and below the carbon plane, creating the bonds and angles required to hold the atoms in place as well as inserting the correct charges and parameters of each new π-electron atom. The process is even more complicated for CNTs which have a circular geometry that requires a certain method of locating the π-electron atom's initial positions. A code within LUNAR called **add\_pi\_electrons.py** has the option of adding in the IFF/IFF-R virtual π-electrons and assigning the corresponding parameters and charges properly.

The **add\_pi\_electrons.py** code has other features that allow for building CNTs and graphite models with PCFF native atom types and then converting only the graphitic carbon atom types (i.e., cg1) to the IFF/IFF-R carbon atom types for CNTs and graphite. The reason why this is of importance is because the IFF/IFF-R force field currently does not have many parameters to model bonds between the new graphite atom types and other PCFF native atom types. In the future, the IFF developers may include parameters or equivalences for polymer bindings to graphite and other aromatic carbons, but for the time being it is very difficult to model any sort of polymer bonding to a CNT or a graphene sheet using PCFF-IFF. Furthermore, there has recently been an uptake in modeling amorphous carbon and glassy carbon with ReaxFF, where there are a variety of different carbon species present in a model beyond ones that relate to graphitic carbon atom types, which requires usage of the native PCFF atom types. The **atom\_typing.py** and **all2lmp.py** codes can be used to convert any ReaxFF model developed in LAMMPS to PCFF-IFF using only the PCFF atom types and then using **add\_pi\_electrons.py** to update the PCFF cp atom type to the IFF cg1 atom type, thus allowing for complex graphitic type systems to be modeled within PCFF-IFF, which would be virtually impossible before the LUNAR codes.

## IFF-R Morse bond update

Any model built with PCFF-IFF, CVFF-IFF, or any class1 or class2 force field that does not already have Morse bonds explicitly defined, can have Morse bonds easily added using **auto\_morse\_bond\_update.py**. Which is a code that is agnostic to the nuances in different force fields and only assumes the bonding parameters are either in the LAMMPS class2 or harmonic format. For usage with PCFF-IFF or CVFF-IFF, the new force field gets the “-R” appended to the force field name (i.e., PCFF-IFF -> PCFF-IFF-R and CVFF-IFF - > CVFF-IFF-R) to signify that the force field is now “Reactive” since bonds using the Morse bond potential are capable of bond dissociation (however, not bond breaking unless using the *“fix bond/break”* command found in LAMMPS). For any other force field, it is recommended to use the “-M” append to the force field name (i.e., DREIDING -> DREINDING-M, OPLS-AA -> OPLS-AA-M) since this is historically how the Morse bond variants have been denoted. This does not make such a grand statement as the “-R” appendage in its capabilities but rather is a more reasonable statement about describing the change from the harmonic forms of the force field to the Morse bond forms of the force field, since it has proven difficult to incorporate bond breaking using classical fix-bond force fields. To build the initial datafile required as an input to **auto\_morse\_bond\_update.py**, **atom\_typing.py,** and **all2lmp.py** can be used. Note, it is recommended to first build your system using the harmonic forms of any force field, obtain a fully equilibrated model in the harmonic form, and then convert to the Morse bond form, since the harmonic form can generate large restoring forces to ensure a properly equilibrated model.

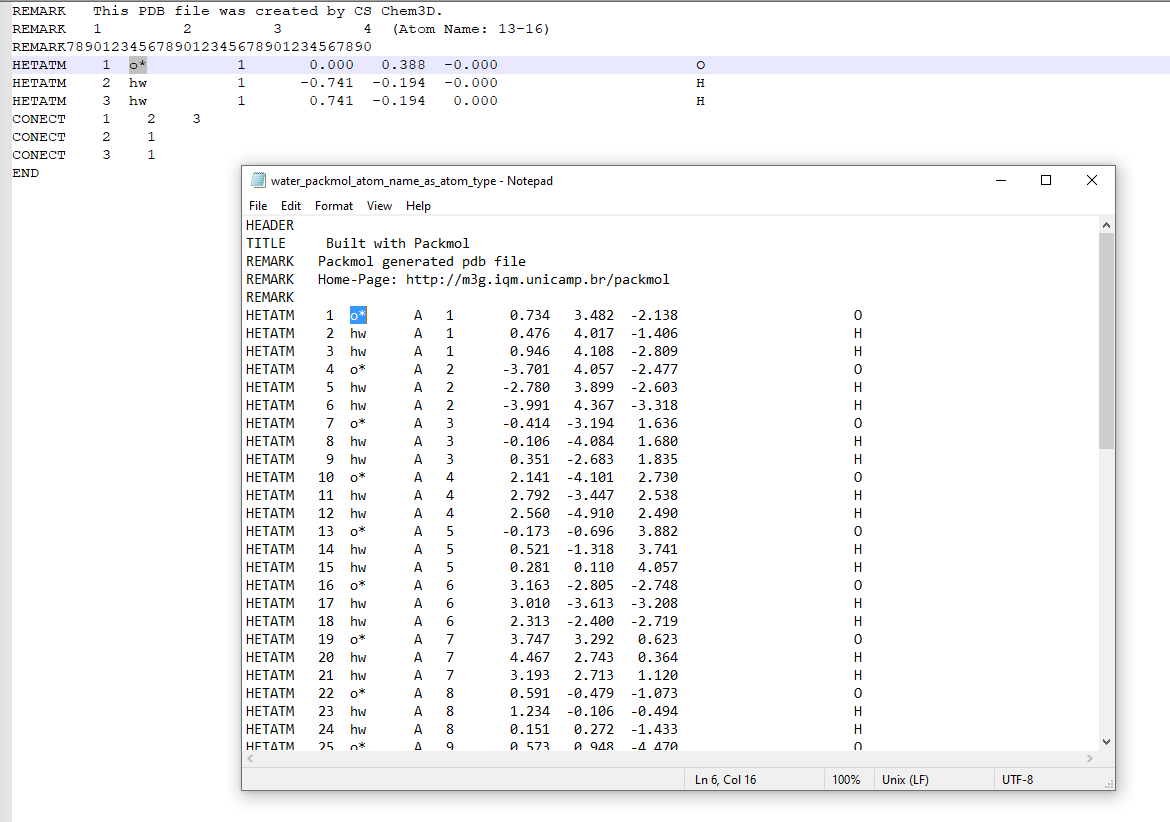
## Molecular Dynamics-based Process Modeling

The research group that Josh Kemppainen belongs to for his Ph.D. primary focus was (and is) to use MD for its ability to produce structure-property relationships to be used in Process Modeling of polymeric and glassy carbon-based materials. The LUNAR codes have been developed with the end goal of making Process Modeling as quick, easy, and effective as possible using MD. This means LUNAR offers the ability to generate LAMMPS inputs quickly and easily, tools to help model chemical reactions, and a level of support for post-processing related to Process Modeling. Three codes are of interest for post-processing, where two deal with quickly analyzing the bonding connectivity evolution from a *“fix bond/react”* simulation to find the gel point as well as find basic molecular weights distribution of the polymer chains in the LAMMPS datafiles which are **cluster\_analysis.py** and **auto\_cluster\_analysis.py**. Additionally, LUNAR offers a methodology to estimate the free volume in polymers and glassy carbon-based materials via the **free\_volume.py** code.

## Packmol interfacing

Currently, the entire LUNAR codes suite has four main methods to interface with packmol. Packmol is a code written in Fortran to “pack” a system of molecules and is meant to provide the initial positions to an MD simulation [15, 16]. LUNAR has a code like packmol called **cell\_builder.py**, however, packmol is more advanced which is why interfacing between LUNAR and packmol has been accounted for in case users of LUNAR want to use packmol. The four main methods are described below all of which use the **.pdb** file format:

* Method 1 (perhaps the most straightforward method)
  + Draw molecules in ChemDraw, Materials Studio, etc.. and export in **.pdb** format.
  + Generate a packmol script and “pack” a large system, using the filetype **pdb**.
  + Load the packmol **.pdb** file in **atom\_typing.py** to set the atom types for a specific FF.
  + Load the outputs from **atom\_typing.py** in **all2lmp.py** with the correct **.frc** FF file to generate a LAMMPS datafile.
  + Considerations for this method:
    - If **atom\_typing.py** does not have an atom typing module set up for the FF you wish to work with, this method is not possible since it becomes nearly impossible to manually atom type a system over a few hundred atoms.
    - If the users wish to adjust any atom type, it can also be nearly impossible due to the system size you will likely be building in packmol.
    - Attempting to build *“fix bond/react”* templates from a large molecular system can be challenging because the atom types MUST be consistent between the reacting monomers (supplied via the packmol output) and the pre-reaction template (built outside of packmol).
* Method 2 (provides some flexibility, but requires some manual editing of the **.pdb** file)
  + Draw molecules in ChemDraw, Materials Studio, etc.. and export in **.pdb** format.
  + Manually edit the “atom name” column of the **.pdb** file (13-16) by setting the FF-specific atom types in the “atom name” column. Packmol will read and maintain the “atom name” column when duplicating your system and the **all2lmp.py** can read in the “atom name” column as the system atom types. An image is provided below of a manually edited **.pdb** file from ChemDraw 3D and the output **.pdb** from a packmol simulation.



* + Generate a packmol script and “pack” a large system, using the filetype **pdb** and using the manually edit **.pdb** file.
  + Then take the outputted **.pdb** file from packmol and load it into **all2lmp.py** setting the **nta\_file** variable as ‘types\_from\_pdb.nta’ which tells **all2lmp.py** to not read any **.nta** file, and look for the atom types inside the **.pdb** file in the “atom name” column.
  + Considerations for this method:
    - In the **.pdb** file columns 13-16 are the “atom name” column, which means any atom type you wish to assign must be less than four characters long and some FFs such as DREIDING require more than four characters to define the atom type. Making this method limited to certain FFs.
* Method 3 (allows for automatic atom typing of small molecules with the possibility of manually adjusting some atom types if needed)
  + Draw molecules in ChemDraw, Materials Studio, etc.. and export in **.pdb** format.
  + Load the **.pdb** file into **atom\_typing.py**, set the FF atom types you would like to assign to the system, and set the **pdb\_file** variable to ‘types’, which will write an output **.pdb** file with the “atom name” columns filled with the FF atom types.
  + Generate a packmol script and “pack” a large system, using the filetype **pdb** and using the automatically assigned atom-typed **.pdb** file from **atom\_typing.py** (which may also be manually edited if needed).
  + Then take the outputted **.pdb** file from packmol and load it into **all2lmp.py** setting the **nta\_file** variable as ‘types\_from\_pdb.nta’ which tells **all2lmp.py** to not read any **.nta** file, and look for the atom types inside the **.pdb** file in the “atom name” column.
  + Considerations for this method:
    - In the **.pdb** file columns 13-16 are the “atom name” column, which means any atom type you wish to assign must be less than four characters long and some FFs such as DREIDING require more than four characters to define the atom type. Making this method limited to certain FFs.
* Method 4 (allows for automatic atom typing of small molecules with the possibility of manually adjusting some atom types if needed and allows for FFs that use more than four characters to define an atom type)
  + Draw molecules in ChemDraw, Materials Studio, etc.. and export in **.pdb** format.
  + Load the **.pdb** file into **atom\_typing.py**, set the FF atom types you would like to assign to the system and set the **pdb\_file** variable to ‘typeIDs’, which will write an output **.pdb** file with the “atom name” columns filled with the atom type integers (atomTypeIDs) and a **.nta** file “style type” option to map the integer values in the **.pdb** “atom name” column to atom types found in the **.nta** file.



* + Generate a packmol script and “pack” a large system, using the filetype **pdb** and the automatically assigned atom type **.pdb** file from **atom\_typing.py** (which may also be manually edited if needed).
  + Then take the outputted **.pdb** file from packmol and load it into **all2lmp.py** setting the **nta\_file** variable as the .nta file that **atom\_typing.py** created where **all2lmp.py** will know to look for the integers inside the **.pdb** file in the “atom name” column to map onto the atom types set in the **.nta** file.
  + Considerations for this method:
    - This method allows for easy changing of atom types even after a packmol simulation.
    - This method allows atom-type names to be longer than four characters long, making it compatible with any FF.
    - If loading multiple files into packmol, this method may be tedious because the atomTypeID listed in the **.nta** file will have to map onto the atomTypeID set in the **.pdb** file, which

A few general considerations for the above packmol/LUNAR/LAMMPS interfacing is that it is done through the usage of the **.pdb** file, which has the following limitations:

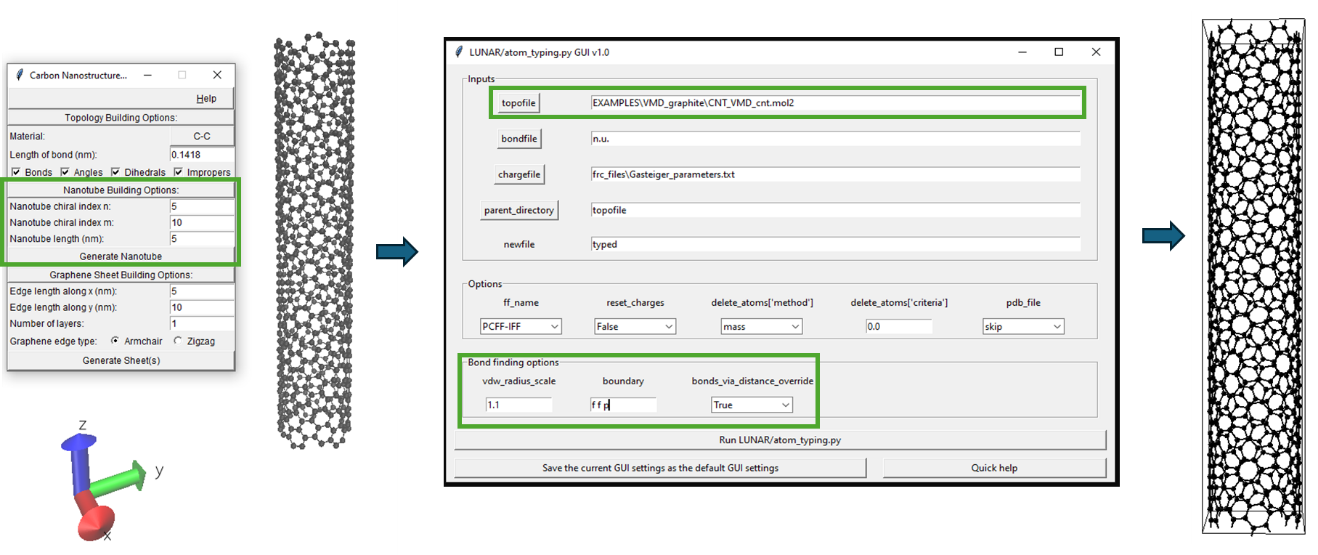
* No atomID can be larger than 99,999 in a **.pdb** file.
* No position can be greater than 999 Å or less than -999 Å **.pdb** file.
* The charge column of **.pdb** files is only two characters long so charging methods again fall back to LUNAR’s responsibility since a floating point number with only two characters is not acceptable for modeling charge in an MD simulation.

If you find your models run into the **.pdb** file limitations you will have to resort to using **cell\_builder.py**, since that code has essentially no limitations other than it currently only reads LAMMPS datafiles and the CoeffIDs must be merged/unified via **bond\_react\_merge.py** between different LAMMPS datafiles. However, if you are using LUNAR, it means you are looking to build inputs to LAMMPS so the LAMMPS datafile limitation for **cell\_builder.py** is not a limitation and the CoeffIDs can be easily unified/merged by preprocessing the files in **bond\_react\_merge.py**. It is also desired that **cell\_builder.py** does not manipulate the CoeffIDs to keep **cell\_builder.py** compatible with building inputs for the LAMMPS *“fix bond/react”* command, where you can run **cell\_builder.py** just on LAMMPS datafiles that already have consistent CoeffIDs between the LAMMPS datafiles and LAMMPS molecule files.

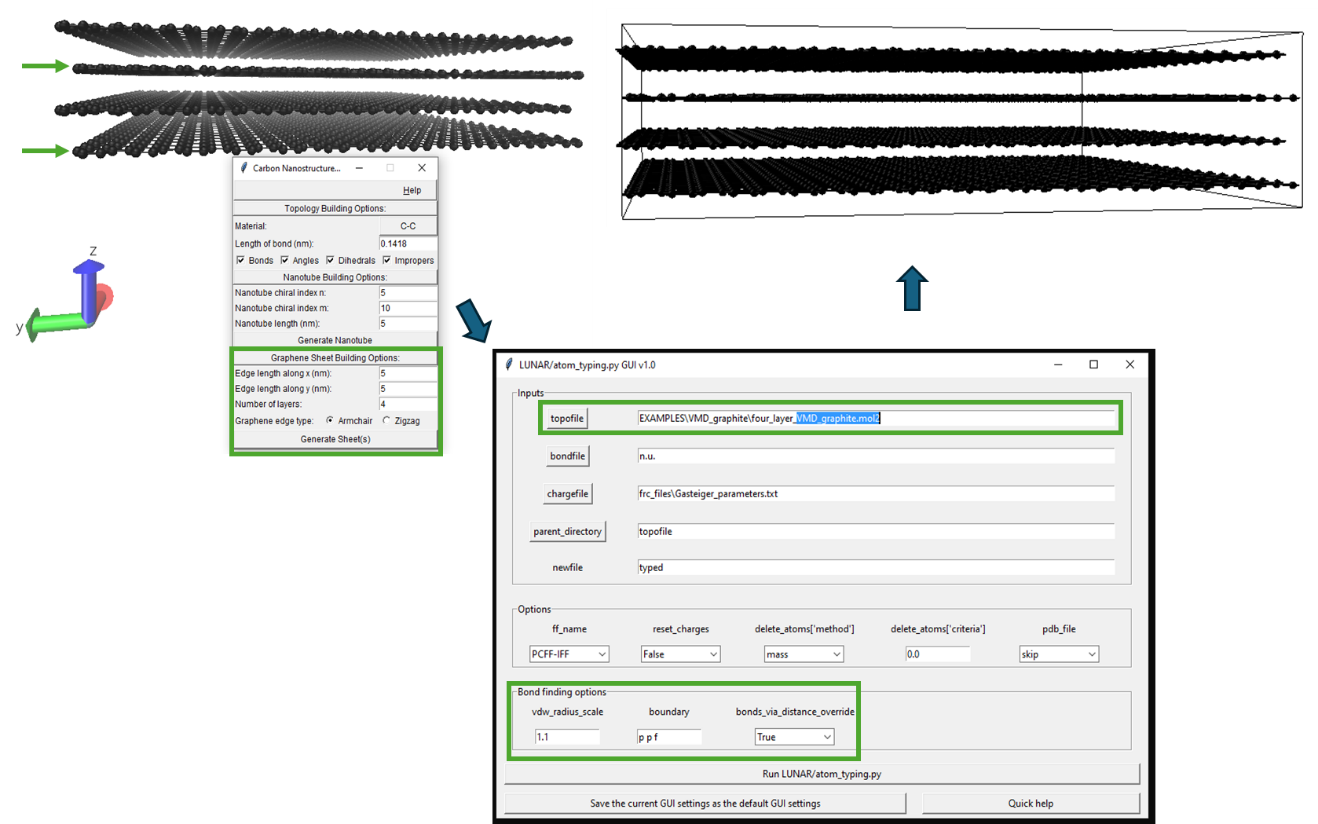
## VMD interfacing with CNT and graphite builders

There are a variety of different method to create carbon nanotubes (CNTs), graphene/graphite sheets, and boron nitride sheets, such as using the LAMMPS *“lattice”* command, using Avogadro, or using VMD Extensions > modeling > Nano tube builder. VMD is perhaps the quickest and easiest method to generate these graphitic type of structures, however VMD does not natively understand periodic boundaries and in most cases, these type of systems should be bonded periodically to use in MD simulations. A method for using VMD graphitic structures with LUNAR, to generate periodically bonded models will be discussed below.

We will first look at how to make a CNT generated with VMD be periodically bonded. You will have to save the coordinates in VMD by clicking File > Save coordinates > Selected atoms: (all) > File type: (mol2), where you will have to explicitly type the *“\*.mol2”* extension in the filename as VMD does not append this automatically. Next you will have to load the file into **atom\_typing.py** as shown below and set the **boundary** as ‘f f p’ since the only direction that is periodic is the z-direction. Finally, you will have to set **bonds\_via\_distant\_overide** as True, to inform the code to reset all bonds, by considering the currently defined boundary. This will reset the bonds, which then can be checked by looking at the outputs in OVITO.



Next we will look at how to make graphite generated with VMD be periodically bonded. This is a little bit more involved as VMD generates graphite in A-B stacking, which shifts alternating layers to establish the proper stacking. This piece of information about alternating stacking must be communicated to the code that reads the .mol2 file and it was decided that a file naming convention will be utilized as this type of modeling building will be a rarer cases. The naming convention is as follows “%%%VMD\_graphite.mol2”, where %%% can be any text and the “VMD\_graphite.mol2” ending will be looked for by the code to switch on the options for setting a proper simulation cell around the A-B stacked graphite and then re-wrap any atoms that are outside of the simulation cell. You will have to save the coordinates in VMD by clicking File > Save coordinates > Selected atoms: (all) > File type: (mol2), where you will have to explicitly type the *“\*VMD\_graphite.mol2”*, where *“\*”* can be anything, but the ending must be *“\*VMD\_graphite.mol2”* as this will inform the code that reads the .mol2 file that the file is a graphite system generated with VMD. Next you will have to load the file into **atom\_typing.py** as shown below and set the **boundary** as ‘p p f’ since the two directions that are periodic are the x- and y-direction. Finally, you will have to set **bonds\_via\_distant\_overide** as True, to inform the code to reset all bonds, by considering the currently defined boundary. This will reset the bonds, which then can be checked by looking at the outputs in OVITO.



# Code: atom\_typing.py

## Purpose

The purpose of **atom\_typing.py** is to automatically assign different force field (FF) atom types to a specific molecular system. Most MD fix-bond FFs rely on something called an atom type which is usually topology-specific or based on a functional group. The atom type gets used to apply the force field parameters to that atom and its neighboring atoms. A graph-theory approach is used to quickly find the local topology of each atom, which is then used to assign an atom type to that specific atom. All force fields are limited to certain molecules/topologies so the code will attempt to set the most accurate atom type for the topology first and if it is successful that will be the atom type that gets assigned; if it is not successful a more general atom type (if the force field has one) will be assigned, if there is not a more general atom type the code will insert an assumed atom type that the user will be made aware of (via of printouts and comments in the written files). Lastly, if there is not an assumed atom type the code will warn the user that the automatic atom typing has failed for that atomID number.

## Code Variables and Execution

**atom\_typing.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **atom\_typing.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 2.

A screenshot of a computer program

Description automatically generated

Figure 2: IDE view of some Python variables for atom\_typing.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 atom\_typing.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 3.

A screenshot of a computer screen

Description automatically generated

Figure 3: Command line override manual printed by running “python3 atom\_typing.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **atom\_typing.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **atom\_typing.py** file. Also note that not all variables found in **atom\_typing.py** can be overwritten at the command line, which means that manipulating them requires opening the **atom\_typing.py** file in a text editor or IDE to set the default values. Examples of command line override:

python3 atom\_typing.py -topo test1.mol -dir testing\_directory -ff PCFF-IFF

python3 atom\_typing.py -topo test2.mol -dir testing\_directory -ff CVFF -reset-charges T

python3 atom\_typing.py -topo CCC1=CC(=C(C(=C1N)C)N)CC.smiles -ff DREIDING

python3 atom\_typing.py -gui

python3 atom\_typing.py -man

python3 LUNAR.py

python3 LUNAR.py 125

It is worth noting that there is a **use\_GUI** Boolean flag in **atom\_typing.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 atom\_typing.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supple command line arguments other than the “-gui” option such as “python3 atom\_typing.py -topo test1.mol -dir testing\_dir”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 atom\_typing.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **atom\_typing.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **atom\_typing.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **atom\_typing.py** GUI are shown in Figure 4. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **atom\_typing.py** file, except for the **topofile** and the **bondfile** variables which will remain as the default this code ships with or whatever the users has manually typed in **atom\_typing.py**.

A screenshot of a computer

Description automatically generated

Figure 4: LUNAR/atom\_typing.py GUI

The **LUNAR.py** and the **atom\_typing.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **atom\_typing.py** can be set to different values, where **GUI\_zoom** in **atom\_typing.py** only affects the GUI that is launched from **atom\_typing.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **atom\_typing.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the “chemistry” of the system and can be generated with software like ChemDraw, Avogadro, BIOVIA Materials Studio, LAMMPS, MolView.org, and VMD. Open Babel can be used to convert most other chemical file formats to ones that are currently compatible with **atom\_typing.py**.
* Compatible formats (The file extension is important, so the code knows how to read it):
  + **.mol** file (ChemDraw, Avogadro, and BIOVIA Materials Studio, etc, …)
  + **.mol2** file (ChemDraw, Avogadro, VMD, and BIOVIA Materials Studio, etc, …)
  + **.pdb** file (ChemDraw, etc,) OFFERS REDIMENTARY SUPPORT FOR .pdb files
  + **.data** file(LAMMPS datafile where the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section)
  + **.smiles** (provide a smiles string with the .smiles extension and the .smiles extension will be removed, and the smiles string will be used to generate the molecule. This option requires **rdkit** to be installed and has a maximum number of atoms limit of 999).
  + NOTE that a **.data** and **.mol2** file may have charges assigned already and the implications of **reset\_charges** variable.

Variable: **bondfile** Override tags: **-bond** or **-b** Usage: **OPTIONAL**

* Is an optional file name to be read into the code. The file format is a LAMMPS-specific file format for ReaxFF simulations that can be generated with the LAMMPS command *“fix reaxff/bonds”* command (<https://docs.lammps.org/fix_reaxff_bonds.html>). The code that reads this file will read all-time series of bond order data and average the bond orders together to come up with a time-averaged bond order. The code will then compare the time-averaged bond order to a defined bond order cut-off (set by the **bondorder** variable) to determine if the bond is strong enough to be considered a bond. Lastly, all atoms will be iterated through and checked if there are too many bonded atoms based on the per-element number of bonded cut-offs (set by the **maxbonded** variable). If there are too many bonded atoms to the specific atom element, the lowest bond order bond(s) will be rejected. If your system already has bonds in it, then this option will not be used and can be set to **bondfile** = ‘n.u.’, where ‘n.u.’ means “not used”. Also, it is up to the user to decide how many times to write to the bond order file within LAMMPS and over how long of a simulation period to write to that file. It is also important to think about the temperatures in the simulation when writing the bond orders since that influences the magnitude of vibrations and the reaction chemistry. A default suggestion would be to write to the bond order file every 0.01 ps for a total of 0.1 ps as shown in the pseudo LAMMPS script below:

timestep 0.1

fix 1 all npt temp 300 300 $(100\*dt) aniso 1 1 $(1000\*dt)

fix 2 all reaxff/bonds 100 bondorder\_filename.reaxff

run 1000

* Please note that very large **bondorder** files that have been written over large simulation times may cause very slow code performance since averaging all the bond order data together is a demanding task. Additionally, it would be very wrong to try to average too many bond orders together over long simulation times since that would give a “bad” understanding of the current bonding connectivity.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **atom\_typing.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the **topofile** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **atom\_typing.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_typed’** and **topofile == ‘detda.data’):**
        + basename = ‘detda\_typed’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘typed-:’** and **topofile == ‘detda.data’):**
        + basename = ‘typed-detda’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as ‘ANYTEXT’. For example:
      * **(newfile = ‘detda\_renamed’** and **topofile = ‘detda.data’):**
        + basename = ‘detda\_renamed’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.
    - **Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.

Variable: **pdb\_file** Override tags: **-pdb** or **-p** Usage: **REQUIRED**

* Is a Python string variable in which to determine if any additional files will be written by **atom\_typing.py**. These options exist to help with interfacing with packmol, but may also be found to be useful outside of interfacing with packmol. Currently supported **pdb\_file** options and their meanings (Case matters):
  + ‘skip’ will not write additional \*\_packmol.pdb and \*\_packmol.nta files.
  + ‘types’ will write additional \*\_packmol.pdb file but not \*\_packmol.nta, with the atom types set in the atom name.
  + ‘typeIDs’ will write additional \*\_packmol.pdb and \*\_packmol.nta files, with atomTypeIDs set in the atom name column of the .pdb file and atom types set in the .nta file using the "style type" method.
* This option was added to provide more methods for interfacing with packmol. Please see the EXAMPLES/packmol\_pdb\_methods/ directory for different methods for interfacing with packmol.

Variable: **ff\_name** Override tags: **-ff** or **-f** Usage: **REQUIRED**

* Is a Python string variable to call which force field atom types to assign to the molecular system read in via the **topofile** variable. Supported **ff\_name** strings (case sensitive) and the associated force field:
  + ‘PCFF-IFF’ (Interface force field based on PCFF)
  + ‘PCFF’ (Polymer consistent force field)
  + ‘compass’ (Compass force field)
  + ‘CVFF-IFF’ (Interface force field based on CVFF)
  + ‘CVFF’ (consistent valence force field)
  + ‘Clay-FF’ (Clay force field)
  + ‘DREIDING’ (DREIDING force field)
  + ‘OPLS-AA’ (OPLS all-atom force field which currently has minimal parameters)
  + ‘general:0’ (will set atom type as elementRINGnb (i.e., C in 6 member ring TYPE = C63))
  + ‘general:1’ (will set atom type as elementRINGnb-(1st-neighs) and (1st-neighs) = (count:elementRINGnb,...))
  + ‘general:2’ (will set atom type as elementRINGnb-(1st)-(2nd) and (ith-neighs) = (count:elementRINGnb,...))
  + ‘general:3’ (will set atom type as elementRINGnb-(1st)-(2nd)-(3rd) and (ith-neighs) = (count:elementRINGnb,...))
  + ‘general:4’ (will set atom type as elementRINGnb-(1st)-(2nd)-(3rd)-(4th) and (ith-neighs) = (count:elementRINGnb,...))
    - The general:N atom type options allow users to generalize local topologies into a string of characters that is endowed with topological meaning. This will allow users to apply atom types in a pseudo-automatic manner if they are working with a force field that an atom typing module has not yet been built. The intended usage of the general:N options is for large system (>100 atoms) atom typing, where manually setting an atom type for each atom is difficult due to the sheer number of atoms. When using the general:N option all unique strings of characters to represent an atom type will be set in an **“equivs”** section at the top of the generate **.nta** file where the user will simply have to set the map from the unique string of characters to a force field specific atom type they desired to work with.

Variable: **chargefile** Override tags: **-charge-file** or **-qf** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the Gasteiger charge parameters that will be used with the Gasteiger charge method if **reset\_charges** is True.

Variable: **reset\_charges** Override tags: **-reset-charges** or **-rq** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to reset charges via the Gasteiger charge method [12]. DREIDING FF is known to use this method but may be used for other FFs as well. Setting **reset\_charges** as True will reset the charges with charges found via the Gasteiger method and setting as False will leave charges as the default read in charges. The charge convergence algorithm is geometry independent, but the Gasteiger initial parameters require knowledge of the hybridization state of the atoms in the system. The hybridization characterization of each atom is set via a hybrid method using topology (bond connectivity only) and minimization of VSPER angles (when topology could not be used). This means for the most accurate results for this charging method the atomic positions and the geometries they create via bond connectivity should be consistent with the VSPER angle theory (i.e., DO NOT have very stretched bonds or atomic positions that create un-realistic geometries).
* Please note the following about **topofile** extensions and charge information, since **atom\_typing.py** WILL maintain the read-in charge and only adjust charge if **reset\_charge** is True.
  + **.mol** files are read-in with all per-atom charges set as ZEROs.
  + **.mol2** files are initialized with all per atom charges as ZEROs and then updated during the reading of the **.mol2** file. So, if you would like to keep your system charge from a **.mol2** file, **reset\_charges** should be False.
  + **.data** files are read-in and the charge that is in the **.data** is left as is, unless **reset\_charges** is True, so if you want to maintain the charge in your system while reading in a **.data** file **reset\_charges** should be False.

Variable: **delete\_atoms** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary containing the keys: ‘method’ and ‘criteria’ which can be used to remove small molecules from any read in the system (such as volatiles). This option is mainly for usage when converting from ReaxFF to any other fix-bond force field because most of the time there are small molecules created by ReaxFF. The Python dictionary keys/values pairs control the following aspects of the **delete\_atoms**:
  + ‘method’:VALUE, where VALUE may be ‘mass’ or ‘size’
  + ‘mass’ means that small molecules will be identified via a mass cut-off value set by the ‘criteria’ key, where any molecule that has a mass less than the value of the ‘criteria’ key will be removed from the system.
  + ‘size’ means that small molecules will be identified via the number of atoms cut-off value set by the ‘criteria’ key, where any molecule that has fewer atoms than the value of the ‘criteria’ key will be removed from the system.
  + ‘criteria’:VALUE, where VALUE may be a float or an int value to set the cut-off value used for determining if the molecule is volatile or not.
* **atom\_typing.py** will always attempt to use **delete\_atoms** and thus the only way to guarantee to not have any molecules are deleted is to set ‘criteria’ as zero, which is the recommended setting for **topofile** that is either **.mol** or **.mol2** or **.pdb** or **.smiles** extension. After **delete\_atoms** are performed, all atomIDs are first sorted in ascending order and then are reset to be contiguous. This means the output files (discussed later) may have a change of atomIDs for some or all atoms in the system. Due to this possible change in atomIDs, it is recommended that only the outputs of this code be used as inputs to **all2lmp.py**.
* If ‘criteria’ is set as zero the atomIDs from the input files will remain consistent with the atomIDs of the output files (hence no renaming of the atomIDs will occur even if the atomIDs are not contiguous).
* If your input files do not have contiguous atomIDs and you wish to use **atom\_typing.py** to make the atomIDs contiguous. While still not deleting atoms, the ‘criteria’ value can be set to a very small non-zero number (such as 0.0001) for either the ‘method’ of removal of ‘mass’ or ‘size’ since every molecule will still be large enough to be kept, but it will tell **atom\_typing.py** to reset the atomIDs.

Variable: **print\_options** Override tags: **-man** or **-opt** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to print the command line override manual and exit the code. True will print out the command line override manual and exit, whereas False will allow the code to run.

Variable: **mass\_map** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as elemental symbols and the corresponding values being a list of masses associated with that element type. The purpose of the **mass\_map** dictionary is dependent on the file extension read in via the **topofile** variable. The following information in **mass\_map** is used in the following ways based on the **topofile** extension:
  + **.smiles**, **.pdb**, **.mol**, and **.mol2**
    - The mass of each element set in the **.smiles** or **.pdb** or **.mol** or **.mol2** file will be set via the first mass listed in the value of the dictionary that corresponds to the key to each element type and will be reflected in the output **.data** file (discussed later).
  + **.data** 
    - The element type of each atom type in the **.data** file will be identified via the mass of each atom type. The mass of each atom type in a **.data** is usually dependent on the force field used to generate the initial **.data** file, which means it allows the code to be easily extendable to a variety of different **.data** files that are initialized from different force fields you may just keep adding onto each list of masses. The code will exit if not all element symbols can be found via this mapping process, in which case you will just add the necessary info to the **mass\_map** dictionary.

Variable: **bondorder** Override tags: **N/A** Usage: **OPTIONAL**

* This option is only applicable for converting from ReaxFF to a fix-bond force field if you have read in a ReaxFF bond order file via the **bondfile** variable.
* **bondorder** is a Python dictionary that contains the bond order cut-offs for each bonding pair of elements. The bonding pairs are set as a tuple of element symbols as the keys and the value is a float value to set the minimum bond order for each bonding pair. NOTE that if you do not know a specific bond order cut-off, 0.3 is generally accepted as the universal cut-off. You may adjust the minimum bond order cut-off as desired or leave it as the 0.3 default.

Variable: **vdw\_radius\_scale** Override tags: **-vdw\_scale** Usage: **OPTIONAL**

* This option is only applicable when the read in **topofile** which does not have bonds in it and no ReaxFF specific **bondfile** is used or when using **bonds\_via\_distance\_override**, where all interatomic distances are computed and van der Waal radii are used to determine a max bond length cutoff to set new system bonds.
* The data type assigned to the **vdw\_radius\_scale** is a float value that allows the user to adjust the percent under or percent over each van der Waal radii cut-off. Setting **vdw\_radius\_scale** as 1.0, means all inter-atomic distance cutoffs are exactly the van der Waal radii. Whereas setting **vdw\_radius\_scale** as 0.8 or 1.2 “scales” the van der Wall cut off by decreasing the cutoff by 20% and increasing the cutoff by 20% respectively.
* The user may go into **LUNAR/src/bonds\_via\_distance.py** to find the Python dictionary **vdw\_radius** to add more elements if needed or to adjust the van der Waal radius for each element.
* Please note that the current implementation of interatomic distance searches may be very slow for large systems (>1000 atoms).

Variable: **boundary** Override tags: **N/A** Usage: **OPTIONAL**

* This option sets the boundary of the simulation cell to use when computing bonds via interatomic distance searching with bond length cutoffs set based on the **vdw\_radius\_scale** variable.
* The data type assigned to the **boundary** is a string with three boundary flags with the same meaning that LAMMPS uses for boundary flags. The string must contain 3 characters each separated by whitespace. Each character can either be ‘p’ for periodic or ‘f’ for non-periodic. Each location in the string sets the x, y, or z faces of the simulation cell to that boundary.

Variable: **bonds\_via\_distance\_override** Override tags: **N/A** Usage: **OPTIONAL**

* This option allows the user to search for bonds via inter-atomic distance searches with the max bond length cutoff set by the **vdw\_radius\_scale** variable and the simulation cell periodicity set by the **boundary** variable.
* By default, if the read in **topofile** has bonds defined, the code will use those bonds, however in some cases, it is desirable to redefine the bonds in the system and find the corresponding atom types. Setting **bonds\_via\_distance\_override** as True allows the ability to redefine the bonds through interatomic distance searching by overriding the default case of using the bonds defined by the **topofile** variable.
* It is important to note that when the **topofile** variable reads in a **.mol** or **.mol2** or **.pdb or .smiles** file format the simulation cell dimensions are set at the maximum span of the molecule ± 0.5 Å on each of the 6-faces of the simulation cell. This allows the user to adjust the **boundary** accordingly to create bonds across the boundary. The most obvious case is trying to build CNTs or graphene sheets/graphite with VMD’s Extensions -> Modeling -> Nanotube Builder and exporting the atom coordinates and bonds to a **.mol2** file. Where VMD will NOT make the carbon system periodically bonded, which is desired for an MD simulation. Setting the **boundary** variable and **bonds\_via\_distance\_override** variableaccordingly will allow for searching of interatomic distances across certain simulation cell boundaries to automatically create the periodic bonds since the simulation cell is set to 1 Å larger in the x, y, and z direction, where all van der Waal radii will be able to identify a bond at that distance. After performing this operation, it is always a good idea to visualize your system to make sure the bonds look proper. The **vdw\_radius\_scale** option can be used to adjust the max bond length cut-offs if needed when performing this kind of model-building procedure.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if any WARNINGS or ERRORS occurred while trying to assign atom types. Another thing to note is that some force fields have more advanced options for setting atom types and sometimes even charges that can be seen in the table of supported atom types where a trailing (T) or (F) or (Q) or a combination means there are Boolean flags in the Python code used to assign those atom types. Where (T) means the flag is True (F) means the flag is False and (Q) means the charge is also assigned at the atom typing step of the model-building procedure. These flags mainly show up for any of the IFF variants of force fields since some of the IFF atom types could be replaced with either CVFF or PCFF variants. So, by default the code will assign only the PCFF or CVFF atom types for PCFF-IFF or CVFF-IFF and to access the automatic atom typing for the IFF atom types explicitly the user must go to the **LUNAR/src/atom\_typing/typing/PCFF\_IFF.py** atom typing script or **LUNAR/src/atom\_typing/typing/CVFF\_IFF.py** atom typing script and adjust the Boolean flags within each of the files. The Boolean flags will be near the top and labeled which will accept True or False values like the other Boolean flags within LUNAR. It is also strongly encouraged after getting comfortable with using **atom\_typing.py**  for the user to look at the atom typing scripts found in the **LUNAR/src/atom\_typing/typing/** directory and the corresponding PowerPoints used to help set the topological considerations for each atom type in the force field the user mainly works with, because once familiar with the atom typing scripts that run in the backend, the user may adjust the scripts or add onto them for their preferences. Additionally, **atom\_typing.py** will be maintained and added onto in the future but is guaranteed to stay internally consistent with the first release such that the typing scripts will always stay compatible.

## Outputs

The code will write three outputs with the name set by the **topofile** “basename” and the append string set by **newfile** with the following extensions:

* **.data** which contains the atoms, bonds, charges, and simulation cell of the kept atoms that have been atom typed with RESET contiguous atomIDs.
* **.nta** which contains the atom types set by the code that maps onto the contiguous atomIDs found in the **.data** file using the “style id” header format. The user may manually adjust these if they desire.
* **.log.lunar** which contains the log of printouts to look back on if needed.

The **.data** and **.nta** file outputs can be used in **all2lmp.py** to get the final LAMMPS datafile with a force field assigned. NOTE that the code will always reset atomIDs even if no atoms are removed, thus it is best practice to use the **.data** and **.nta** outputs of **atom\_typing.py** as inputs into **all2lmp.py** and not the read in the **topofile** of **atom\_typing.py** and the outputted **.nta** file of **atom\_typing.py** as inputs to **all2lmp.py**, since the atom types in the **.nta** file may not map onto the atomIDs in the read in **topofile** of **atom\_typing.py**.

# Code: all2lmp.py

## Purpose

The purpose of **all2lmp.py** is to automatically find all topological (bonds, angles, dihedrals, and impropers) requirements to run an MD simulation in LAMMPS and then assign different force field (FF) parameters to a molecular system. This code either requires the user to use **atom\_typing.py** to find the atom types of their molecular system for the specific force field or for the user to manually assign atom types. A graph-theory approach is used to quickly find the angles, dihedrals, and impropers from bonding connectivity that is supplied via the inputs. The **all2lmp.py** code is a spinoff of the older **msi2lmp.exe**, which addresses all the limitations that **msi2lmp.exe** has. This means users of **msi2lmp.exe** should have an easy time adapting to **all2lmp.py** and enjoy a more automated molecular building tool that supports a variety of different chemical files including continued support for the **.car** and **.mdf** files that **msi2lmp.exe** would use. Users of **msi2lmp.exe** who have access to Materials Studio may also use Materials Studio for automatically assigning atom types and partial atomic charges. For those who do not have access to Materials Studio **all2lmp.py** supports enough other chemical file formats that can be generated in free or other open-source software and **atom\_typing.py** should be your first choice for automatic atom type assignment.

## Code Variables and Execution

**all2lmp.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **all2lmp.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 5.

A screenshot of a computer program

Description automatically generated

Figure 5: IDE view of some Python variables for all2lmp.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 all2lmp.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 6.

A screenshot of a computer screen

Description automatically generated

Figure 6: Command line override manual printed by running “python3 all2lmp.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **all2lmp.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **all2lmp.py** file. Examples of command line override:

python3 all2lmp.py -topo test1.data -nta test1.nta -class 2 -frc pcff.frc -dir testing\_directory

python3 all2lmp.py -topo test2.data -nta test2.nta -class 1 -frc cvff.frc -reset-charges T

python3 all2lmp.py -topo test3.data -nta test3.nta -class d -frc DREIDING.frc

python3 all2lmp.py -gui

python3 all2lmp.py -man

python3 LUNAR.py

python3 LUNAR.py 75

It is worth noting that there is a **use\_GUI** Boolean flag in **all2lmp.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 all2lmp.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 all2lmp.py -topo test1.mol -dir testing\_dir”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 all2lmp.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **all2lmp.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **all2lmp.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **all2lmp.py** GUI are shown in Figure 7. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **all2lmp.py** file, except for the **topofile** and the **nta\_file** variables which will remain as the default this code ships with or whatever the users has manually typed in **all2lmp.py**.

A screenshot of a computer

Description automatically generated

Figure 7: LUNAR/all2lmp.py GUI

The **LUNAR.py** and the **all2lmp.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **all2lmp.py** can be set to different values, whereas **GUI\_zoom** in **all2lmp.py** only affects the GUI that is launched from **all2lmp.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **all2lmp.py** GUI from **LUNAR.py** GUI. Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the “chemistry” of the system and can be generated with software like ChemDraw, Avogadro, BIOVIA Materials Studio, LAMMPS, MolView.org, and VMD. Open Babel can be used to convert most other chemical file formats to ones that are currently compatible with **all2lmp.py**.
* Compatible formats (The file extension is important, so the code knows how to read it):
  + **.mol** file (ChemDraw, Avogadro, MolView.org, and BIOVIA Materials Studio, etc, …)
  + **.mol2** file (ChemDraw, Avogadro, VMD, and BIOVIA Materials Studio, etc, …)
  + **.pdb** file (ChemDraw, etc,) OFFERS REDIMENTARY SUPPORT FOR .pdb files
  + **.data** file(LAMMPS datafile where the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section)
  + **.mdf** (BIOVIA Materials Studio file that will be paired with the **.car** file set in the **nta\_file** variable, to give continued support of the workflows that previously used **msi2lmp.exe**). A handy trick to remember that the **.mdf** file is paired with the **topofile** variable is that the “topo” part of **topofile** is associated with the word “topology” and topology in MD often is used to refer to the bonding connectivity, where the bonding connectivity is defined in the **.mdf** file and not the **.car** file.
  + NOTE that a **.data**, **.mdf/.car**, and **.mol2** file may have charges assigned already and the implications of **reset\_charges** variable.

Variable: **nta\_file** Override tags: **-nta** or **-n** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the “**n**ew **t**ype **a**ssignment” of atom types to map onto the atoms in the **topofile**. The format of this file will be discussed below, and the required or optional sections may be in any given **.nta** file. The **.nta** file extension is important and if the provided file does not have the **.nta** extension, **all2lmp.py** will exit with an error about providing an unsupported file type.
* Compatible formats (The file extension is important, so the code knows how to read it):
  + **.nta** file (file format specific to **all2lmp.py** that can be generated by **atom\_typing.py**)
  + **.car** file (BIOVIA Materials Studio file that will be paired with the **.mdf** file set in the **topofile** variable, to give continued support of the workflows that previously used **msi2lmp.exe**).
* The format of the **.nta** will now be discussed. The **.nta** file supports comments where anything trailing the “#” character will be treated as a comment and ignored. The first line will always be treated as a header and ignored. The **.nta** file is like a LAMMPS **.data** where it is broken down into sections, with each section having a specific style. The only difference is that after declaring each section based on a header keyword there CAN NOT be any white space between the header of the section and the start of information. Within each line, the **.nta** file is interpreted based on whitespace delimiters, where any number of whitespaces may exist but must have at least one whitespace to delimit different groupings of characters. There also exist simple keywords that are read “inline” where the keyword does not signify a section but rather applies some system-specific setting. A pseudo **.nta** file is shown below to get a feel for the format of the **.nta** file.

HEADER, this line will always be ignored

section1 style1 # anything following the “#” character will be ignored

1 cp

2 cp

3 hc

section2 style2

cp 0.1

hc -0.1

opt1a opt1b opt1c …

opt2a

:

:

* There only is one **required** section that must exist in the **.nta** file and it has two different styles: **“style id”** or **“style type”**, which sets that atom type to map onto the atoms that are in the **topofile**. The **“style id”** header implies that the atom type set in this section will be mapped onto the atomIDs found in the **topofile**. The **“style id”** method is currently compatible with **topofile**’s with the **.mol** or **.sdf** or **.mol2** or **.data** or **.pdb** file extensions since all these file formats have atomIDs set in the file. The **“style type”** header implies that the atom type set in this section will be mapped onto the atomTypeIDs found in the **topofile**, where an atomTypeID is only in an already built LAMMPS **.data** files and corresponding to the atom typeID (seen in the Masses and Pair Coeffs section of a **.data** file). This means the **“style type”** method is only compatible with a **.data topofile** extension. **all2lmp.py** will check that every atom in the **topofile** has an atom type set in the **.nta** file and will exit with an error to avoid possible misuse of the **“style type”** method and other **topofile** formats that do not have a **.data** extension. The **“style type”** method will allow users one method of converting a LAMMPS datafile from one force field assignment to another. Alternatively, to change force fields one may use **atom\_typing.py** on a LAMMPS **.data** file and then run the outputs in **all2lmp.py**. An example of the **“style id”** and **“style type”** sections is shown below (please note you may use **“style id”** or **“style type”**, but not both in the same **.nta** file).

Example of “style id” format of setting atom types, which will be mapped onto atomIDs

style id # atomID atom-type

1 cp

2 hc

:

Example of “style type” format of setting atom types, which will be mapped onto atomTypeIDs

style type # atomTypeID atom-type

1 cp

2 op

:

* The following sections and/or keywords are **optional** and may be added to the **.nta** file to perform certain system-specific tasks in assigning certain parts of the force field:
  + **“equivs”** section which allows for a user to quickly change some of the atom types in either the **“style id”** section or the **“style type”** section to a different atom type. The intended use case for the **“equivs”** section is mainly for pairing with **atom\_typing.py** and the usage of the **general:N** force field atom types. However, users may find the **“equivs”** section useful for quickly changing certain atom types out for others to see if a given **.frc** force field file has parameters when using a different atom type in place of another. Note that similar things can also be accomplished with text editors using find and replace methods (it is up to the user based on their preference, but if using a **.nta** from **atom\_typing.py** with the **general:N** typing method, it is recommended to use the **“equivs”** option such that a written “history” of setting the atom type to local topology exists to look back onto if needed later). Examples are given below of the **“equivs”** section (Note that the **“equivs”** section may appear anywhere in the **.nta** file, i.e., above or below the **“style id”** or **“style type”** section). Also note that usage of the **“charge nta”** section which will be discussed later, that if using the **“equivs”** method and the **“charge nta”** method to apply charges to the system, the atom type that is in column two of the **“equivs”** section will be used.

Example of “equivs” section mapping all hc in the “style id” section to hpan

style id

1 c2

2 cp

3 hc

4 hc

:

equivs

hc hpan

:

Example of “equivs” section on ‘general:1’ atom typing from **atom\_typing.py**

equivs

C04-(1;C04,1;C63,2;H01) c2

C04-(1;C04,3;H01) c3

C63-(1;C04,2;C63) cp

H01-(1;C04) hc

:

style id

1. C04-(1;C04,1;C63,2;H01)
2. C04-(1;C04,3;H01)
3. C63-(1;C04,2;C63)

4 H01-(1;C63)

:

#--------------------------------------------------------------------------------------------#

# When using “charge nta” and using “equivs” option, the atom type set in #

# Column two of the “equivs” section will be used in the “charge nta” section #

#--------------------------------------------------------------------------------------------#

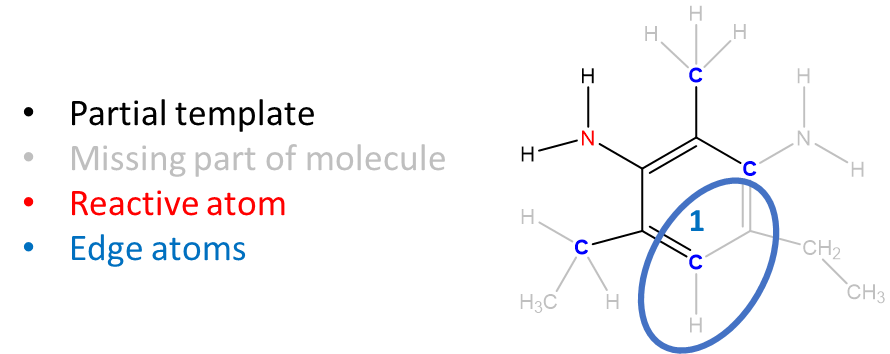
charge nta

hc 0.1111

cp 0.2222

:

* **“edge id”** section allows users to “append” dummy atoms to atomIDs that are deemed to be edge atoms for building “*fix bond/react”* (REACTER) templates. The reason this section exists is because most of the force fields provided in LUNAR apply charges via the “bond-increment” method, where the connectivity of atoms is used to apply the force field charges. However, when building *“fix bond/react”* templates most of the time the molecular topology must be reduced leaving undercoordinated atoms, which will HAVE the incorrect charge if using the “bond-increment” method to apply a charge to the system (please see the **reset\_charges** option for **all2lmp.py** on turning on or off this charging method). It is possible to manually adjust the charge on these edge atoms for *“fix bond/react”* templates, however, the usage of the **“edge id”** section may also be used when first building the **.nta** files for *“fix bond/react”* simulations. When using the **“edge id”** section the atom types assigned to the atomID will be iterated through when assigning charges to account for the atoms that would have been there if the topology was not reduced. The **“edge id”** section may be placed anywhere in the **.nta** file. An example is provided below illustrating a partial reaction template for a DETDA molecule, where the **“edge id”** example exists for atomID 1. When **all2lmp.py** is supplied with the **“edge id”** example below it will assign charge as if the atom type hc and cp were part of the partial template and thus the charge of the edge atom will be consistent with the charge of the full DETDA molecule.



Example of “edge id” to correctly charge edge atom by adding dummy cp and hc atoms

style id

1 cp

2 cp

3 hc

:

edge id

1 cp hc

:

* Certain force fields exist that do not provide an algorithmic way of applying charge to a system (such as the “bond-increment” or the “Gasteiger” method), but rather depend on manually setting the charge for each atom in the system. Such as Clay-FF and the IFF atom types of PCFF-IFF and CVFF-IFF. Manual setting charges can be a very time-consuming and error-ridden process for large systems. **all2lmp.py** offers some level of support for this via the **“charge id”** or **“charge type”** or **“charge nta”** sections that may be added to the **.nta** file. When using the **“charge id“** or **“charge type”** or **“charge nta”** sections, the charge added to the system is performed after the **reset\_charges** option in **all2lmp.py**. For example, if you are trying to build a LAMMPS datafile with PCFF-IFF and have two molecules: one using PCFF atom types and the other using IFF atom types, you may have **reset\_charges** set to True so that charges will be assigned to the PCFF atom types (IFF atom types will remain having zero charge since no “bond-increments” exist for those types). Then use any of the **“charge id“** or **“charge type”** or **“charge nta”** sections to add the charges to the IFF atom types. The **“id”** or **“type”** or **“nta”** styles are meant to be consistent between different sections of the **.nta** file, such that **“style id”** and **“charge id”** apply charges based on atomID. Similarly, **“style type”** and **“charge type”** apply charges based on atomTypeID (only compatible with LAMMPS **.data** file). The **“charge nta”** method means set charges based on atom type characters (i.e., cp, hc, c2 …). Below are examples of using each of the **“charge id“** or **“charge type”** or **“charge nta”** sections.

Example of “charge id” to add a charge to atomIDs

style id # This section could be style id or style type

1 c2

2 cp

3 hc

4 hc

:

charge id

1 0.1111 # will set the charge of atomID-1 as 0.1111 (EXAMPLE ONLY)

2 0.2222 # will set the charge of atomID-2 as 0.2222 (EXAMPLE ONLY)

:

Example of “charge type” to add a charge to atomTypeIDs (.data files ONLY)

style id # This section could be style id or style type

1 c2

2 cp

3 hc

4 hc

:

charge type

1 0.1111 # will set the charge of all atomTypeID-1 atoms as 0.1111 (EXAMPLE ONLY)

2 0.2222 # will set the charge of all atomTypeID-2 atoms as 0.2222 (EXAMPLE ONLY)

:

Example of “charge nta” to add charge to atom types

style id # This section could be style id or style type

1 c2

2 cp

3 hc

4 hc

:

charge nta

c2 0.1111 # will set the charge of all c2 atoms to 0.1111 (EXAMPLE ONLY)

hc 0.2222 # will set the charge of all hc atoms to 0.2222 (EXAMPLE ONLY)

:

* Ideally when setting the charge of atoms, the net charge of most molecules (if not complete molecular systems with differing molecules), should have a net zero charge, due to how LAMMPS has implemented the long-range kspace solvers. If manually setting the charge of that atom using the **“charge id“** or **“charge type”** or **“charge nta”** creates a system having a non-zero net charge it can be difficult to track down what charge added was incorrect. Ideally one would track down why the system is not charge neutral. However, if a high throughput method is needed to quickly ensure the charge of the system is enforced to be zero **all2lmp.py** offers some keywords to ensure this. The responsible usage of these options falls upon the users of LUNAR and LUNAR is not to be held responsible for careless usage of the options being described below. The following keyword options may be used to ensure a net zero charge by applying a fixed charge to a different grouping of atoms, **“neutralize system charge all”** or **“neutralize system charge zero”** or **“neutralize system charge bond-inc”** or **“neutralize system charge user-defined”**.
  + **"neutralize system charge all"** will make the system charge neutral by adding a fixed value to "all" atoms in the system
  + **"neutralize system charge zero"** will make the system charge neutral by adding a fixed value to atoms that have a "zero" charge in the system
  + **"neutralize system charge bond-inc"** will make the system charge neutral by adding a fixed value to atoms that are defined as "bond-inc" atoms. "bond-inc" atoms will be "all" atoms in the system if not using any of the optional "charge <opt>" headers. If using any of the "charge <opt>" headers, "bond-inc" atoms are defined as atoms that have not had charge set via the "charge <opt>" headers.
  + **"neutralize system charge user-defined"** will make the system charge neutral by adding a fixed value to atoms that are defined as "user-defined" atoms. "user-defined" atoms will be atoms in the system using any of the "charge <opt>" headers and will ONLY have a fixed charge added to the atoms defined by the "charge <opt>" headers.

If **reset\_charges** is True, the “bond-increment” method/algorithm already ensures a charge-neutral system and usage of the **“neutralize system charge all”** or **“neutralize system charge zero”** or **“neutralize system charge bond-inc”** or **“neutralize system charge user-defined”** is not required and usually should not be used, except in special cases. Please be cognizant of using the **“edge id”** section, **reset\_charges** as True, and any of these neutralize charge system keywords for generating *“fix bond/react”* reduced templates, because a reduced topology template for *“fix bond/react”* SHOULD have a non-zero net charge due to how *“fix bond/react”* superimposes the templates onto the main molecular system (NOTE see *“fix bond/react” custom\_charges* options to change this type of operation). These special neutral charge keywords may be placed anywhere within the **.nta** files, with a couple of examples provided below.

Example of “charge nta” to add charge to atom types and neutralize system charge

style id # This section could be style id or style type

1 c2

2 cp

:

charge nta

c2 0.1111 # will set charge of all c2 atoms to 0.1111 (EXAMPLE ONLY)

hc 0.2222 # will set the charge of all hc atoms to 0.2222 (EXAMPLE ONLY)

:

neutralize system charge all

Example of “charge nta” to add charge to atom types and neutralize system charge

style id # This section could be style id or style type

1 c2

2 cp

:

charge nta

c2 0.1111 # will set charge of all c2 atoms to 0.1111 (EXAMPLE ONLY)

hc 0.2222 # will set the charge of all hc atoms to 0.2222 (EXAMPLE ONLY)

:

neutralize system charge zero

* If the **nta\_file** variable is set to‘types\_from\_pdb.nta’, **all2lmp.py** will look for atom type strings in the location of the atom name column of the **.pdb** file and then apply those atom types to the system. This method requires manual editing of the **.pdb** file and is mainly meant for an integration method with packmol since packmol will maintain the atom name column. Note the following about manual editing of **.pdb** files:
  + Are interpreted based on character indexes and not white space delimiters, which can make manually adjusting **.pdb** files error-prone. It is recommended to look up the guidelines for **.pdb** file format BEFORE attempting to manually edit ANY **.pdb** file.
  + Please look in the EXAMPLES/packmol\_pdb\_methods/ directory for three different integration methods with packmol and **LUNAR**.
* If the **nta\_file** variable is set to‘topofile’, **all2lmp.py** will create a string for the **nta\_file** variable that is the basename of the **topofile** with the extension **.nta**. Majority of the time the **topofile** and **nta\_file** will be imported directly from **atom\_typing.py**, thus both files have will have consistent basenames.
* If the **nta\_file** variable is set to a string that is an **ff\_name** that **atom\_typing.py** supports, **atom\_typing.py** will be called from within **all2lmp.py**. This can speed-up building LAMMPS datafiles because it can be a single step process to go from a chemical file format to a LAMMPS datafile format. There are however cases where it is still desired to run **atom\_typing.py** first, edit the **.nta** file and then pass the files into **all2lmp.py**. The currently supported strings that can be set in the **nta\_file** variable are:
  + ‘PCFF-IFF’ (Interface force field based on PCFF)
  + ‘PCFF’ (Polymer consistent force field)
  + ‘compass’ (Compass force field)
  + ‘CVFF-IFF’ (Interface force field based on CVFF)
  + ‘CVFF’ (consistent valence force field)
  + ‘Clay-FF’ (Clay force field)
  + ‘DREIDING’ (DREIDING force field)
  + ‘OPLS-AA’ (OPLS all-atom force field which currently has minimal parameters)
  + Which will then call **atom\_typing.py** to set atom types based on that force field type. Additionally, a ‘-q’ can be appended to any of the force field names above to set the charges via the Gasteiger method found in **atom\_typing.py**, which will then automatically set the **reset\_charges** variable in **all2lmp.py** to False, to use then use Gasteiger charging. For example, setting the **nta\_file** variable to ‘DREIDING-q’ will set atom types based on DREIDING atom types and set the system charge using the Gasteiger method.
  + This method does not make use of all **atom\_typing.py** features, where variables for **atom\_typing.py** are defined in the following manner:
    - Loaded from **atom\_typing.py**:
      * **mass\_map**
      * **bondorder**
      * **maxbonded**
      * **boundary**
      * **vdw\_radius\_scale**
      * **bonds\_via\_distance\_override**
      * **pdb\_file**
      * **chargefile**
      * Thus, changing what there variables are set to in **atom\_typing.py** will adjust the defaults of how **all2lmp.py** will call **atom\_typing.py**.
    - Internally set by **all2lmp.py**:
      * **topofile** (name of **topofile** read into **all2lmp.py**)
      * **bondfile =** ‘n.u.’ (not supported to read ReaxFF **bondfile**, yet)
      * **bondfile =** ‘typed\_from\_within\_all2lmp’
      * **ff\_name** (name of **nta\_file** read into **all2lmp.py**)
      * **reset\_charges** (based on if ‘-q’ is append on the end of the **nta\_file** name)
      * **parent\_directory** (name of **parent\_directory** read into **all2lmp.py**)
  + After **atom\_typing.py** is run **all2lmp.py** will then update the **topofile** and **nta\_file** variables accordingly to read in the newly atom typed system. All these variable changes will be printed out when **all2lmp.py** is performing these tasks.

Variable: **frc\_file** Override tags: **-frc** or **-f** Usage: **REQUIRED**

* Will read in the file that is assigned to the Python variable or the override tag. This file contains the force field parameters that will be applied to the “chemistry” defined by the **topofile** and **nta\_file**. The force field file MUST be in BIOVIA Materials Studio **.frc** format and **all2lmp.py** will check if the file extension is **.frc**.
* **all2lmp.py** will search for the parameters using the exact atom type and then move to search using equivalences if needed, and then move to search with auto-equivalent types if needed and **use\_auto\_equivalence** is True. This means **all2lmp.py** will always use the most specific parameter possible and only move to use more general parameters if needed. One case where this logic can get confusing is with parameters that use the “\*” or “\*N” wildcard operators found in certain **.frc** formatted force fields or sections in the **.frc** formatted force field file. In such cases, **all2lmp.py** will first attempt to not use any wildcards and search for the most specific match and only move to using wildcards if needed. Once searching using wildcards **all2lmp.py** searching algorithms are designed to minimize the number of wildcard operators used to find a match, such that **all2lmp.py** will first only search for a single wildcard operator and move to search for a match using two wildcard operators if needed, and then move to search use three wildcard operators if needed. An example is provided below to give an example:
  + Assume **all2lmp.py** is trying to find parameters for a **c5 op hc** angle:
    - **Attempt1-exact atom types:** look for exact **c5 op hc** angle parameters in forward and reverse permutations
    - **Attempt2-equivalent lookup table:** convert **c5**->**cp**; **op**->**op**; **hc**->**hc** so look for an **cp op hc** angle parameters in forward and reverse permutations
    - **Attempt3-auto-equivalent lookup table:**
      * Convert **c5**->**cp\_**; **op**->**op\_**; **hc**->**hc\_** so look for auto-equivalent forms + wildcard options.
        + **Attempt3a-exact auto-equivalent:** look for **cp\_ op\_ hc\_** angle parameters in forward and reverse permutations
        + **Attempt3b-wildcards auto-equivalent:** look for **\* op\_ hc\_** angle parameters in forward and reverse permutations
        + **Attempt3c-wildcards auto-equivalent:** look for **cp\_ op\_ \*** angle parameters in forward and reverse permutations
        + **Attempt3d-wildcards auto-equivalent:** look for **\* op\_ \*** angle parameters in forward and reverse permutations
  + The example above illustrates how many different permutations **all2lmp.py** is coded to handle and has been coded to guarantee the most specific parameters are used. Thus, modelers can worry about other details of building MD models. Note that for Dihedral and Improper searching, there is a maximum possible usage of three wildcard positions and **all2lmp.py** is coded to handle all different numbers of permutations one can imagine (especially for impropers). There are cases in which **all2lmp.py** may attempt around 50 different permutations of using and not using wildcard lookups so if **all2lmp.py** cannot find a suitable parameter, it means no parameters exist in the given **.frc** file.
  + One last note about finding parameters versus atom type selection and dealing with the “rhetoric” of I choose to use this atom type over another since the “equivalences” in the equivalence table are the same. Notice that **all2lmp.py** will first attempt to look up parameters without applying any equivalences, so in some cases even though two atom types may have the same equivalence, it is possible that a first search using the exact type can yield different results than looking up the “equivalent” type. It is a similar scenario for parameter lookups using the auto-equivalent types. **atom\_typing.py** can be used to reduce this type of atom type selection errors, but if a person is insistent on setting their atom types, and has found themselves justifying one atom type verse another simply because the equivalences table is the same between the atom types, they should take time to think which atom type should be used.

Variable: **assumed** Override tags: **-asm** or **-a** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the sections of different atom type/elemental configurations which can be used to generalize certain parameters. The purpose of this file is to add another layer of generalization to parameters to help automatically assign very general and assumed parameters if the force field read in by the **frc\_file** does not currently support parameters for certain systems being defined by the **topofile** and **nta\_file**. The usage of the **assumed** file is controlled by the **use\_assumed\_auto\_fill**, where **use\_assumed\_auto\_fill** should ALMOST ALWAYS be False. Typically, if **all2lmp.py** has a lot of printouts of **“Unable to find ….”**, it means a certain atom type is wrong or should be replaced with a more general atom type such that parameters can be found. However, there are unique cases where even generalizing all atom types and using a force field defined by **frc\_file** and the system defined by **topofile** and **nta\_file**, **all2lmp.py** may not be able to provide a completely parameterized system. Specifically, when converting from a ReaxFF simulation to fix-bond force field simulation. For such cases using **assumed** and **use\_assumed\_auto\_fill** can provide an automated approach to getting parameters. The responsible usage of this option again falls at the discretion of the users of LUNAR and LUNAR DOESN’T hold any responsibility for careless usage of this option.
* The way **all2lmp.py** is written is to first attempt to find parameters based on the exact atom types, then move to equivalences if needed, then move to auto-equivalences (if **use\_auto\_equivalence** is True) if needed, then move to using the generalized assumed mapping set by the **assumed** file (if **use\_assumed\_auto\_fill** is True). This means **all2lmp.py** will ALWAYS be using the most specific parameter possible and only keeps moving to a more general set of parameters if needed. The **assumed** file option provides the user of LUNAR the ability to automatically assign parameters using certain assumptions.
* The goal of the **assumed** file is to generalize each parameter type down to basic elements (i.e., a cp-cp bond to a C-C bond, a cp-cp-hc angle to a C-C-H angle, …). This generalization has a lot of assumptions but does offer the ability to automatically insert parameters, with the assumption of general transferability of parameters and that **“some sort of parameter is better than no parameter”**.
* The file format will now be discussed, where anything following the “#” character is understood to be a comment. The format of the file is based on header sections with one whitespace above and below the header. Under each header the user will supply the elemental compositions and then the atom types that should be mapped onto that elemental composition. **all2lmp.py** will check the equivalent and auto-equivalent sections of the **.frc** file to try to find the map (please note for setting these atom types you must look up in the bond coeffs, angle coeffs, etc sections of the **.frc** file and not just use the equivalent or auto-equivalent tables, since it is not always guaranteed that a parameter exists between any random pairing(s) of atom types/equivalents/auto-equivalents). An example of the **assumed** file is provided below.

Bond Coeffs # Map of elements of a bond-to-bond atom type

C C c\_ c\_

C H c\_ h\_

Angle Coeffs # Map of elements of an angle-to-angle atom type

C C C c\_ c'\_ c\_

O O O \* o\_ \*

Dihedral Coeffs # Map of elements of a dihedral-to-dihedral atom type

C C C C \* c=\_ c=\_ \*

O O O O \* o\_ o\_ \*

Improper Coeffs # Map of elements of an improper-to-improper atom type

C C C C \* c'\_ \* \*

O O O O \* o\_ \* \*

# A more verbose ready to use example for PCFF-IFF can

# be found in LUNAR/frc\_files/general\_assumed\_equivs.coeffs

Variable: **add2box** Override tags: **-add2box** or **-a2b** Usage: **REQUIRED**

* **add2box** is a Python float or int value that can be positive or negative and will be used to adjust each of the 6-faces of the simulation cell box dimensions (angstroms). If the read-in **topofile** has none-zero image flags or image flags are derived that are none-zero **add2box** will not be used since it is generally a bad idea to adjust the simulation cell dimensions if there are periodic molecules spanning the current box. The box dimensions of each supported file format defaults are provided below:
  + **.mol** or **.sdf** or **.mol2** or **.pdb** or **.car/.mdf** (without box defined) will have image flags set to zero and simulation cell set at the extent of atoms with a 0.5-angstrom buffer.
  + **.data** or **.car/.mdf** (with box defined) will leave the simulation cell dimensions as is. A **.data** file will have the image flags left as is and a **.car/.mdf** (with box defined) will have the image flags derived.
* If **add2box** is set as 0 no modification to the default simulation cell dimensions will occur.

Variable: **shift** Override tags: **-sx** or **-sy** or **-sz** Usage: **REQUIRED**

* Is a Python dictionary with the keys set as ‘x’, ‘y’, and ‘z’ which controls the optional shift to apply to a molecular systems. The values of the dictionary are int or float values to set the shift in each direction. This shifts the atoms and the simulation cell. If ‘x’, ‘y’, and ‘z’ are set to ZERO, this option is not used.

Variable: **rotate** Override tags: **-rx** or **-ry** or **-rz** Usage: **REQUIRED**

* Is a Python dictionary with the keys set as ‘x’, ‘y’, and ‘z’ which controls the optional rotation (in degrees) that can be applied to the molecular system about the x-axis, y-axis, and z-axis respectively. The values of the dictionary are int or float values to set the rotation. To use this option the system must be none-periodic (i.e. all image flags are zero). Before the system is rotated, it is centered about (0,0,0), rotated, and then shifted back to its original location. After the system has been rotated, the simulation cell is redefined with the same amount of “x-padding”, “y-padding”, and “z-padding”, between the atoms and the simulation cell before the rotation. If ‘x’, ‘y’, and ‘z’ are set to ZERO, this option is not used.

Variable: **ignore\_missing\_parameters** Override tags: **-ignore** or **-i** Usage: **REQUIRED**

* **ignore\_missing\_parameters** is a Python variable (True or False) to ignore warnings about missing parameters. If the molecular system cannot be fully parameterized (i.e. if a parameter could not be found for every atom type, bond type, angle type, dihedral type, improper type or partial charge in the frc\_file), the code will ZERO all parameters before writing the LAMMPS datafile. To override this behavior set **ignore\_missing\_parameters** as True, to get a parameterized LAMMPS datafile with ZERO's only in the missing parameter locations.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **all2lmp.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the **topofile** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **all2lmp.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the **topofile** path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_IFF’** and **topofile == ‘detda.data’):**
        + basename = ‘detda\_IFF’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘IFF-:’** and **topofile == ‘detda.data’):**
        + basename = ‘IFF-detda’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as ‘ANYTEXT’. For example:
      * **(newfile = ‘detda\_renamed’** and **topofile = ‘detda.data’):**
        + basename = ‘detda\_renamed’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.
    - **Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.

Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **all2lmp.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix-bond force field the ‘full’ atom style is the best choice, whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice.

Variable: **ff\_class** Override tags: **-class** or **-c** Usage: **REQUIRED**

* Is a Python string or int variable that sets the “class” of the force field. Different force fields require different amounts of information and parameters, where **ff\_class** informs **all2lmp.py** how to find the topologies such as bonds, angles, dihedrals, and impropers and then how to look up corresponding parameters that are read-in via the **frc\_file** variable. Please note that if you provide an inconsistent definition of **ff\_class** and **frc\_file**, **all2lmp.py** will exit with an error that the **ff\_class** is inconsistent with the data in the **frc\_file**. The following **ff\_class** options are available and what force fields typically use these class definitions:
  + 0 = class0 (int data type - FF files: opls-AA EXPERIMENTAL)
  + 1 = class1 (int data type - FF files: cvff-IFF, cvff, clayff)
  + 2 = class2 (int data type - FF files: PCFF-IFF, PCFF, compass)
  + ‘d’ = DREIDING (str data type - FF file: all2lmp\_dreiding.frc)
  + ‘r’ = reaxFF (str data type - FF files: all2lmp reaxff specfic .frc file)
  + ‘s1’ = skeleton datafile for class1, where skeleton means a datafile without coeffs in it
  + ‘s2’ = skeleton datafile for class2, where skeleton means a datafile without coeffs in it
* The ‘s1’ and ‘s2’ options will find all N-body topologies and the coeffs typeIDs. The topologies will have atomIDs in the Bonds, Angles, Dihedrals, and Impropers sections that match the ordering of the atom types in the coefficient comment. The purpose of the skeleton option is to allow users to generate a complete LAMMPS datafile without having **all2lmp.py**, and attempt to look up parameters from a **frc\_file** since **all2lmp.py** would be (and is currently) limited to only the **.frc** files it ships with or **.frc** files others may have. There are proprietary **.frc** files out there, that require purchasing to use. The usage of the skeleton options is viewed as a rare use case but is there to offer support to users who wish to generate their parameters and their atom typing scheme. Then use **all2lmp.py** to generate as much as possible and only make the user insert the parameters of their choice since all of the unique coeffs typeIDs will be found by **all2lmp.py** and placed in the written datafile with comments. The skeleton options may also offer a level of support for generating coarse-grained models, however, tools like moltemplate and EMC already offer support for coarse-graining and may be worth looking into if coarse-grained models are desired.

Variable: **use\_auto\_equivalence** Override tags: **-auto-equivs** or **-auto-e** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to search the auto-equivalent parameters if needed. **all2lmp.py** will search for the parameters using the exact atom type then move to search using equivalences if needed, and then move to search with auto-equivalent types if needed if **use\_auto\_equivalence** is True. Setting **use\_auto\_equivalence** as False will default **all2lmp.py** to find parameters as **msi2lmp.exe** does. It is recommended to have **use\_auto\_equivalence** as True so more parameters can be automatically found.

Variable: **use\_morse\_bonds** Override tags: **-morse-bond** or **-mb** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to use Morse bonds over harmonic bonds, where **all2lmp.py** will search for the Morse bond parameters instead of harmonic bond parameters. This option is currently only compatible with **ff\_class** 1 or ‘d’ since most class1 force fields already have Morse bond parameters and the DREIDING **.frc** file supplied with LUNAR also has the Morse bond parameters defined. Please note that **use\_morse\_bonds** and **use\_auto\_equivalences** are compatible, thus if you are attempting to search for Morse bonds and no exact match or equivalent match can be found, and if **use\_auto\_equivalence** is True the auto-equivalences will be searched as well. If you would like to add Morse bonds to a system that uses any other **ff\_class** than 1 or ‘d’ or would like to add Morse bonds after performing LAMMPS simulations investigate **LUNAR/auto\_morse\_bond\_update.py**.

Variable: **reset\_molids** Override tags: **-reset-molids** or **-rm** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to reset the molIDs of each atom. The “clusters” of atoms are determined via the bonding connectivity, where the criteria for atoms to be part of the same “cluster” is that the atoms must be linked by at least one covalent bond to the “cluster”. The clusters are then sorted by number of atoms, where the largest number of atoms is identified as cluster one, then molIDs are incremented and are assigned to each of the remaining clusters (i.e., the largest cluster of atoms will have molID 1, and then the smallest cluster will have a molID of NCLUSTERS found). Please keep in mind that **atom\_style** ‘full’ or ‘molecular’ is the only **atom\_style**’sthat LUNAR supports that use the molIDs.

Variable: **reset\_charges** Override tags: **-reset-charges** or **-rq** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to reset charges via the bond-increment method. The bond-increments are specified in the read-in **frc\_file** and are searched by first looking for an exact atom type match, then moving to equivalent match if needed, and finally moving to auto-equivalent match if needed and **use\_auto\_equivalence** is True. MOST of the **.frc** formatted force fields have bond increments, however, force fields like DRIEDING do not use the bond-increment method and thus no bond increments will be supplied in the **.frc** file.
* Please note the following about **topofile** extensions and charge information, since **all2lmp.py** WILL maintain the read-in charge and only adjust the charge if **reset\_charge** is True.
  + **.mol** files are read-in with all per-atom charges set as ZEROs.
  + **.mol2** and **.pdb** files are initialized with all per atom charges as ZEROs and then updated during the reading of the file. So, if you would like to keep your system charged from a **.mol2** and **.pdb** file, **reset\_charges** should be False.
  + **.data** files are read-in and the charge that is in the **.data** is left as is, unless **reset\_charges** is True, so if you want to maintain the charge in your system while reading in a **.data** file **reset\_charges** should be False.
  + **.car/.mdf** files are read-in and the charge that is in the **.car/.mdf** is left as is, unless **reset\_charges** is True, so if you want to maintain the charge in your system while reading in a **.car/.mdf** file **reset\_charges** should be False.

Variable: **use\_assumed\_auto\_fill** Override tags: **-assumed-equivs** or **-ae** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to use the **assumed** file unique to **all2lmp.py**, to supplement the equivalent and auto-equivalent parameter search by adding another layer of generalization for atom types to parameter mapping. The usage of the **assumed** file is controlled by the **use\_assumed\_auto\_fill**, where **use\_assumed\_auto\_fill** should ALMOST ALWAYS be False. Typically, if **all2lmp.py** has a lot of printouts of **“Unable to find ….”**, it means a certain atom type is wrong or should be replaced with a more general atom type such that parameters can be found. However, there are unique cases where even generalizing all atom types and using a force field defined by **frc\_file** and the system defined by **topofile** and **nta\_file**, **all2lmp.py** may not be able to provide a completely parameterized system. Specifically, when converting from a ReaxFF simulation to fix-bond force field simulation. The responsible usage of this option again falls to the discretion of the users of LUNAR and LUNAR DOESN’T hold any responsibility for careless usage of this option.
* The way **all2lmp.py** is written is to first attempt to find parameters based on the exact atom types, then move to equivalences if needed, then move to auto-equivalences (if **use\_auto\_equivalence** is True) if needed, then move to using the generalized assumed mapping set by the **assumed** file (if **use\_assumed\_auto\_fill** is True). This means **all2lmp.py** will ALWAYS be using the most specific parameter possible and only keeps moving to more general if needed. The **assumed** file option provides the user of LUNAR the ability to automatically assign parameters using certain assumptions.

Variable: **write\_txt\_comments** Override tags: **-write-comments** or **-wc** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write a **.txt** file with the name set by the **topofile** basenameand the **newfile** extension. The original intent of the **write\_txt\_comments** option was to produce a log file that provides the developer of **all2lmp.py** with information to check against the **.frc** file and ensure that **all2lmp.py** is assigning parameters correctly. It is recommended to have **write\_txt\_comments** as True so that parameter assignments are trackable in case later-stage manipulation of parameters is needed. Furthermore, since **all2lmp.py** offers the four possible parameter searches (exact, equivalent, auto-equivalent, and assumed) and it is useful to distinguish parameters from “exact” and “equivalent” searches vs the parameters from the “auto-equivalent” and “assumed” searches. The parameters that are from the “exact” and “equivalent” searches are fully parameterized interactions unique to that specific force field. Whereas the parameters found via the “auto-equivalent” search are generalized parameters that are not unique to any specific force field and the parameters found by the “assumed” method are user-defined based on an assumed mapping of atom types to elements (regardless of the hybridization state of the atoms).

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file and thepossible LAMMPS **.lmpmol** molecule file (see **write\_bond\_react**). LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and helps manual creation of the input files (i.e., reaction templates). If users of LUNAR want to manually generate the required files for using *“fix bond/react”*, it is recommended to have the **include\_type\_labels** option as True since this will help in the manual creation processes of these files. However, LUNAR offers another tool called **bond\_react\_merge.py** that automates as much as possible in generating *“fix bond/react”* templates. It is recommended to use **bond\_react\_merge.py** instead of manually creating templates since manual creation can be a very error-ridden, slow process. Also note that **bond\_react\_merge.py** can add type labels into the written merged files even if the read-in files DO NOT have type labels defined, since all of LUNAR has a consistent commenting scheme.

Variable: **write\_bond\_react** Override tags: **-write-bond-react** or **-wbr** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write a LAMMPS molecule file that is suitable for using the *“fix bond/react”* command. The molecule filename will be set by the **topofile** basename and the **newfile** extension and will have a **.lmpmol** file extension and an additional file will be written with the **.ecoeffs** file extension that will contain the parameters for the **.lmpmol** file.
* The **.lmpmol** file will be written with the following sections:
  + Types
  + Charges
  + Coords
  + Bonds (if any are present)
  + Angles (if any are present)
  + Dihedrals (if any are present)
  + Impropers (if any are present)
* The written **.lmpmol** file may need modifications to “unify” or “merge” all the CoeffTypeIDs (i.e., Masses, Pair Coeffs, Bond Coeffs, etc), with LAMMPS **.data** file for direct usage in a *“fix bond/react”* simulation. See **LUNAR/bond\_react\_merge.py** which can automate this task.
* NOTE that other LAMMPS commands such as *“fix deposit”*, *“fix pour”*, *“fix rigid/small”*, *“fix shake”*, *“fix gcmc”*, *“create\_atoms”*, and *“atom\_style template”*, currently also use the LAMMPS molecule file and that **write\_bond\_react** in **all2lmp.py** or **LUNAR/bond\_react\_merge.py** can be used to generate files for those LAMMPS commands. The naming of the variables and codes in LUNAR is because they were built for using *“fix bond/react”*, which is likely the most complicated LAMMPS command out of all the LAMMPS commands that use the LAMMPS molecule file to automate the creation of files for. The developer of LUNAR also works a lot with the *“fix bond/react”* command.

Variable: **print\_options** Override tags: **-man** or **-opt** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to print the command line override manual and exit the code. True will print out the command line override manual and exit, whereas False will allow the code to run.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if any WARNINGS or ERRORS or Unable to find parameters occurred while trying to find the topologies and assign the force field to your system.

## Outputs

The code will write minimally one output, with options to write up to four outputs with the name set by the **topofile** “basename” and the append string set by **newfile** with the following extensions:

* + **.data** which contains the atoms, bonds, angles, dihedrals, impropers, charges, all FF parameters, and simulation cell. This file is ready for a LAMMPS simulation or to be continued processing in LUNAR using **bond\_react\_merge.py** or **cell\_builder.py** or **auto\_morse\_bond\_update.py** or **add\_pi\_electrons.py**. This file will always be generated after running **all2lmp.py**.
  + **.txt** which contains verbose comments about the FF parameters and comments about the charging methods. This file will be generated after running **all2lmp.py** if **write\_txt\_comments** is True.
  + **.log.lunar** which contains the printouts of **all2lmp.py**.
  + **.lmpmol** which is a LAMMPS molecule file and will be generated after running **all2lmp.py** if **write\_bond\_react** is True.
  + **.ecoeffs** which is a LAMMPS datafile format and contains the energy parameters for **.lmpmol** file and will be generated after running **all2lmp.py** if **write\_bond\_react** is True.

## Reproducibility, publication, and citing LUNAR

Typically, any MD-published paper leaves out details on how the MD models were parameterized and only discusses which force fields were used. If you are using LUNAR to generate work to be published you must cite LUNAR. Additionally, the writer of LUNAR would like to offer some suggestions for publishing work using LUNAR and the force field assignment processes, such that a culture in MD publishing can be fostered for reproducibility. As talked about in the **frc\_file**, section **all2lmp.py** goes through a very rigorous process to assign force field parameters and is likely the most rigorous process out there of all the currently available tools, including the commercially available tools [2-11]. Additionally, the assignment of the force field parameters relies on the atom types being set. If a user of LUNAR sets atom types using **atom\_typing.py**, they should just state in their paper that they used the atom types set by **atom\_typing.py**. However, if the users decide to adjust any of the atom types set by **atom\_typing.py** or decide to manually set atom types it is recommended to create a figure such as Figure 8 to add to their publication or put in the Supplemental Information (Figure 8 was produced using ChemDraw 2D).



Figure 8: Best practice for publishing which parameters were used in developed MD models

Providing all the used atom types in a publication as shown in Figure 8, will allow other users of LUNAR to generate the same models that others have published using the same FF parameters since the atom types enforce the parameter assignment. It will also force users of LUNAR to be careful about their atom type selection since they will be aware that it will be peer-reviewed. However, the developer of LUNAR does not want to create a peer-review environment that makes it impossible to get beyond the initial MD modeling building stage, just because a certain force field lacks parameters to build the model. So, reviewers please be lenient in criticizing atom-type selection, which has historically not been discussed in any of the current literature to allow for the fostering of a more open and reasonable publishing environment for creating reproducible results in the MD field.

# Code: bond\_react\_merge.py

## Purpose

The purpose of **bond\_react\_merge.py** is to automate as many aspects as possible for preparing input files for a *“fix bond/react”* (REACTER) LAMMPS simulation. The main aspects that **bond\_react\_merge.py** aims to automate are the “unification/merging” of the LAMMPS TypeIDs between any arbitrary number of LAMMPS datafiles, the automatic generation of the “Reaction Map file”, some level of support for dealing with the charging of reduced topologies, and the ability to script the reduction of template topologies. **bond\_react\_merge.py** also aims at being force field agnostic and should work with most classical fix-bond force fields. The limitation of **bond\_react\_merge.py** is that it can only “unify/merge” LAMMPS datafiles that have the **all2lmp.py** style of commenting or the **all2lmp.py** style for deriving type labels. See **LUNAR/bond\_react\_merge\_prep.py** or **LUNAR/all2lmp.py** for either converting your LAMMPS datafile to be compatible with **bond\_react\_merge.py** or generating a LAMMPS datafile that is compatible with **bond\_react\_merge.py**.

If your LAMMPS datafile is generated using another tool that is not **all2lmp.py** but has been converted to a format that is compatible with **bond\_react\_merge.py**, using **bond\_react\_merge\_prep.py**, please be aware of how the tool used to generate the original LAMMPS datafile handles that atomID ordering in the Bonds, Angles, Dihedrals, and Impropers section of the datafile. **bond\_react\_merge.py** and **bond\_react\_merge\_prep.py** assumes that the ordering of the atomIDs in the Bonds, Angles, Dihedrals, and Impropers sections are consistent with the ordering of the atom types for the parameters that define those interactions (i.e., if atomID1=c5, atomID2=op, atomID3=hc, and the angle parameter is for hc-op-c5, then the ordering of the angle atomIDs in the angles section should be 3 2 1 and not 1 2 3). The strict rules for the ordering of atomIDs are only important in some force fields and not others, such as: impropers using a dihedral type of function to enforce planarity as opposed to a true out-of-plane enforcement function or the use of crossterms in class2 FFs. Inconsistent ordering can cause incorrect results. **all2lmp.py** has been coded to always ensure proper atomID to atom-type parameter mapping to be consistent to avoid any issues (even if it is not required for certain FFs). **bond\_react\_merge\_prep.py** will only add comments and does not change any ordering of the atomIDs. This problem is even further magnified when trying to “unify/merge” all parameters across a large selection of LAMMPS datafiles. So it is also of best practice that all files read into **bond\_react\_merge.py**, have been generated with the same methods and tools (preferably with LUNAR even though there is a level of support to generate LAMMPS datafiles outside of LUNAR to use in **bond\_react\_merge.py**).

## Code Variables and Execution

**bond\_react\_merge.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **bond\_react\_merge.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 9.

A screenshot of a computer program

Description automatically generated

Figure 9: IDE view of some Python variables for bond\_react\_merge.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 bond\_react\_merge.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 10.

A screenshot of a computer program

Description automatically generated

Figure 10: Command line override manual printed by running “python3 bond\_react\_merge.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **bond\_react\_merge.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **bond\_react\_merge.py** file. Examples of command line override:

python3 bond\_react\_merge.py -files data1:mol1.data,data2:mol2.data,pre1:pre1.data,post1:post1.data

python3 bond\_react\_merge.py -files infile:merge\_files.txt -atomstyle full -class 2

python3 bond\_react\_merge.py -files infile:merge\_files.txt -class 2 -map T -edge 0

python3 bond\_react\_merge.py -gui

python3 bond\_react\_merge.py -man

python3 LUNAR.py

python3 LUNAR.py 125

It is worth noting that there is a **use\_GUI** Boolean flag in **bond\_react\_merge.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 bond\_react\_merge.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 bond\_react\_merge.py -files infile:merge\_files.txt -class 2”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 bond\_react\_merge.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **bond\_react\_merge.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **bond\_react\_merge.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **bond\_react\_merge.py** GUI are shown in Figure 11. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **bond\_react\_merge.py** file, except for the **files** variable which will remain as the default this code ships with or whatever the users has manually typed in **bond\_react\_merge.py**.

A screenshot of a computer

Description automatically generated

Figure 11: LUNAR/bond\_react\_merge.py GUI

The number of files in the “files stack” is controlled by the **maxfiles** variable (which takes an int value) found in **bond\_react\_merge.py** and controls the number of file “boxes” that **bond\_react\_merge.py** will load with. However, if **bond\_react\_merge.py** GUI is running and you attempt to “overload” the files the GUI will automatically add more boxes and insert the “overloaded” files, thus the **maxfiles** only sets the initial number of files the GUI will load with and then the GUI supports “overloading” of files if needed. Additionally, the “add file(s) to stack” button can be used to add a single file to the stack or to grab a group of files to add to the stack. When adding a file to the “stack” the “file-tag” WILL NOT be updated and the user must set the file-tag for each file accordingly. See the **files** variable discussion about file-tags. A shortcut naming convention for LAMMPS datafiles has been devised such that when loading a file using the GUI onto the “stack” the file-tag can be automatically assigned. The naming convention is summarized as:

* %%%%%tag=**FILETAG**.data; where %%%%% can be any text
* The **FILETAG** must be located between the “tag=” string and the “.data” or “.txt” ending
* The **FILETAG** can be any of the supported file tags listed in the **files** variable section
* Some examples:
  + detda\_typed\_IFF\_tag=**data1**.data (will load file with file-tag = **data1**)
  + pre\_reaction\_1\_typed\_IFF\_tag=**pre1**.data (will load file with file-tag = **pre1**)
  + post\_reaction\_1\_typed\_IFF\_tag=**post1**.data (will load file with file-tag = **post1**)
  + merge\_files\_tag=**infile**.txt (will load file with file-tag = **infile**)

The naming convention was initially devised so the developer of **bond\_react\_merge.py** could test the code quickly and effectively, however, users of **bond\_react\_merge.py** may find it a nice feature to use if they learn that their preferred method of running **bond\_react\_merge.py** ends up being via the GUI.

The **LUNAR.py** and the **bond\_react\_merge.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **bond\_react\_merge.py** can be set to different values, where **GUI\_zoom** in **bond\_react\_merge.py** only affects the GUI that is launched from **bond\_react\_merge.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **bond\_react\_merge.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **files** Override tags: **-files** or **-f** Usage: **REQUIRED**

* Is a Python dictionary with the keys set as “file-tags” and the values set as the files to read in with the corresponding “file-tag”. The file value can have the filename or the filename with a global path or the filename with a path relative to LUNAR’s location on your computer. The file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section. The following file tags are supported (an integer value follows the tag letters to make the key unique):
  + data1, data2, ... dataN for an arbitrary amount of starting molecules
  + pre1, pre2, ... preN for an arbitrary amount of pre-reactions
  + post1, post2, ... postN for an arbitrary amount of post reactions
  + infile for reading in input script that specifies file names
  + NOTE: if using **generate\_map\_file** the N-ID assigned between preN/postN MUST BE CONSISTENT!!!!
* The format of the “infile”will now be discussed. The “infile” supports comments where anything trailing the “#” character will be treated as a comment and ignored. The “file-tags” should follow the same rules as listed above. The file will be read by searching for a “file-tag” and then using whitespace delimiters to find the corresponding filename. Currently supported options to add in an "infile":
  + path="dir/layout/paths" (path to append to each filename specified – bound between “-characters)
  + parent\_directory="dir" (overrides the Python parent\_directory variable – bound between “”-characters)
  + dataN example1.data (the dataN "file-tag" followed by white space and the tagged datafile)
  + preN pre\_rxn1.data (the preN "file-tag" followed by white space and the tagged datafile)
  + postN pre\_rxn1.data (the postN "file-tag" followed by white space and the tagged datafile)
* An example “infile” format is provided below (note the “infile” can have any file extension ending, but “.txt” is likely the easiest to work with).

# anything following the “#” character will be ignored

# Specify a desired path to append to the front of each filename (optional however if not present

# and the files are a path from LUNAR on your computer each file below must have that path

# specified in front of the filename)

path = "EXAMPLES/bond\_react\_merge\_tests/test1"

# file-tag filename comment (required)

data1 detda\_typed\_IFF\_tag.data # This datafile will have all coeffs in it

data2 dgeba\_typed\_IFF\_tag.data # This datafile will have all coeffs in it

pre1 pre\_reaction\_1\_typed\_IFF.data # for rxn1

post1 post\_reaction\_1\_typed\_IFF.data # for rxn1

pre2 pre\_reaction\_2\_typed\_IFF.data # for rxn2

post2 post\_reaction\_2\_typed\_IFF.data # for rxn2

: :

# Specify the parent\_directory of where to write results (optional)

parent\_directory = "EXAMPLES/bond\_react\_merge\_tests/test1/run"

# anything following the “#” character will be ignored

# file-tag filename with path set

data1 EXAMPLES/bond\_react\_merge\_tests/test1/detda\_typed\_IFF\_tag

data2 EXAMPLES/bond\_react\_merge\_tests/test1/dgeba\_typed\_IFF\_tag.data

pre1 EXAMPLES/bond\_react\_merge\_tests/test1/pre\_reaction\_1\_typed\_IFF.data

post1 EXAMPLES/bond\_react\_merge\_tests/test1/post\_reaction\_1\_typed\_IFF.data

pre2 EXAMPLES/bond\_react\_merge\_tests/test1/pre\_reaction\_2\_typed\_IFF.data

post2 EXAMPLES/bond\_react\_merge\_tests/test1/post\_reaction\_2\_typed\_IFF.data

: :

# Specify the parent\_directory of where to write results to (optional)

parent\_directory = "EXAMPLES/bond\_react\_merge\_tests/test1/run"

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **bond\_react\_merge.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the first file in the **files** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **bond\_react\_merge.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

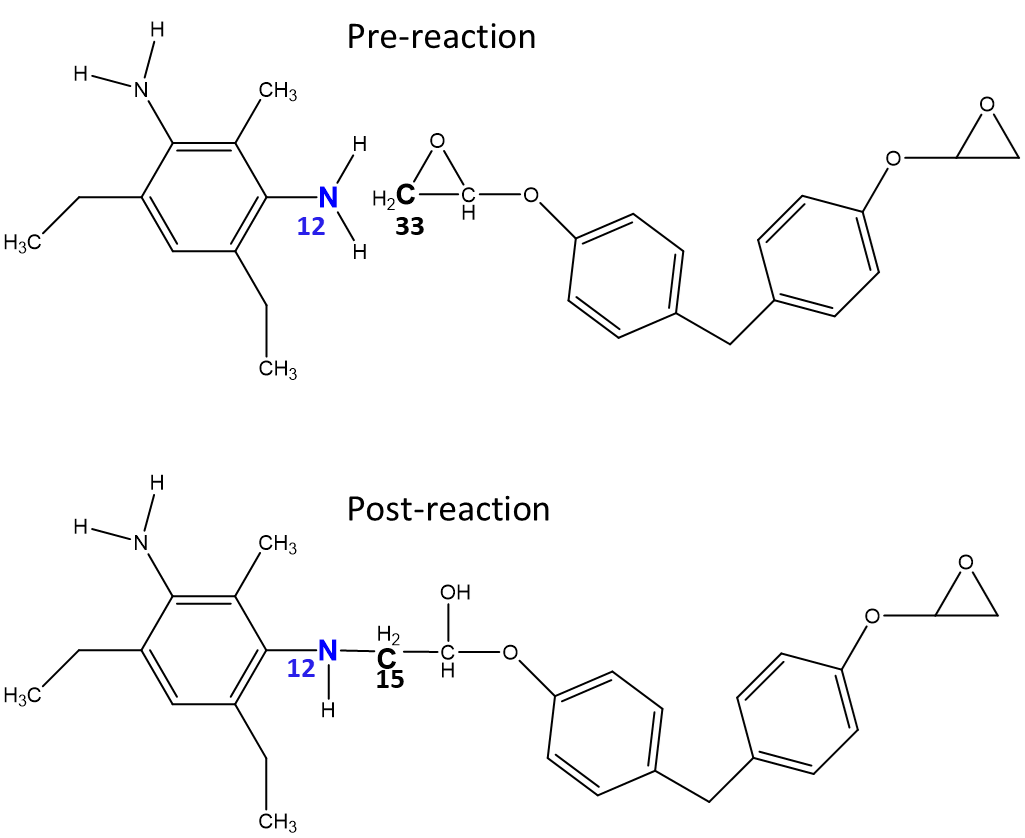
‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_merged’** and **topofile == ‘detda.data’):**
        + basename = ‘detda\_merged’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘merged-:’** and **topofile == ‘detda.data’):**
        + basename = ‘merged-detda’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as the file tag of each file and ‘ANYTEXT’ will be a suffix appended to the file tag. For example:
      * **(newfile == ‘-merged’**, **topofile == ‘detda.data’**, and tag **== ‘data1’):**
        + basename = ‘data1-merged’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.
    - **Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.

Variable: **generate\_map\_file** Override tags: **-map** or **-m** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to control if **bond\_react\_merge.py** will attempt to generate a map file relating the **preN** tagged file to the **postN** tagged file. The map between the **preN-tagged** file to the **postN-tagged** file is generated using a pseudo-graph-theory-based algorithm that relies on the cost of the map between atomIDs. The costs between the different atomIDs are based on a chemistry-informed difference of features and a relative depth (number of atoms away) from topological features to help inform the cost-setting algorithm. The algorithm developed is quite advanced and seems to work well. About 20 different reactions in varying topological complexity have been tested and about 95% of the time the algorithm gets everything on the map completely correct. However, since this is an extremely difficult task to completely automate it is up to the users of **generate\_map\_file**, to ensure the map from the **preN-tagged** file to the **postN-tagged** file is correct. See the **write\_rxn\_mol2files** and the **write\_rxn\_datafiles** Boolean variables to write reaction LAMMPS molecules files that can be viewed in software like OVITO, ChemDraw, Avogadro, VMD, Material Studio, etc to allow for easier checking of the map between the **preN** and **postN** tagged files.
* Additional information may be provided in the header of the **preN** tagged file to help **generate\_map\_file** find the correct map for the **N-ith** reaction. Other information may be provided in the **preN** tagged file to automatically reduce the topology of the reaction template or to allow **generate\_map\_file** to work with **CreateIDs**.
* **BondingIDs = [1, 2, 3, 4]** may be added to the header of any of the **preN** tagged file which is a string that looks like a Python variable and a Python list. The variable name **BondingIDs** specifies what this info is and must be present for usage of this option! This option is to specify "linking" atomIDs in the pre-reaction template and the post-reaction template to help the code set equivalences. NOTE that the usage of this option is NOT required, and the code will still attempt to find the correct mapping without this being specified in the **preN** tagged file. However, if **BondingIDs** are specified it will help ensure the code can find the mapping between the **preN** and **postN** tagged file by using them as starting points for a BFS (Breadth First-Search) traversal, mapping the pre-atomIDs to the post-atomIDs at each sequential depth stage emanating from the **BondingIDs**. Not all reaction templates can use this option since it assumes minimal topological changes from the pre to the post-reaction, where the "linking" pair can be seen in both the pre and post-reaction template. The **BondingIDs** option is equivalent to that of AutoMapper's --ba option (if you are familiar), where 4 atomIDs MUST BE specified and the order is important. The first two IDs are the IDs for the "linking" atomIDs in the pre-reaction and the second two IDs are for the "linking" atomIDs in the post-reaction template. The lst atomID and the 3rd atomID MUST BE equivalents between the pre and post-reaction template and the 2nd atomID and 4th atomID MUST BE equivalents between the pre and post-reaction template. To use this option simply put the string of **BondingIDs = [1, 2, 3, 4]**, in the header of the corresponding pre-reaction datafile before running **bond\_react\_merge.py**. NOTE the location in the header is not important if the string is in the format shown above. An example reaction template is shown for reacting a DETDA molecule with a DGEBF molecule to illustrate **BondingIDs**, where atomIDs **12** and **33** are the linking atoms in the pre-reaction template and atomIDs **12** and **15** are the same linking atomIDs in the post-reaction template.



HEADER, Pre-Rxn datafile; **BondingIDs = [12, 33, 12, 15];** Position in header doesn’t matter.

74 atoms

77 bonds

: :

9 atom types

15 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

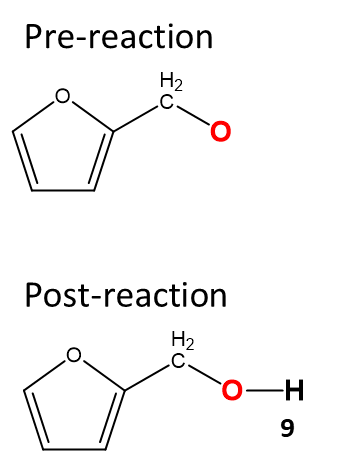
Masses

1 12.01115000 # c2

2 12.01115000 # c3

: : :

* **CreateIDs = [31, 32]** may be added to the header of any of the **preN** tagged file which is a string that looks like a Python variable and a Python list. The variable name **CreateIDs** specifies what this info is and must be present for usage of this option! This option is to specify the atomIDs that are atoms that *“fix bond/react”* can create. These atomIDs are the atomIDs found in the post-reaction template and DO NOT have a map to any pre-reaction atomIDs. If the reaction template has any atomIDs that are being created, THIS MUST APPEAR in the **preN** tagged file for the reaction IF you want to use the **generate\_map\_file** option to create the map file. If the reaction creates new atomIDs and they are not specified by this option, and you try to use the **generate\_map\_file** option the code will exit with an ERROR saying the pre-reaction template and post-reaction template DO NOT have the same number of atoms. To use this option simply put the string of **CreateIDs** **= [1, 2, 3],** in the header of the corresponding **preN** tagged datafile before reading the file into **bond\_react\_merge.py**. NOTE the location in the header is not important if the string is in the format shown above. An example reaction template is shown for a pseudo furfuryl alcohol molecule where an extra hydrogen atom (atomID **9**) is added to the oxygen atom. Where the post-reaction template now has an extra atom(s) and the atom(s) is a **CreateIDs**.



HEADER, Pre-Rxn datafile; **CreateIDs = [9];** Position in header doesn’t matter.

12 atoms

12 bonds

: :

5 atom types

12 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

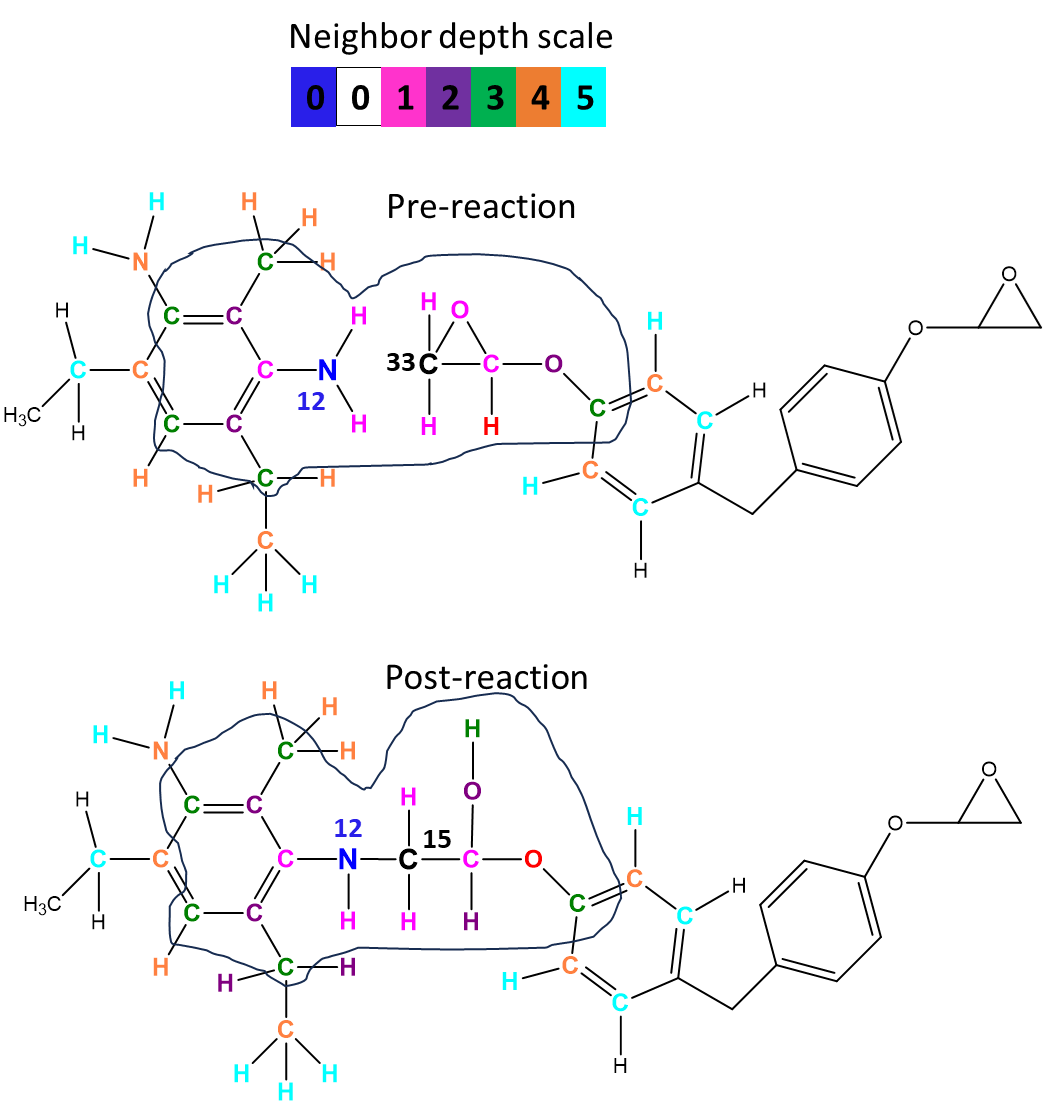
Masses

1 12.01115000 # c5

2 12.01115000 # c2

: : :

* **Reduce = [1, 2, 3, 4, 3]** may be added to the header of any of the **preN** tagged file which is a string that looks like a Python variable and a Python list. The variable name **Reduce** specifies what this info is and must be present for the usage of this option! If the length of the **Reduce** list is 5 elements long with the following meaning of each index: **[preID1, preID2, postID1, postID2, depth]**, where **preID1** and **postID1** must be equivalent atoms and **preID2** and **postID2** must be equivalent atoms and **depth** is the Breadth-First Traversal (DFS) depth of atoms to keep emanating outward from the **ID1** and **ID2** locations. Usage of this option will reduce the number of atoms in the **preN/postN** tagged files, where all atoms in the **depth** range from **ID1** and **ID2** are kept. The location in the header does not matter for the usage of this option. NOTE that **ID1** and **ID2** need not be "linking" atomIDs but should be near the "center" of the post-reaction template. Using **Reduce** with a list length of 5 imposes a **depth** restriction from **ID1** and **ID2**, to be identical. See the coupling of **Reduce** and **BondingIDs** to get around the **depth** restriction. An example reaction template is shown for reacting a DETDA molecule with a DGEBF molecule to illustrate **Reduce**, where atomIDs **12** and **33** are the linking atoms in the pre-reaction template and atomIDs **12** and **15** are the same linking atomIDs in the post-reaction template and a depth of **3** is used to reduce the template. The atoms circled will be the kept atoms and every other atom will be removed from the template.



HEADER, Pre-Rxn datafile; **Reduce = [12, 33, 12, 15, 3];** Position in header doesn’t matter.

74 atoms

77 bonds

: :

9 atom types

15 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

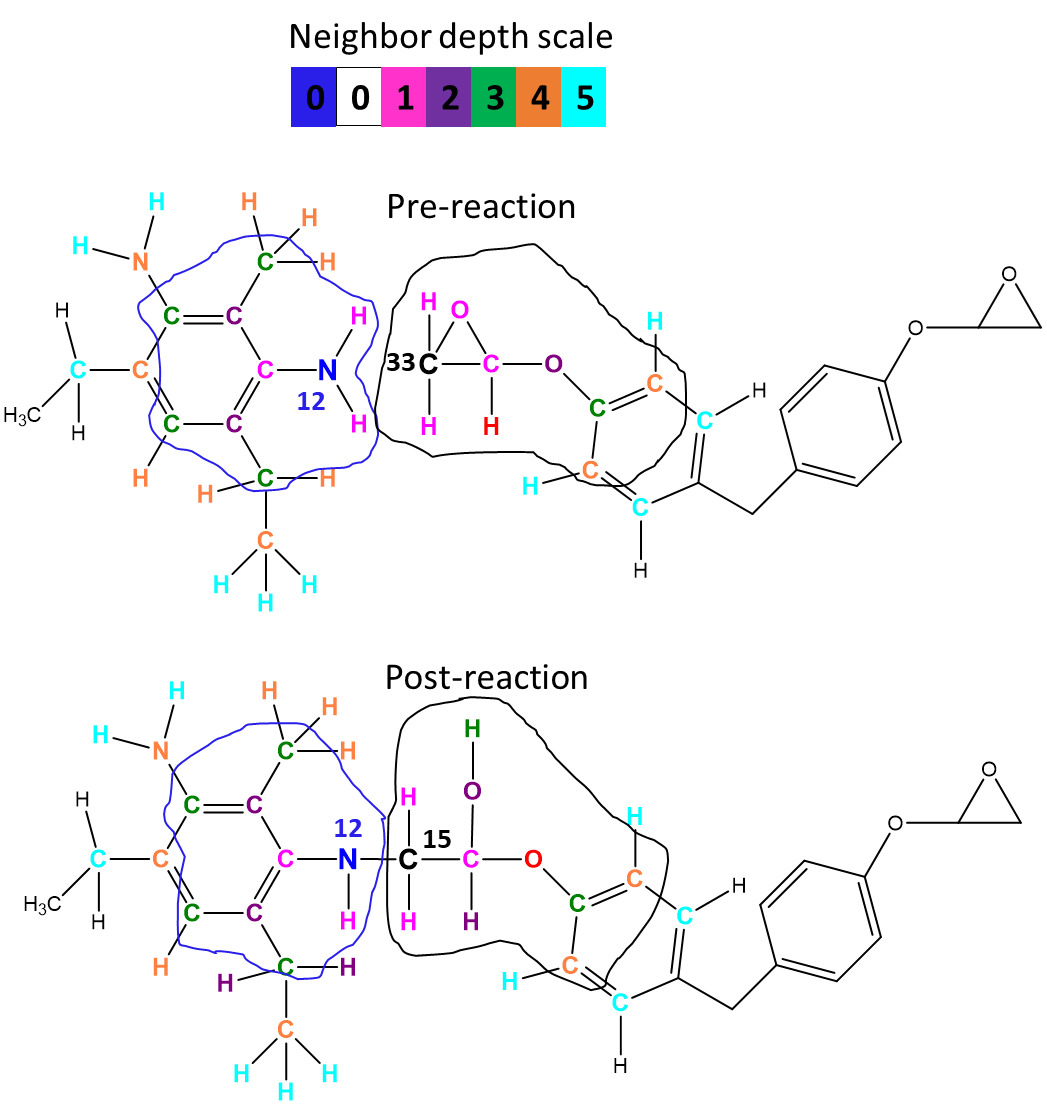
Masses

1 12.01115000 # c2

2 12.01115000 # c3

: : :

* **Reduce = [3, 5]; BondingIDs = [12, 33, 12, 15];** or  **BondingIDs = [12, 33, 12, 15]; Reduce = [3, 5], etc ...** maybe added to the header of any of the **preN** tagged files which is a string that looks like a Python variable and a Python list. The coupling of **Reduce** and **BondingIDs** specifies what this info is and must be present for the usage of this option! If the length of the **Reduce** list is 2 elements in this case and the length of **BondingIDs** is 4 elements. The **BondingIDs** are used as the points to emanate out from and the 2 elements of the **Reduce** list set the max depth to search from for atoms to keep. When coupling both **Reduce** and **BondingIDs** options the indexes have the following meanings: **BondingIDs = [preID1, preID2, postID1, postID2]; Reduce = [depth1, depth2];** where **depth1** sets the max depth searching from **preID1** and **postID1** and **depth2** sets the max depth searching from **preID2** and **postID2**. NOTE **ID1** and **ID2** MUST be linking IDs and assume the understanding discussed in the **BondingIDs** section also that **depth1** and **depth2** may only differ by 1 neighbor unit since in the post-reaction template the neighbor depths will only ever be offset by 1 neighbor unit (due to joining of molecules). An example reaction template is shown for reacting a DETDA molecule with a DGEBF molecule to illustrate **BondingIDs** and **Reduce** coupling, where atomIDs **12** and **33** are the linking atoms in the pre-reaction template and atomIDs **12** and **15** are the same linking atomIDs in the post-reaction template and a depth of **3** and **4** is used to reduce the template. The atoms circled will be the kept atoms and every other atom will be removed from the template.



HEADER, Pre-Rxn datafile; **BondingIDs = [12, 33, 12, 15];** **Reduce = [3, 4]**;

74 atoms

77 bonds

: :

9 atom types

15 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

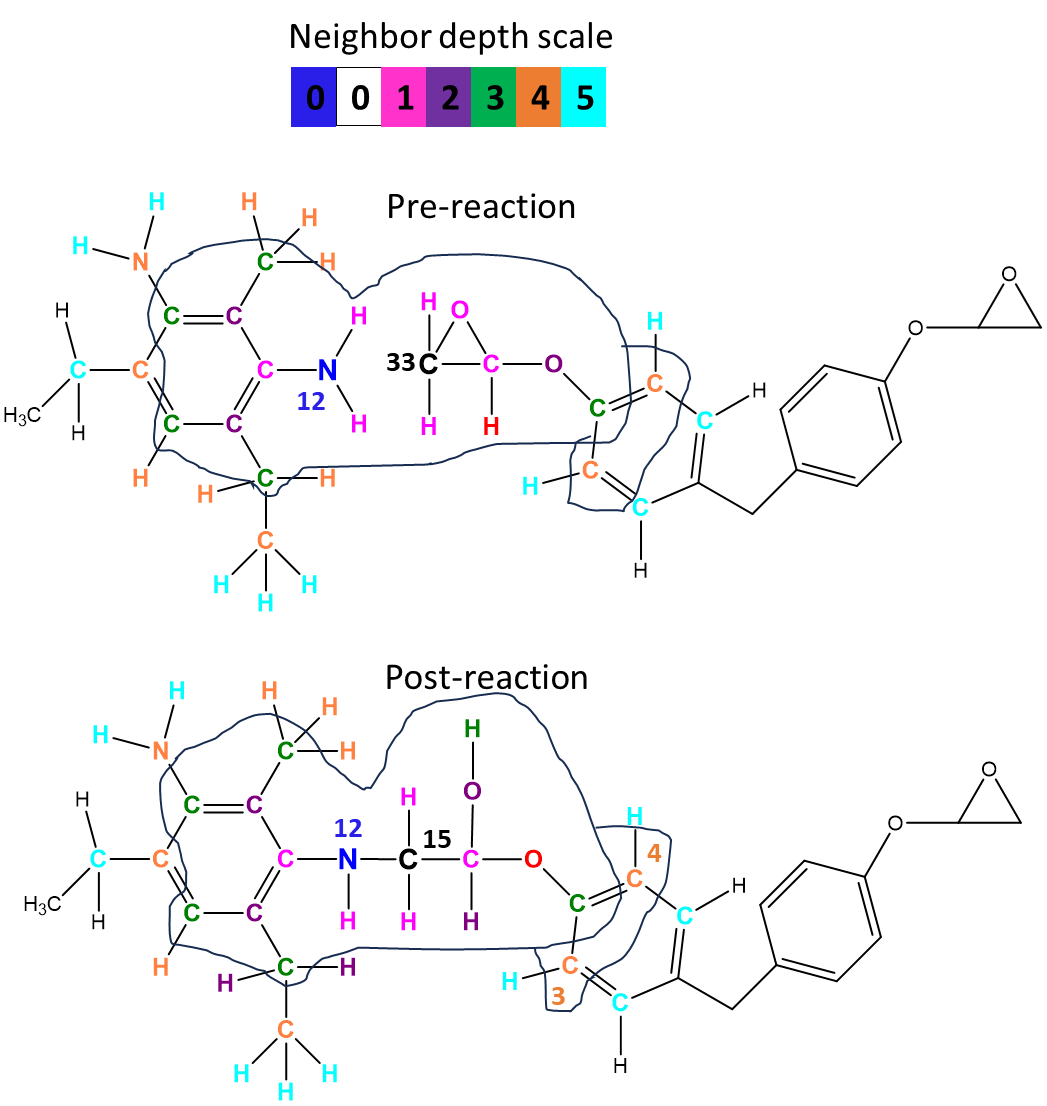
Masses

1 12.01115000 # c2

2 12.01115000 # c3

: : :

* **Keep = [1, 2, 3] or Remove = [4, 5, 6]** may be added to the header of any of the **preN** tagged file which is a string that looks like a Python variable and a Python list. The name **Keep** can be used to "keep" certain atoms when using the **Reduce** lists. To keep the atomIDs you desire, just add them to the **Keep** list. **Remove** can "remove" certain atomIDs, which can be used with the **Reduce** option or a standalone option if for some reason you accidentally built a reaction template with atoms where they should not belong. The addition of **Keep** and **Remove** options were added to allow **bond\_react\_merge.py** to reduce topologies in a more controlled manner and deal with the limitations that any **depth** from the two “linkingIDs”may only differ by 1-neighbor unit since in the post-reaction the molecules are joined together. An example reaction template is shown for reacting a DETDA molecule with a DGEBF molecule to illustrate the coupling of **Reduce** and **Keep**, where atomIDs **12** and **33** are the linking atoms in the pre-reaction template and atomIDs **12** and **15** are the same linking atomIDs in the post-reaction template and a depth of **3** is used to reduce the template. Additionally, a **Keep** list will be added to keep atomIDs **3** and **4**. The atoms circled will be the kept atoms and every other atom will be removed from the template. Notice the usage of the “**BondingIDs = [12, 33, 12, 15];** **Reduce = [3, 4]**;” and “**Reduce = [12, 33, 12, 15, 3];** **Keep = [3, 4];**”, can create the same reduced topologies so it is up to the user of these options to decide which method appeals most to them. Alternatively, the same thing can be accomplished using **Reduce/Remove** or **Reduce/Remove/BondingIDs**, lists where a large **depth** is supplied to the **Reduce** list and then using the **Remove** list to get rid of other atoms.



HEADER, Pre-Rxn datafile; **Reduce = [12, 33, 12, 15, 3];** **Keep = [3, 4];**

74 atoms

77 bonds

: :

9 atom types

15 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

Masses

1 12.01115000 # c2

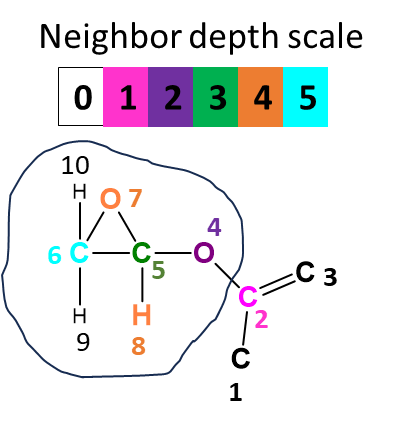
2 12.01115000 # c3

: : :

* In all the examples above notice the usage of the semi-colon character “;”, this is not a requirement, but if you write enough code you may already be used to using the “;” to create multiple “inline” variables. This is a preference that the writer of **bond\_react\_merge.py** likes but is not required. The code that parses the HEADER, of the LAMMPS datafile is robust enough to know when to stop reading each option based on the ending square bracket character “]”. Likewise, the parser knows when to start reading based on the beginning square bracket character “[”, thus whitespace issues should not exist.
* All LUNAR codes maintain HEADER info and will keep appending information to it as files are processed through the different LUNAR codes. This means, that by the time files are ready for usage in **bond\_react\_merge.py** the headers of the files may be very long, and it may be worth deleting certain text from the header to make room for the HEADER options listed above. Additionally, LAMMPS only allows for HEADERS of 256 characters in length so keep that in mind for managing very long headers. If your HEADER goes over 256 characters LAMMPS will likely exit with an “invalid file error”. All LUNAR codes enforce the 256-character limit, but manually typing in the headers may create headers larger than 256 characters.
* It is also worth noting that you DO NOT need to use **bond\_react\_merge.py**’s reduce option as shown above to reduce a template. You may also use other chemical drawing tools to build your initial reaction templates that are already reduced. The options listed above are to provide another method of building *“fix bond/react”*, templates and it is ultimately up to the user of the LUNAR codes to decide which methods best suit their needs and abilities. A benefit of using the template reduction methods above is that even experienced *“fix bond/react”* users often create incorrectly reduced templates, which generally means starting over by redrawing -> re-atom typing -> re-assignment the force field, where if you have a complete topology of your two reacting molecules you can just keep playing with the strings in the HEADERS and localize the re-work to only the **bond\_react\_merge.py** stage which should be a quicker method then going through the entire redrawing -> re-atom typing -> re-assignment the force field process. Additionally, the charge of edge atoms will already be properly dealt with and the atom types on the edge atoms or near edge atoms will already be consistent with the reacting monomers and the reaction templates after using **atom\_typing.py** since there is no missing topology and **atom\_typing.py** can assign correct atom types to all atoms.
* When using the **generate\_map\_file** option in  **bond\_react\_merge.py** it will also auto-generate a “in.fix\_bond\_react.script” with all the reaction templates and the *“fix bond/react”* command set using general settings. It will also auto-generate the force field settings based on the “style hint” in the read-in LAMMPS datafiles. This option can be useful to generate a LAMMPS input script quickly for many reactions or useful for beginners who are still learning the LAMMPS scripting language.

Variable: **molecule\_file\_options** Override tags: **-molecule** or **-mol** Usage: **REQUIRED**

* Is a Python list to put strings into, which can add sections to the written-in LAMMPS molecule file (the *“fix bond/react”* template where **bond\_react\_merge.py** will write with the **\*.lmpmol** ending). If the list is empty no additional sections will be written to the LAMMPS molecule file and the default sections that **bond\_react\_merge.py** writes will only appear. The purpose of the **molecule\_file\_options** is to add a level of automated support for adding additional sections to the LAMMPS molecule file, since **bond\_react\_merge.py**, will only write the minimal amount of sections needed to run a *“fix bond/react”* simulation. Adding more sections can be useful for changing the default operations of a *“fix bond/react”* simulation such as the usage of the *“custom\_charges”* keyword that can change how *“fix bond/react”* will superimpose charges. Additionally as mentioned previously in the **write\_bond\_react** section of the **all2lmp.py** chapter the naming of the codes has been based on the *“fix bond/react”* command since it is likely one of the most challenging commands to generate files for and the developer of LUNAR mainly uses *“fix bond/react”*, however other LAMMPS commands such as *“fix deposit”*, *“fix pour”*, *“fix rigid/small”*, *“fix shake”*, *“fix gcmc”*, *“create\_atoms”*, and *“atom\_style template”* also use the LAMMPS molecule file. The ability to add additional sections to the LAMMPS molecule file may be necessary for usage in the other LAMMPS commands that use the LAMMPS molecule file. To use any of the **molecule\_file\_options**, the **generate\_map\_file** Boolean MUST be True. The following are optional strings that may be added to the **molecule\_file\_options**:
  + ‘molecule’ which will find moleculeIDs via cluster analysis and order them from the largest-to-smallest cluster and then add a molecule section to the written **\*\_merge.lmpmol** LAMMPS molecule files.
  + ‘fragment/ID/OPTION’ (ordering matters and the delimiter is the ‘/’ character).
    - index0 = 'fragment' evokes an option to add a fragment section to the molecule file
    - index1 = 'ID' is the user-defined ID of the fragment
    - index2 = 'OPTION' is a user-defined option.
    - Currently available options:
    - ‘custom\_charges/depth\_from\_edge\_**N**’ where custom\_charges specifies the code to build options for *“fix bond/react” custom\_charges* option and depth\_from\_edge\_**N** tells the code to build a fragmentID that only has atoms at a certain depth **N** away from any edge atom. This is useful if the charging method used could not assign a proper charge to the edge atoms or near-edge atoms of the molecule template. This code has another option: **map\_near\_edge\_rxn\_charges** which works well to map charges from any **dataN** or **postN** tagged files onto edge atoms and a certain depth from edge atoms, but this is another workaround to the charging issue of atoms near an edge or edge atoms themselves. Much like **map\_near\_edge\_rxn\_charges** the meaning of depth for this option is as follows:
      * **0** = **preN** tagged files will have a molecule file with a fragmentID with all atoms specified EXCEPT atoms at depth **0** from any edge (the edge atoms ONLY).
      * **1** = **preN** tagged files will have a molecule file with a fragmentID with all atoms specified EXCEPT all atoms at depth **1** from any edge.
      * **N** = **preN** tagged files will have a molecule file with a fragmentID with all atoms specified EXCEPT all atoms at depth **N** from any edge.
      * Example Python string specifying this option to have a fragmentID in the **preN/postN** tagged molecule files that only have atoms in the fragmentID that are **1** depth from any edge atom:
        + ‘fragment/eqedge/custom\_charges/depth\_from\_edge\_1’
        + Will add a framentID called qedge in the written LAMMPS molecule file as illustrated below, where atomID **1** and **3** are edgeIDs and setting depth\_from\_edge\_1 as **1** will find all neighbors that are a depth of **1** away any edge and group them, then subtract that grouping of atoms from all other atoms. The remaining atomIDs will then be placed in the qedge fragmentID, which then can be passed as an argument into the *“fix bond/react” custom\_charges* option to only superimpose the charges that have correct charges.



HEADER, > bond\_react\_merge: v1.9 / 31 July 2023 molecule file

10 atoms

10 bonds

: :

Fragments

qedge 4 5 6 7 8 9 10 # fragment/qedge/custom\_charges/depth\_from\_edge\_1

Types

1 5 # cp

2 5 # cp

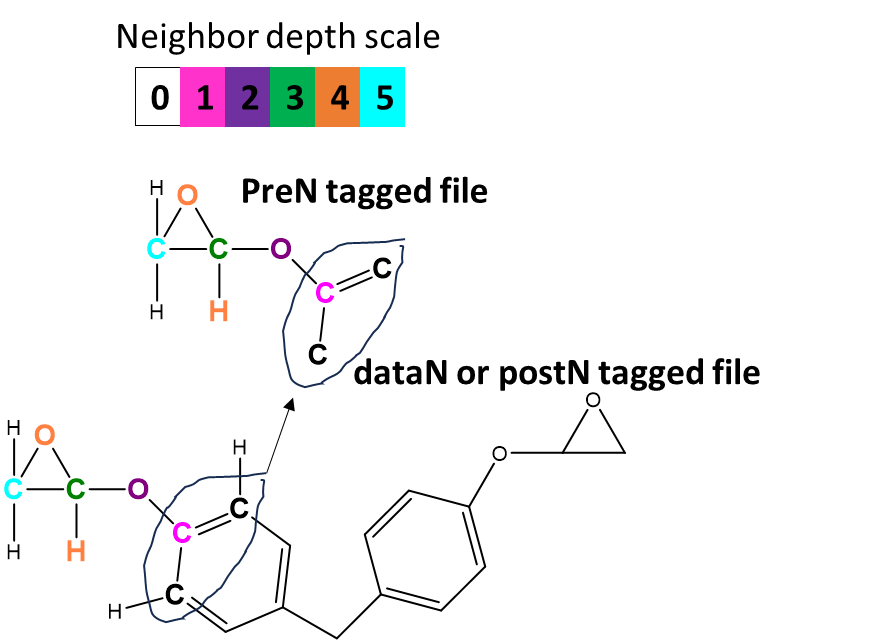
3 5 # cp

4 11 # oc

: : : :

Variable: **map\_near\_edge\_rxn\_charges** Override tags: **-edge** or **-e** Usage: **REQUIRED**

* Is a Python variable that will accept either an int value or a False Boolean. The purpose of **map\_near\_edge\_rxn\_charges** is to map charges of near edge atoms from **dataN** or **postN** tagged files onto a **preN** and **postN** tagged file. It is assumed that there is a **dataN** or **postN** file that has an exact topological match to the **preN** tagged files. The **dataN** or **postN** tagged file may contain many molecules per file. Once the **preN** tagged file is found to be a subset of the **dataN** or **postN** tagged file the charges will be mapped up to N-neighbors away and the **postN** tagged files will be updated via the equivalences found via the **generate\_map\_file** option. This means using the **map\_near\_edge\_rxn\_charges** requires **generate\_map\_file** option to be True. The sequential building of reaction templates such that post-reaction1 becomes pre-reaction2 is supported in this topological search, however, the int value assigned to the reaction “file-tags” must increase in value as the sequential building of templates occurs. The following are valid inputs and their meanings:
  + **False** = DO NOT attempt to match topologies for **preN** tagged files to **dataN** or **postN** tagged files and leave charges as is in **preN/postN** tagged files.
  + **0** = match **preN** tagged files to **dataN** or **postN** tagged file and update only edge atom charges. This depth should be used if the charge was set via the bond-increment method (int value sets depth from the edge and missing bond-increments only affect the edge atom itself).
  + **1** = match **preN** tagged files to **dataN** or **postN** tagged file and update edge atom charges + **1** neighbor-deeper (int value sets depth from the edge).
  + **2** = match **preN** tagged files to **dataN** or **postN** tagged file and update edge atom charges + **2** neighbor-deeper (int value sets depth from the edge).
  + **N** = match **preN** tagged files to **dataN** or **postN** tagged file and update edge atom charges + **N** neighbor-deeper (int value sets depth from the edge).
* An example is provided for **map\_near\_edge\_rxn\_charges = 1**, where a **preN** tagged file topology is matched with a **dataN** or **postN** tagged file, and then the charges from the **dataN** or **postN** tagged file will be super-imposed into the **preN** tagged file. Thus, allowing a charge mapping from a full topology to a reduced topology which can be used to update the edge atom charges. Also, note that the written LAMMPS molecule file will have comments provided on any atom charge that was manipulated using this method, so users have an easier time checking if the charge mapping was correct, which should be performed if using this method to update edge atom charges since it assumes that **generate\_map\_file** can generate a completely correct map file (which it typically can, but still needs to be checked).



HEADER, > bond\_react\_merge: v1.9 / 31 July 2023 molecule file

10 atoms

10 bonds

: :

Types

1 5 # cp

2 5 # cp

3 5 # cp

4 11 # oc

: : : :

Charges

1. -0.1268 # cp Charge mapping: 0 -> -0.1268 mapped from atomID (cp/C): 4 in **dataN** or **postN** file
2. 0.02650 # cp Charge mapping: 1 -> 0.02650 mapped from atomID (cp/C): 5 in **dataN** or **postN** file
3. -0.1268 # cp Charge mapping: 0 -> -0.1268 mapped from atomID (cp/C): 6 in **dataN** or **postN** file
4. -0.1595 # oc
5. 0.02700 # c2

: : : :

* Default should be to keep **map\_near\_edge\_rxn\_charges = False** if charges on edge atoms are already correct. Also note that if a **dataN** tagged file is large (>30,000 atoms), the topological matching process may be slow. This option has shown to be reliable in getting the exact topological match. However, if you are using this option it is recommended to check that the mapped charges are correct.

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file and thepossible LAMMPS **.lmpmol** molecule file. LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and does help with the manual creation of the files. **bond\_react\_merge.py** can add type labels into the written merged files even if the read-in files DO NOT have type labels defined since all of LUNAR has a consistent commenting scheme where **bond\_react\_merge.py** can derive the type labels due to this consistent commenting scheme. If type labels are used and a LAMMPS simulation is run LAMMPS will strip all comments from the simulated datafiles, but not the type labels. Since **bond\_react\_merge.py** requires **all2lmp.py** style of comments to merge/unify the coeffs it means a LAMMPS datafile that has been simulated is not compatible with **bond\_react\_merge.py** and comments would either be added in by processing the file with **all2lmp.py** using a “style type” format in the **nta\_file** or comments would have to be added with **bond\_react\_merge\_prep.py** using a “style type” format in the **cta\_file**. Alternatively, if type labels are used LAMMPS will maintain the type labels and a LAMMPS datafile that has been simulated can then be read directly into **bond\_react\_merge.py** again since on the backend the type labels will be used to “rebuild” the comments that LAMMPS removes from the simulated datafiles. One limitation to this is that the code that rebuilds the comments from type labels assumes the **all2lmp.py** style of type labels (i.e., if you want to read in a LAMMPS datafile with type labels only and no comments into **bond\_react\_merge.py**, the type labels must be in the LUNAR style of type labels).

Variable: **write\_rxn\_mol2files** Override tags: **-write-rxn-mol2files** or **-wrm** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write pre/post reaction **.mol2** files which ChemDraw/VMD/Avogadro, etc can natively read. This option is useful if you would like to visualize the pre/post-reaction templates in ChemDraw or VMD Avogadro etc. This option also works well with the **generate\_map\_file** option since it is a good idea to check if the map file finds all equivalences properly and ChemDraw allows multiple molecules to be opened at a time.

Variable: **write\_rxn\_datafiles** Override tags: **-write-rxn-datafiles** or **-wrd** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write pre/post reaction **.data** files with all coeffs inside. This option is mainly for troubleshooting purposes but can be used for validation that all coeff types and all topological types are mapped properly onto the newly typed coeffs. It allows for the visualization of the **\*.data** reactions in OVTIO, but could also create files that are compatible with AutoMapper (in case users want to try generating map files using AutoMapper or use AutoMapper's reduce template functionality).

Variable: **write\_moleculefiles** Override tags: **-write-moleculefiles** or **-wmf** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write out a molecule file (**\*.lmpmol)** file for any file that has a **dataN** tag. This file will have all the coeff types and all topological types are mapped properly and can be used with the LAMMPS *“create\_atoms”* command to build large random systems.
* This option will also generate a file called “force\_field.data”, which can be used with the LAMMPS *“read\_data”* command, to read in the force field to use create atoms. Additionally, an “in.create\_atoms.script” will be generated to show an example of how to read in the force field, the LAMMPS molecule files, and use the *“create\_atoms”* command.

Variable: **print\_options** Override tags: **-man** or **-opt** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to print the command line override manual and exit the code. True will print out the command line override manual and exit, whereas False will allow the code to run.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

The code will write minimally one output for every input file. The output filename will be the basename of the read-in file with the **\*\_merged.EXT** ending, where the file extension is based on the type of file to be written. A breakdown of files vs file tags is described below:

* **dataN** tagged file(s) will only have a single file output with the same basename but with the **\*.data** extension. These file(s) will have all the newly merged/unified force field coefficients in them and may be read into LAMMPS without the usage of any offsets. You may read many **dataN** tagged files into LAMMPS at a time after they come out of a **bond\_react\_merge.py** processing stage because each **dataN** tagged file has the same coeffs and the same LAMMPS CoeffIDs so reading in one file after another into LAMMPS will just keep overwriting the previous coefficients and CoeffIDs in the LAMMPS data structures. Alternatively, you can read the **dataN** tagged files into **LUNAR/cell\_builder.py** to generate a “well mixed” randomized starting condition for a LAMMPS simulation. **LUNAR/cell\_builder.py** has easy control over complex stoichiometric mixing and allows the user to control the level of randomness, please see the **cell\_builder.py** chapter for further details.
* **preN/postN** tagged file(s) will minimally have a file output with the same basename but with the **\*.lmpmol** extension. These files are the LAMMPS molecule files that are used as the templates for a *“fix bond/react”* simulation. These files will have the same LAMMPS CoeffIDs as the **dataN** tagged files or the same type labels as the **dataN** tagged files if **include\_type\_labels** is True. Additionally, if **generate\_map\_file** is True, two reaction map files will be generated with the names **preN-postN\_rxn-map\_commented.txt** and **preN-postN\_rxn-map\_uncommented.txt**, where **N** is the consistent int value between the **preN/postN** tagged files. The commented version provides comments about the mapping of atom types and how things were found, whereas the uncomment version is the version to use within a LAMMPS *“fix bond/react”* simulation, because currently *“fix bond/react”* cannot read reaction map files with comments in them. As mentioned previously, if using **generate\_map\_file** it is up to the user to ensure a proper mapping occurs since it is a complex task. The **generate\_map\_file** algorithm has proven to be robust, but given that getting a map file wrong can affect the rest of the simulations and possibly the property prediction it is best to check the map file. Since *“fix bond/react”* can only read a reaction map file that does not have comments it is advised to use the **preN-postN\_rxn-map\_uncommented.txt** to check for a correct map and update any information that may need updating. Lastly, if **write\_rxn\_mol2files** is True there will be written **\*\_merged.mol2** with the same basename as the **preN/postN** tagged file(s) and if **write\_rxn\_datafiles** is True there will be written **\*.data** with the same basename as the **preN/postN** tagged file(s).
* **in.fix\_bond\_react.script** will be generated if **generate\_map\_file** is True, where this output is a LAMMPS input script with general *“fix bond/react”* settings and will have all the **preN/postN** tagged files written along with the **preN-postN\_rxn-map\_uncommented.txt** for each reactionID. The force field section will be initialized from the style hints from the read-in files (if your read-in files do not have style hints the force field styles will be set as N/A. If you generated your initial datafiles with **all2lmp.py** your style hints in the files will be correct and consistent). The only thing that will for sure need modifying in this LAMMPS script is the LAMMPS datafile to be read-in (or multiples) via the *“read\_data”* command. This is typically dependent on the desired mixing ratio of molecules and **bond\_react\_merge.py** has no information about this. There are comments and suggestions on how to read your LAMMPS datafiles to the script or you may use **cell\_builder.py** to generate a single LAMMPS datafile to read into the script.
* **in.create\_atoms.script** and **force\_field.data** will be generated if **write\_moleculefiles** is True, which will show how to use the outputs of **bond\_react\_merge.py** to use the “create\_atoms” command.
* **.log.lunar** which contains the printouts of **bond\_react\_merge.py**.

# Code: auto\_morse\_bond\_update.py

## Purpose

The purpose of **auto\_morse\_bond\_update.py** is to automate the process of switching out harmonic bonds to Morse bonds as well as provide LAMMPS “include” type of scripts to be able to use the *“fix bond/break”* command quickly and easily. Throughout this chapter, the term “harmonic” bond will be used to describe either the “quadratic” harmonic bond or the “quartic” harmonic bond that exists in either a “class1” or “class2” force field, respectively. Note that LAMMPS syntax will use the term “harmonic” for “class1” force fields and the term “class2” for “class2” force fields. The harmonic bonds are converted to a Morse bond by finding the bond order (i.e., 1=single, 1.5=aromatic, 2=double, and 3=triple) of each bondTypeID in the LAMMPS datafile, through a graph theory-based approach with chemical intuition and assigning an experimental bond dissociation energy to that bondTypeID (the experimental values are found in *“src/auto\_morse\_bond/Morse\_parameters.txt”*). The alpha parameter is then fit by minimizing the sum of residuals of the harmonic bond and the Morse bond near the equilibrium position. Finally, a new LAMMPS datafile is written with the new Morse bond parameters in a hybrid style, where some bondTypeIDs are left as harmonic. This is due to how IFF-R has defined the process of updating the Morse bonds, such that only bond types that can carry any load, will be eligible for a Morse bond update and this code’s purpose was designed around that constraint.

## Code Variables and Execution

**auto\_morse\_bond\_update.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **auto\_morse\_bond\_update.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 12.

A screenshot of a computer program

Description automatically generated

Figure 12: IDE view of some Python variables for auto\_morse\_bond\_update.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 auto\_morse\_bond\_update.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 13.

A screenshot of a computer screen

Description automatically generated

Figure 13: Command line override manual printed by running “python3 auto\_morse\_bond\_update.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **auto\_morse\_bond\_update.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **auto\_morse\_bond\_update.py** file. Examples of command line override:

python3 auto\_morse\_bond\_update.py -topo example1.data -class 2 -atomstyle full

python3 auto\_morse\_bond\_update.py -topo example2.data -class 2 -type-labels T

python3 auto\_morse\_bond\_update.py -topo example3.data -xterms T

python3 auto\_morse\_bond\_update.py -gui

python3 auto\_morse\_bond\_update.py -man

python3 LUNAR.py

python3 LUNAR.py 125

It is worth noting that there is a **use\_GUI** Boolean flag in **auto\_morse\_bond\_update.py** to use the GUI or to not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 auto\_morse\_bond\_update.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 auto\_morse\_bond\_update.py -topo example. data -class 2”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 bond\_react\_merge.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **auto\_morse\_bond\_update.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **auto\_morse\_bond\_update.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **auto\_morse\_bond\_update.py** GUI are shown in Figure 14. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **auto\_morse\_bond\_update.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the users has manually typed in **auto\_morse\_bond\_update.py**.

A screenshot of a computer

Description automatically generated

Figure 14: LUNAR/auto\_morse\_bond\_update.py GUI

The **LUNAR.py** and the **auto\_morse\_bond\_update.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **auto\_morse\_bond\_update.py** can be set to different values, where **GUI\_zoom** in **auto\_morse\_bond\_update.py** only affects the GUI that is launched from **auto\_morse\_bond\_update.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **auto\_morse\_bond\_update.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file must be a LAMMPS **.data** fileand the bond coefficients must either be in “class2” or “harmonic” format. The **ff\_class** variable must also be consistent with the format of the bond coefficients. Additionally, the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **morsefile** Override tags: **-morse** or **-m** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the bond typing rules and the corresponding Morse bond dissociation energy coefficients. The **morsefile** currently has most of the bond typing rules set for all elemental/hybridization bonding configurations of the elements set in the **mass\_map** dictionary. The **morsefile** may be added onto or adjusted as desired to users liking, however, the file is meant to be as comprehensive as possible and should rarely need to be modified.

Variable: **zero\_effected\_xterms** Override tags: **-xterms** or **-zex** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to zero any affected crossterms in “class2” formatted LAMMPS datafiles and is only compatible with **ff\_class = 2**,if **ff\_class = 1** and **zero\_effected\_xterms = True,** this option will do nothing to the outputted LAMMPS datafile since “class1” FF’s do not have crossterms. The purpose for zeroing the affected crossterms is that some of the crossterms (bondbond, bondangle, middlebondtorsion, endbondtorsion, and bondbond13) use bond length to compute their interactions. However, once the bond length is no longer constrained by the harmonic bond, the affected crossterms get extremely large and cause the MD simulation to crash. This is because the initial derivation of the crossterms in the late 1990s was to be able to add anharmonic vibrational states to an FF, which uses harmonic bonds, but assumes that the harmonic bonds can constrain the rapidly changing crossterms potentials to only affect the vibrational range of motion while assuming the dissociation range of motion would not be reached. However, once the dissociation range of motion is available due to the addition of the Morse bonds, the crossterm's rapid changes can cause the simulation to crash. To be able to add Morse bonds to such “class2” MD models it was found that zeroing the crossterms is required to obtain stable simulations. One downside to zeroing the crossterms is that the system density is usually lowered by ~0.2 g/cm3, and it was found that an approximate 13% increase in the vdw sigma value for the hydrogens bonded to carbons (i.e., 2.3 adjusted to 2.0), can maintain the same system density as if the crossterms were not zeroed. It is recommended to try not zeroing the crossterms to see if you have a stable simulation and only zeroing them if needed.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_morse\_bond’** and **topofile == ‘detda.data’):**
        + basename = ‘detda\_morse\_bond’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘morse\_bond -:’** and **topofile == ‘detda.data’):**
        + basename = ‘morse\_bond-detda’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as ‘ANYTEXT’. For example:
      * **(newfile = ‘detda\_renamed’** and **topofile = ‘detda.data’):**
        + basename = ‘detda\_renamed’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.
    - **Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.

Variable: **radius\_specs** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as ‘start’, ‘end’, and ‘increment’ all take float or int type values. These values are used to generate the Morse bond vs harmonic bond plots shown below.

A graph of a graph

Description automatically generated

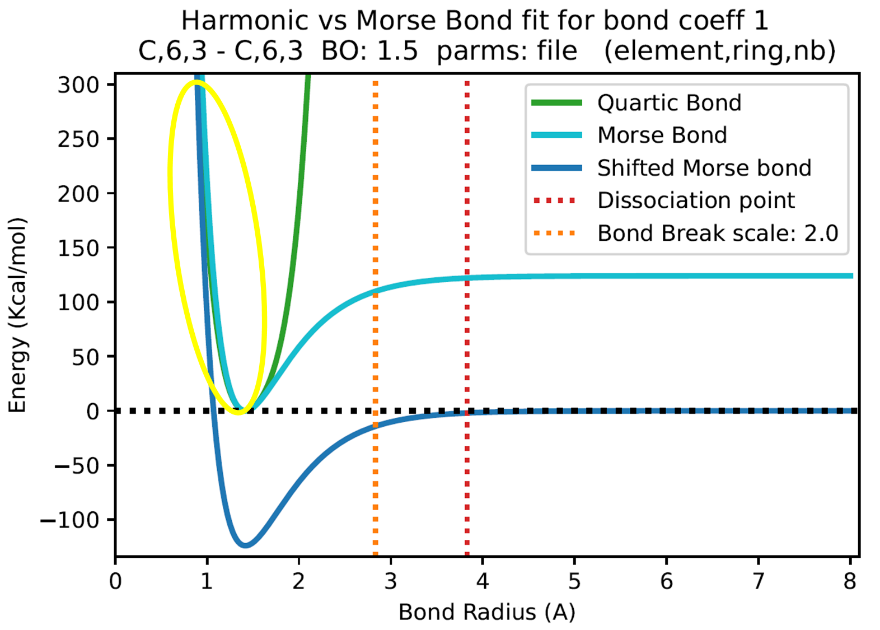
* The key values all do as they sound, where ‘start’ is where the start of the data begins (should almost always be 0.0), where ‘end’ is where the end of the data (should almost always be 8.0), and ‘increment’ is the data increment (should almost always be 0.01). This also sets the data used to fit the Morse bond alpha parameter “around” the harmonic bond, so if the increment is decreased the fitting code will take longer to run. Smaller increment values under 0.01 do not change the alpha parameter fit as compared to the 0.01 increment values so 0.01 is the accepted default value.

Variable: **alpha\_specs** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as ‘start’, ‘end’, and ‘increment’ all take float or int type values. These values are used to generate the possible alpha parameters to fit onto the harmonic Morse bond. The key values all do as they sound, where ‘start’ is where the start of the data begins (should almost always be 0.5), where ‘end’ is where the end of the data (should almost always be 3.5), and ‘increment’ is the data increment (should almost always be 0.1). These values set the constraints of the Morse bond alpha parameter for the fitting process.

Variable: **alpha\_scale** Override tags: **-alpha** or **-a** Usage: **REQUIRED**

* Is a Python float value to serve as a multiplier value to the fit alpha parameter. If **alpha\_scale** is 1.0, it means that the minimized alpha parameter is multiplied by 1.0, thus using the exact minimized alpha parameter.Using an **alpha\_scale** of 0.9 or 1.1 will decrease the alpha parameter by 10% or increase the alpha parameter by 10% respectively. The default **alpha\_scale** should be 1.0.
* The minimization of the alpha parameter is defined to be consistent with the IFF-R methodology of fitting the alpha parameter in the near equilibrium position, such that the compression end of the harmonic bond and Morse bond are the same as shown in the yellow circled region below.



* This fitting method inevitably also affects the tension side of the Morse bond stiffness as well. It may be desirable to adjust this fit due to various reasons such as increasing or decreasing the Morse bond stiffness to get the MD models to match experimental mechanical properties predictions or to change the vibrational state such that the compression side of the Morse bond does not create the same issues that the harmonic bond has for vibrations of atoms “colliding” with each other’s vdw radii if not using the LAMMPS *“special\_bonds lj/coul 0 0 1”* command. All scenarios described above are edge cases and **alpha\_scale** allows for those edge cases. It has been shown using PCFF-IFF-R (using the PCFF atom types), that accurate mechanical property prediction occurs when using **alpha\_scale = 1.0** and thus is the recommended default.
* A side note is that there is a closed-form solution to find the alpha parameter used for a quadratic harmonic bond, but not one for a quartic harmonic bond. This is why a minimization technique is being used in **auto\_morse\_bond\_update.py**.

Variable: **bondbreak\_scale** Override tags: **-bondbreak** or **-bb** Usage: **REQUIRED**

* Is a Python float value to serve as a multiplier value to the equilibrium **r0** value of each bondTypeID. The purpose of **bondbreak\_scale** is to adjust the **rcut** value in the LAMMPS include the file that **auto\_morse\_bond\_update.py** can write with **files2write[‘write\_bondbreak’] = True**, which auto-generates *“fix bond/break”* commands with **rcut = r0\*bondbreak\_scale**. The **r0\*bondbreak\_scale** is consistent with how IFF-R has been defined to be used with the *“fix bond/break”* command, where the default range of **bondbreak\_scale** is **1.75-2.0**.

Variable: **include\_rcut** Override tags: **-rcut** or **-irc** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include a **rcut** value in the Morse bond coefficients, which allows for the shifting of the Morse bond to create near zero energy discontinuity once a bond is broken. This Morse bond shifting has been shown to help stabilize systems when using *“fix bond/break”* using the *“fix temp/csvr”,* canonical sampling thermostat. The standard Morse bond parameters layout is described below and will result from setting **include\_rcut** as **False**, which is compatible with the standard LAMMPS Morse bond potential.

Bond Coeffs # morse

1 morse D r0 alpha

* If **include\_rcut** is **True**, the Morse bond parameters layout is described below and is only compatible if you compile LAMMPS with the USER-MORSE/active/bond\_morse.cpp and USER-MORSE/active/bond\_morse.h files.

Bond Coeffs # morse

1 morse D r0 alpha rcut

* Where the **rcut** value will shift the Morse bonding potential as shown in dark blue below and **rcut** is controlled by the **r0** and **bondbreak\_scale** via **rcut = r0\*bondbreak\_scale**.

A graph of a graph

Description automatically generated

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **auto\_morse\_bond\_update.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the **topofile** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **auto\_morse\_bond\_update.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ is that a relative directory can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **auto\_morse\_bond\_update.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix-bond force field the ‘full’ atom style is the best choice. Whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice. Also note that **auto\_morse\_bond\_update.py** will find all “clusters” in the read-in **topofile** and then reset the molIDs, which means if you are reading in a **topofile** in ‘charge’ format (does not have molIDs column), new molIDs will be found and the written LAMMPS datafile can then be written in the ‘full’ format with consistent molIDs reset by **auto\_morse\_bond\_update.py**.

Variable: **ff\_class** Override tags: **-class** or **-c** Usage: **REQUIRED**

* Is a Python int variable in which to set what “class” of force field. **ff\_class** informs **auto\_morse\_bond\_update.py** how to find the bond coefficients such that Morse bonds may fit around the harmonic bond. The following **ff\_class** options are available and what force fields typically use these class definitions:
  + 1 = class1 (int data type; bond coeff style = harmonic)
  + 2 = class2 (int data type; bond coeff style = class2)
* Most LAMMPS datafile generators will supply the LAMMPS “style hint” to the Coefficient sections of your LAMMPS datafile and LAMMPS will maintain the “style hints”, so in general if you do not know your force field class, you may look at your LAMMPS datafile to see the format of the “Bond Coeffs” section. Here are also some general FF names and their “Bond Coeff” styles.
  + 1 = class1 (int data type - FF files: cvff-IFF, cvff, clayff, opls-AA, DRIEDING)
  + 2 = class2 (int data type - FF files: PCFF-IFF, PCFF, compass)

Variable: **min\_bond\_length** Override tags: **-minr0** or **-mr0** Usage: **REQUIRED**

* Is a Python int or float variable to set which bondTypeIDs are eligible to be updated from harmonic to a Morse bond. The IFF-R force field has been defined such that a hybrid bond description is allowed such that only bonds that “carry” load may be updated from a harmonic bond to a Morse bond. This is a difficult and vague metric to decide which bonds are eligible to be updated from a topological point of view. The method that **auto\_morse\_bond\_update.py** deals with this is twofold: 1) the experimental bond dissociation energies and bond typing rules found in *“src/auto\_morse\_bond/Morse\_parameters.txt”* file are only for bond types that are known to be “load carrying” type of bonds (i.e., no C-H, O-H, … etc. type of bonds are defined in the file) and 2) the user controls the minimum equilibrium bond lengths that maybe eligible to be updated from a harmonic bond to a Morse bond via the **min\_bond\_length** variable. This allows the user to skip searching for shorter bond lengths such as bond lengths between any element and a hydrogen atom, where that type of bond leads to a “terminal” atom, which does not connect to the rest of the main topological structure. Additionally, the user may choose to skip converting bondTypeID(s) by setting them in the **coeffs2skip** variable.

Variable: **coeffs2skip** Override tags: **-skip** or **-c2s** Usage: **REQUIRED**

* Is a Python list to add any bondTypeIDs to skip converting from a harmonic bond to a Morse bond. As previously discussed in the **min\_bond\_length** variable on how **auto\_morse\_bond\_update.py** decides which bondTypeIDs may be eligible for converting from a harmonic bond to a Morse bond, there also exists another method to stop the conversion process by using the **coeffs2skip**. The Python list will take int values that correspond to the bondTypeIDs in the read-in **topofile**. This option requires users to have a very intimate knowledge of their system and the bondTypeIDs, which most do not. This option is a rare use case, but it exists for the possibility of needing it.

Variable: **mass\_map** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as elemental symbols and the corresponding values being a list of masses associated with that element type. The purpose of the **mass\_map** dictionary is to set the element type of each atom type in the **.data** file, which will be identified via the mass of each atom type. The mass of each atom type in a **.data** is usually dependent on the force field used to generate the initial **.data** file. To easily extend the code to a variety of different **.data** files that are initialized from different force fields, you just need to add the force field’s mass to the existing list of masses. The code will exit if all element symbols cannot be found via this mapping process, in which case you will just add the necessary info to the **mass\_map** dictionary.

Variable: **files2write** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as representative names of the different files that **auto\_morse\_bond\_update.py** can write and the values are Booleans (True or False), to control whether the code will write that file or not. The following are the supported keys and what type of file it controls:
  + ‘write\_log\_file’ which is a **\*.txt** file with the name basename\_**newfile**.txt, where “basename” is the “basename” of the **topofile** and the **newfile** is the string provided via the **newfile** variable. This file contains all the printouts seen when running **auto\_morse\_bond\_update.py**.
  + ‘write\_datafile’ which is a **\*.data** file with the name basename\_**newfile**.data, where “basename” is the “basename” of the **topofile** and the **newfile** is the string provided via the **newfile** variable. This file is the LAMMPS datafile with the new combinations of the harmonic bonds and the Morse bonds (and possibly zeroed crossterms via the **zero\_effected\_xterms** variable). In general, it will always be desired to write this file, however since this code can auto-generate LAMMPS “include” type of files for *“fix bond/break”* it may be desired to only use **auto\_morse\_bond\_update.py** for that purpose, which is controlled by the **files2write** ‘write\_bondbreak’ key.
  + ‘write\_pdffile’ which is a **\*.pdf** file with the name basename\_**newfile**.pdf, where “basename” is the “basename” of the **topofile** and the **newfile** is the string provided via the **newfile** variable. This file contains all the plots showing the Morse bond fit onto the harmonic bond to display the “quality” of fit that **auto\_morse\_bond\_update.py** has generated based on the minimization of the sum of residuals.
  + ‘write\_bondbreak’ which is a **\*.script** file with the name basename\_**newfile**.script, where “basename” is the “basename” of the **topofile** and the **newfile** is the string provided via the **newfile** variable. This file contains readily usable *“fix bond/break”* commands for LAMMPS and should be imported into the master LAMMPS input file by using “include basename\_newfile.script”. Where the **rcut** value is controlled by the **r0** value of the harmonic bond and the **bondbreak\_scale** variable of **auto\_morse\_bond\_update.py**. The LAMMPS *“fix bond/break”* command requires whichever bondTypeIDs that are desired to be simulated as a breakable bond, must be set in a separate *“fix”*, which can make manually building such fixes very time-consuming for systems with many bondTypeIDs. The generated **\*.script** is set up to be an “include” script where you can use the LAMMPS “include filename” command to link your master LAMMPS script to this one. Alternatively, you can copy and paste from the file to your master LAMMPS script if you prefer that method of building your LAMMPS scripts. The file generated by **auto\_morse\_bond\_update.py** also contains some useful comments that can be quickly copied and pasted into the master LAMMPS script. For instance, the *“fix bond/break” “fix IDs”* can be copiedinto a LAMMPS *“thermo\_style custom”* of your master LAMMPS script such that the number of bonds that break during the simulation is written to the LAMMPS logfile.

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file. LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and does help with the manual creation of the files. **auto\_morse\_bond\_update.py** can add type labels into the written files even if the read-in files DO NOT have type labels defined but have the **all2lmp.py** style of comments since all of LUNAR has a consistent commenting scheme where **auto\_morse\_bond\_update.py** can derive the type labels due to this consistent commenting scheme.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

* Please refer to the **files2write** section to see the type of outputs **auto\_morse\_bond\_update.py** generates and how to control if they are written or not and a **.log.lunar** which contains the printouts of **auto\_morse\_bond\_update.py**.

# Code: cell\_builder.py

## Purpose

The purpose of **cell\_builder.py** is to “mix” molecules by randomly placing them and randomly rotating them on a lattice grid set via a cubic lattice. The lattice locations are set via the **distance\_scale** variable which is a multiplier value to the longest molecule supplied to the code and if greater than one it means no atom positions will overlap. **cell\_builder.py** only reads LAMMPS datafiles and can apply offsets to the different TypeIDs or merge different TypeIDs of different LAMMPS datafiles to join them together. The **cell\_builder.py** is like Packmol, in the fact that it can prepare large systems for MD butdiffers from Packmol since it only accepts LAMMPS datafiles, meaning users can set atom types on a model with a small number of atoms and then use **cell\_builder.py** to build massively large systems with minimal effort for dealing with atom typing/charging or managing the LAMMPS coefficients for different force fields. NOTE that **atom\_typing.py** and **all2lmp.py** can read **.pdb** files which can read the outputs of a Packmol model if desired. **cell\_builder.py** has easy stoichiometric control over the mixing ratio of molecules. The density of a **cell\_builder.py** model is close to gas due to how it deals with the issue of overlapping atoms. However, **cell\_builder.py** will automatically generate a LAMMPS input script to shrink the simulation cell dimensions uniformly in an MD simulation producing a perfect cube where the final density is controlled by a LAMMPS variable. This will allow the “densification” from a gaseous state to a liquid state, where the packing of the molecules is controlled by the force fields' non-bonded interactions. The primary use case of **cell\_builder.py** is for the preparation of a LAMMPS *“fix bond/react”* simulation, however **cell\_builder.py** maybe used for other scenarios as well. The auto generated LAMMPS script to shrink the simulation cell dimensions will keep the simulation box centered about 0, 0, 0.

## Code Variables and Execution

**cell\_builder.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **cell\_builder.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 15.

A screenshot of a computer program

Description automatically generated

Figure 15: IDE view of some Python variables for cell\_builder.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 cell\_builder.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 16.

A screenshot of a computer screen

Description automatically generated

Figure 16: Command line override manual printed by running “python3 cell\_builder.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **cell\_builder.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **cell\_builder.py** file. Examples of command line override:

python3 cell\_builder.py -files mol1.data:1,mol2.data:2 -atomstyle full -duplicate 50

python3 cell\_builder.py -files mol1.data:1,mol2.data:2 -type-labels T

python3 cell\_builder.py -rx 90 -ry 180 -rz 270 -reset-molids T

python3 cell\_builder.py -gui

python3 cell\_builder.py -man

python3 LUNAR.py

python3 LUNAR.py 75

It is worth noting that there is a **use\_GUI** Boolean flag in **cell\_builder.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 cell\_builder.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 cell\_builder.py -rx 90 -ry 90 -rz 90 -rm T”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 cell\_builder.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **cell\_builder.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **cell\_builder.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **cell\_builder.py** GUI are shown in Figure 17. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **cell\_builder.py** file, except for the **files** variable which will remain as the default this code ships with or whatever the users have manually typed in **cell\_builder.py**.

A screenshot of a computer

Description automatically generated

Figure 17: LUNAR/cell\_builder.py GUI

The number of files in the “files stack” is controlled by the **maxfiles** variable (which takes an int value) found in **cell\_builder.py** and controls the number of file “boxes” that **cell\_builder.py** will load with. However, if **cell\_builder.py** GUI is running and you attempt to “overload” the files the GUI will automatically add more boxes and insert the “overloaded” files, thus the **maxfiles** only sets the initial number of files the GUI will load with and then the GUI supports “overloading” of files if needed. Additionally, the “add file(s) to stack” button can be used to add a single file to the stack or to grab a group of files to add to the stack. When adding a file to the “stack” the “qty” will be initialized as 1 and the user must update the “qty” for each file accordingly. See the **files** variable discussion about “qty”. A shortcut naming convention for LAMMPS datafiles has been devised such that when loading a file using the GUI onto the “stack” the “qty” can be automatically assigned. The naming convention is summarized as:

* %%%%%qty=**NMOLECULES**.data; where %%%%% can be any text
* The **NMOLECULES** must be located between the “qty=” string and the “.data” and must be an integer value.

Some examples:

* + detda\_typed\_IFF\_qty=**1**.data (will load file with qty = **1**)
  + dgeba\_typed\_IFF\_qty=**2**.data (will load file with qty = **2**)

The naming convention was initially devised so the developer of **cell\_builder.py** could test the code quickly and effectively, however, users of **cell\_builder.py** may find it a nice feature to use if they learn that their preferred method of running **cell\_builder.py** ends up being via the GUI.

The **LUNAR.py** and the **cell\_builder.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **cell\_builder.py** can be set to different values, where **GUI\_zoom** in **cell\_builder.py** only affects the GUI that is launched from **cell\_builder.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **cell\_builder.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **files** Override tags: **-files** or **-f** Usage: **REQUIRED**

* Is a Python dictionary with the keys set as “files” and the values set as the “qty” of that file. The file key can have the filename or the filename with a global path or the filename with a path relative to LUNAR’s location on your computer. The “qty” sets the number of those files to read in and allows for easy stoichiometric control to create relative numbers of different monomer types. It is advised to pick the smallest “qty” as possible and then use the **duplicate** variable to adjust the size of the outputted file. The files can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

The “qty” maybe set to ZERO to read in a LAMMPS datafile as a system. Many LAMMPS datafiles can be read in this way, where each time a LAMMPS datafile is read in the simulation cell will keep adjusting to accommodate the new simulation cell. During this process the simulation cell will either grow or stay the same. No checks are performed if grouping a simulation cell where a particular molecule is periodically bonded, thus careful consideration must be given to how the simulation cell is setup in each file that is read in with the “qty” set to ZERO. This method will allow a “system” to be defined and then molecules can be added to the system using the lattice points that **cell\_builder.py** derives. NOTE that during the creation of lattice points, each lattice point will be checked to see if it will create overlapping atoms with the system defined via the read in files with the qty set to ZERO. Additionally, users may find that the simulation cell of the read in files with the “qty” set to ZERO may need to be modified to allow cell\_builder.py to insert the number of molecules users supply to the code. Additionally, these files can have the entire molecular system shifted by placing a “Shifts” dictionary in the header of the file before reading the file into **cell\_builder.py**. Please see the“**Optional Python dictionaries to place in the Header of read-in LAMMPS datafiles**”for further information.

* The “qty” maybe set to ZERO to read in a LAMMPS type script to set groups of atoms, which will be maintained as each molecule gets duplicated. The LAMMPS type script that **cell\_builder.py** supports is very similar to using the LAMMPS *“group”* command. Before a group of atoms is defined a filename must be provided before the group of atoms using the *“read\_data FILENAME.data”* format, where every group “under” the *“read\_data”* string will apply the grouping of atoms to that file. Where you may then use another *“read\_data”* to set up the groups for another file. The *cell\_builder.py* group command is summarized as:
  + group ID style args
    - “group” defines the command
    - “ID” is a user-defined name of the group
    - “style” defines the style. Currently supported styles are "id" and "type"
    - "args" list of one or more atomIDs via style "id" or atomTypes via style "type". Any entry in list can be a sequence formatted as A:B or A:B:C where
      * A = starting index, B = ending index
      * C = increment between indices, 1 if not specified
  + This file supports comments via the "#" where anything trailing the "#" character will be treated as a comment and ignored. This file also supports uses of the "&" as the next line continuation character.
  + Note that if any atoms are not assigned to a specific group, cell\_builder.py will automatically generate an “unassigned” groupID and put all unassigned atoms in that group.
  + An example script is provided below (additionally find a full example script at LUNAR/EXAMPLES/cell\_builder/ system1\_detda\_dgebf\_grouping\_example.script).

# This is a comment

# define a group of atoms for file1

read\_data file1.data

group aromatic id 1:6 10:16 20 21 22 23 24 25 &

30 31 32 40 41

group aliphatic type 1:3

# define a group of atoms for file2

read\_data file2.data

group amine type 1 5

Variable: **force\_field\_joining** Override tags: **-ff-join** or **-ffj** Usage: **REQUIRED**

* This variable defines how to handle the force field between multiple LAMMPS datafiles, where the following options exist:
  + ‘none’ which assumes all LAMMPS coefficient types are the same in all the read in files and applies no offset to the files as they are being used to generate a large molecular system. This option should be used if your files where processed with **LUNAR/bond\_react\_merge.py** to ensure that the force field between the output system is consistent with the reaction templates.
  + ‘merge’ which applies the merging processes present **LUNAR/bond\_react\_merge.py** to merge all coefficient types amongst all read in files. This option requires that all LAMMPS datafiles have the **LUNAR/all2lmp.py** style of comments.
  + ‘offset’ which applies an offset to each coefficient type in each file as it is read into **cell\_builder.py**.

Variable: **duplicate** Override tags: **-duplicate** or **-dup** Usage: **REQUIRED**

* Is a Python int value to set the number of times to duplicate the **files** dictionary. The stoichiometric mix ratio will be controlled by the “qty” value in the **files** dictionary. The **duplicate** variable can be thought of as the LAMMPS *“replicate”* command except **the duplicate** will randomly place molecules starting from a 0,0,0 position in a pseudospherical manner and deals with the issues of overlapped atoms via the **distance\_scale** variable. The minimum integer value you may give to **duplicate** is 1 with no maximum integer value. The code is heavily optimized and should run quickly for most MD model sizes (a 500,000 atom system took about 15 seconds to run on Python v3.11.4 on Linux Ubuntu).

Variable: **distance\_scale** Override tags: **-dist-scale** or **-ds** Usage: **REQUIRED**

* Is a Python float or int value to set the distance multiplier value of the “longest” molecule of the system defined via the read-in from the **files** dictionary and controls the spacing of the lattice points where each molecule will randomly be placed and randomly rotated about. A **distance\_scale** of 1 ensures no atom positions will be overlapped during the random rotations of each molecule however it cannot guarantee that not all vdw radii or the non-bonded molecules will overlap. Therefore, a safe default **distance\_scale** value of 1.2 is recommended (20% more space than the longest molecule).

Variable: **seed** Override tags: **-seed** or **-s** Usage: **REQUIRED**

* Is a Python positive integer value or ZERO to set a seed for generating random numbers to account for reproducibility concerns of generating random initial positions. If the seed value is zero the seed will default to using the current system time.
* Please note that specifying the **max\_rotations**, **distance\_scale**, and **group\_monomers\_locally** must also be specified so others can use the same settings to be able to reproduce an exact copy of your work.

Variable: **group\_monomers\_locally** Override tags: **-grp-mono** or **-grp** Usage: **REQUIRED**

* Is a Python Boolean (True or False) to group the monomers in the **files** dictionary locally and then randomly place and rotate the local grouping of monomers. This is useful if you want to keep the proximity of a hardener and resin molecules to allow for quicker crosslinking simulations or if you would like to add in the artifact that every hardener and resin start in a relatively close position to help achieve a 100% conversion.
* An image is provided below to demonstrate the difference for an EPON 862/DETDA system where (a) **group\_monomers\_locally** = False and (b) **group\_monomers\_locally** = True.

A diagram of a cell

Description automatically generated

Variable: **domain** Override tags: **-domain** or **-dn** Usage: **REQUIRED**

* Is a Python string variable in which to set the lattice domain. The lattice points will always be generated about the 0, 0, 0 position in x, y, and z. However, the number of lattice points in the x, y, and z directions can be set by the user or set to be cubic. The following options are available:
  + ‘cubic’ which automatically determines the number of lattice points required based on the **qty** of files and the **duplicate** variable.
  + ‘Ni x Nj x Nk’ where ‘Ni’ is the number of lattice points in the x-direction, ‘Nj’ is the number of lattice points in the y-direction, and ‘Nk’ is the number of lattice points in the z-direction. Please note the following about ‘Ni x Nj x Nk’:
    - If **group\_monomers\_locally** is False, ‘Ni x Nj x Nk’ must be greater than **duplicate**\*sum(**qty** of all files), to allow for enough lattice points. If there is not enough lattice points, the code will exit with an ERROR.
    - If **group\_monomers\_locally** is True, ‘Ni x Nj x Nk’ must be greater than **duplicate** value, to allow for enough lattice points. If there is not enough lattice points, the code will exit with an ERROR. Additionally, by default during the initial grouping of monomers a ‘cubic’ lattice will be used.

Variable: **max\_rotations** Override tags: **-rx** or **-ry** or **-rz** or **-rall** Usage: **REQUIRED**

* Is a Python dictionary with the keys set as ‘x’, ‘y’, and ‘z’ which controls the maximum random rotation (in degrees) applied to each molecule about the x-axis, y-axis, and z-axis respectively. The values of the dictionary are int or float values to set the maximum rotation. A good default value for all maximum rotations is 360 degrees but may be adjusted if the need arises. Please note that the **distance\_scale** variable can control how “close” the molecules are placed so the usage of **max\_rotations** and **distance\_scale** go hand in hand.

Variable: **unwrap\_atoms\_via\_image\_flags** Override tags: **-unwrap** or **-u** Usage: **REQUIRED**

* Is a Python Boolean (True or False) to unwrap atoms via image flags during the initial reading-in of the **files** dictionary. If your system has all image flags as zeros the atom positions will remain the same, however, if they are none zero the atoms will be displaced by the image flag multiplied by the simulation cell dimension. The default value to **unwrap\_atoms\_via\_image\_flags** should be True. There are cases however that **cell\_builder.py** may be used to randomize ReaxFF simulation “cubes” randomly and allow them to react to one another based on the ReaxFF parameterization to build complex randomized nano-micro structures, in which **unwrap\_atoms\_via\_image\_flags**  can be set as False. Note that once the atom positions have been unwrapped, no “rewrapping” of the molecules will occur such as what happens with the LAMMPS *“replicate”* command, this is because the simulation cell dimension that **cell\_builder.py** sets ensures no molecules are periodically bonded with the end goal of **cell\_builder.py** being to initialize systems for a LAMMPS *“fix bond/react”* simulation where the starting monomers are relatively small. This also restricts the usage of **cell\_builder.py** where it cannot effectively deal with periodically bonded large molecules.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output file(s) “basename”. Where unlike most of the LUNAR codes, the “basename” set by **newfile**, **cell\_builder.py** is the actual filename output, whereas, in the rest of the LUNAR codes, the **newfile** string would get append to the **topofile** “basename”.

Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **cell\_builder.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix bond force field the ‘full’ atom style is the best choice. whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice. Also note that **cell\_builder.py** will find all “clusters” in the new systemand then reset the molIDs (if **reset\_molids** is True), which means if using the atom style as ‘full’ format molIDs will be consistent.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **cell\_builder.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the first file in the **files** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **cell\_builder.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file. LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and does help with the manual creation of the files. **cell\_builder.py** can add type labels into the written merged files even if the read-in files DO NOT have type labels defined since all of LUNAR has a consistent commenting scheme where **cell\_builder.py** can derive the type labels due to this consistent commenting scheme. If type labels are used and a LAMMPS simulation is run LAMMPS will strip all comments from the simulated datafiles, but not the type labels. Since **bond\_react\_merge.py** requires **all2lmp.py** style of comments to merge/unify the coeffs it means a LAMMPS datafile that has been simulated is not compatible with **bond\_react\_merge.py** and comments would either be added in by processing the file with **all2lmp.py** using a “style type” format in the **nta\_file** or comments would have to be added with **bond\_react\_merge\_prep.py** using a “style type” format in the **cta\_file**. Alternatively, if type labels are used LAMMPS will maintain the type labels and a LAMMPS datafile that has been simulated can then be read directly into **bond\_react\_merge.py** again since on the backend the type labels will be used to “rebuild” the comments that LAMMPS removes from the simulated datafiles. One limitation to this is that the code that rebuilds the comments from type labels assumes the **all2lmp.py** style of type labels (i.e., if you want to read in a LAMMPS datafile with type labels only and no comments into **bond\_react\_merge.py**, the type labels must be in the LUNAR style of type labels).

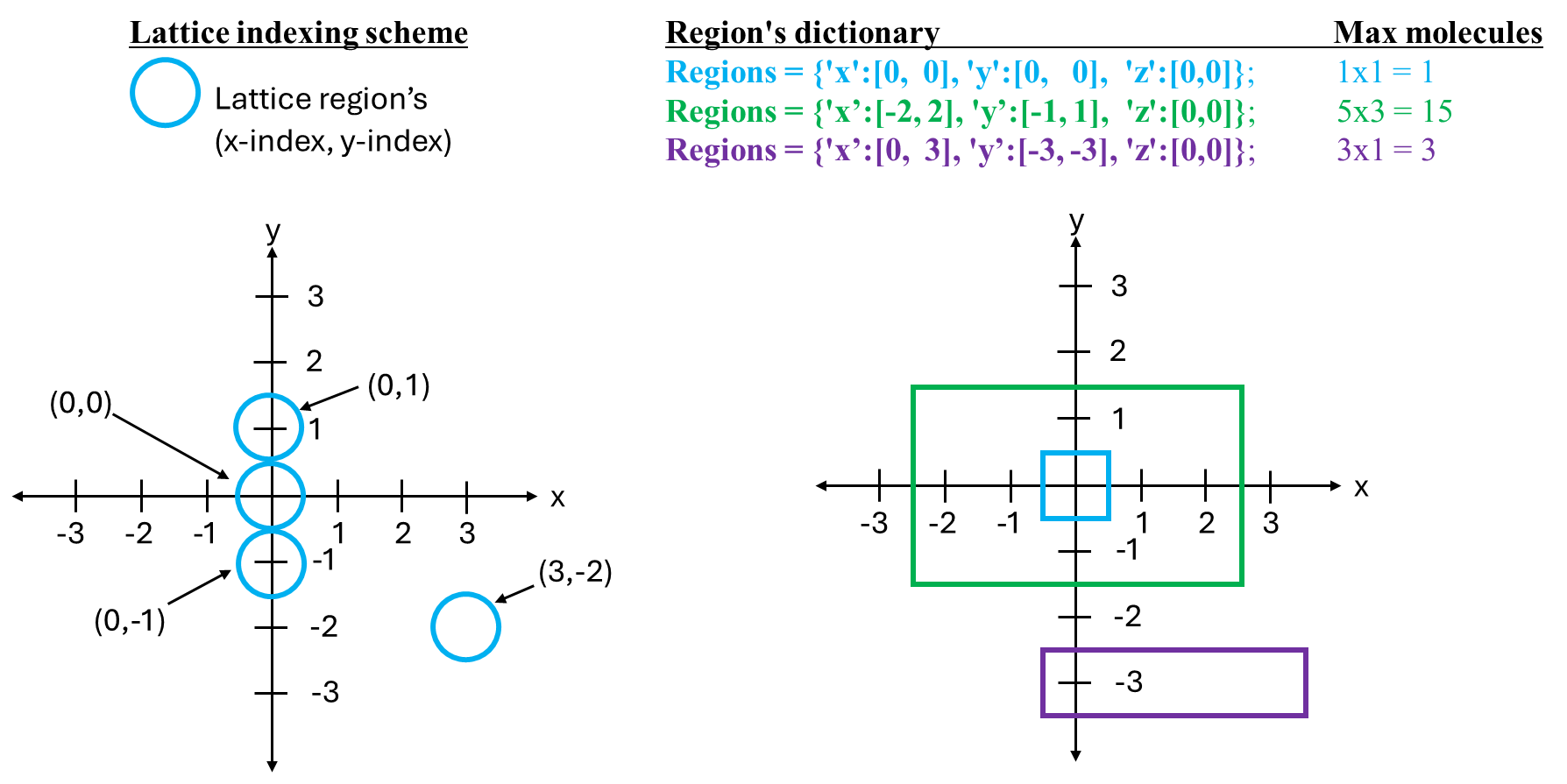
Variable: **reset\_molids** Override tags: **-reset-molids** or **-rm** Usage: **REQUIRED**

* This option only makes sense if you use a LAMMPS atom style that supports molIDs, such as the "full" atom style. Defining molIDs in different ways can be useful as a pre/post processing as it provides a method to group atoms. Additionally, groups of atoms within LAMMPS can be defined via molIDs or certain LAMMPS commands can make use of molIDs (if consistently defined) to control operation of the command such as the “fix bond/react” “molecule [off or inter or intra]” options.
* Is a Python string variable with the following options and their meanings:
  + ‘skip’ will use the molIDs set in the files that are read in. If the file format supplied to **cell\_builder.py** does not have molIDs, every atom molID will default to one.
  + ‘files’ will set molIDs based on the order the files are read into **cell\_builder.py**, where every atom in the first file will be assigned to molID one, every atom in the second file will be assigned to molID two and so on.
  + ‘offset’ will offset the molIDs in each file as each file is read in. If every atoms molID are the same in each file the ‘offset’ method and the ‘files’ method will create the same result. If the file contains different molIDs on different atoms and you wish to maintain the distinction, the ‘offset’ is the method to use.
  + ‘clusters’ will perform a cluster analysis, where the “clusters” of atoms are determined via bonding connectivity, where the criteria for atoms to be part of the same “cluster” is that the atoms must be linked by at least one covalent bond to the cluster. The clusters are then sorted by the number of atoms, where the largest number of atoms is identified as cluster one, then molIDs are incremented and are assigned to each of the remaining clusters (i.e., the largest cluster of atoms will have molID 1, and then the smallest cluster will have a molID of NCLUSTERS found).
* Depending on the different analysis and/or visualization the different **reset\_molid** options may be useful, since they can be used to identity groups of atoms, which can be tedious depending on the type of modeling that is being used.

**Optional Python dictionaries to place in the Header of read-in LAMMPS datafiles**

Additional settings maybe placed in the header of any of the LAMMPS files to read into **cell\_builder.py**. These options will help build more controlled systems, such as placing certain molecule types in a specific region and applying different maximum allowable rotations to the atoms in the LAMMPS datafile. The main purpose of controlled placement of molecules would be for solvation simulation, however these options maybe found useful for other types of simulation cells. The header options are described below:

* The **Shift = {'x': X-shift, 'y': Y-shift, 'z': Z-shift}** dictionary can be added to any file that will have a **qty** value of ZERO, to shift the molecular system by the shift value specified in the file. The atoms and the simulation cell bounds will be shifted in each direction.
  + This method is only applicable to files with a **qty** of 0 (i.e. for defining a system).
* The **Regions = {'x':[xlo, xhi], 'y':[ylo, yhi], 'z':[zlo,zi]}** dictionary maybe placed in the header of the file to control the placement of the atoms in the file to a specific region while **cell\_builder.py** is placing the atoms/molecules. The **ilo** and **ihi** where **i=x,y,z** is described and illustrated below:
  + **ilo** = lower lattice index
  + **ihi** = higher lattice index
  + Where any lattice point between ilo and ihi will be used to place molecules in that region. The lattice points in **cell\_builder.py** are indexed like image flags where (0,0,0) lattice point is the center of the simulation cell, (1,1,1) is one lattice point up in the x, ,y and z-directions and (-1,-1,-1) is one lattice point down in the x, ,y and z-directions. As illustrated by the image below.



* + Notice that when setting the region, you must be aware of the maximum number of molecules that can be placed in that region and adjust the **Regions**, **qty**, and **duplicate** variables accordingly to be able to fit all your molecules in that region. Also note that when **cell\_builder.py** is placing molecules in a specific region, it will do so randomly based on the **seed** value.
  + This method is only applicable to files with a **qty** greater than 0 (i.e. for adding molecules to a system and not defining a system.
* The **Rotations = {'x':rotmax-X, 'y': rotmax-Y, 'z': rotmax-Z}** dictionary maybe placed in the header of the file to control the maximum random rotation to apply to the atoms within the file. This will “shield” the atoms to the global **max\_rotations** variable, in case it is desired to control the maximum amount of rotations of the atoms being placed in a specified region. If **Rotations** dictionary is not in the LAMMPS datafile and **Regions** dictionary is all atoms in the LAMMPS datafile will be controlled by the **max\_rotations** variable found in **cell\_builder.py**. The **Rotations** dictionary in the header cannot be used without the **Regions** dictionary.
  + This method is only applicable to files with a **qty** greater than 0 (i.e. for adding molecules to a system and not defining a system.
* An example of how to place the **Regions** and **Rotations** dictionaries in the header of a LAMMPS datafile before reading that file into **cell\_builder.py** is provided below.

HEADER, **Regions = {'x':[0, 0], 'y':[0, 0], 'z':[0,0]}**; **Rotations = {'x':10, 'y':15, 'z':20}**;

74 atoms

77 bonds

: :

9 atom types

15 bond types

: :

-11.773101351 9.624098649 xlo xhi

-6.993501351 8.614298649 ylo yhi

-3.628504054 2.485395946 zlo zhi

Masses

1 12.01115000 # c2

2 12.01115000 # c3

: : :

* The limitations of this method is that only one LAMMPS datafile that is read into **cell\_builder.py** may have the Regions and Rotations specified like this, since **cell\_builder.py** will give preference of placing atoms in a specified region first and then the remaining lattice points will be used by other files.
* It is advised that if using the region command to set **duplicate** as 1 and then control the system size and stoichiometric ratio via the qty tag of the read in files.
* When using **group\_monomers\_locally** the **Regions** optionis only used on the first building of the sub cell, then if using the **duplicate** variable with a greater than 1 value the sub-cell will be duplicated and randomly placed and rotated to build a large system to model solvation.
* In all the examples above notice the usage of the semi-colon character “;”, this is not a requirement, but if you write enough code you may already be used to using the “;” to create multiple “inline” variables. This is a preference that the writer of **cell\_builder.py** likes but is not required.
* All LUNAR codes maintain HEADER info and will keep appending information to it as files are processed through the different LUNAR codes. This means, that by the time files are ready for usage in **cell\_builder.py** the headers of the files may be very long, and it may be worth deleting certain text from the header to make room for the HEADER options listed above. Additionally, LAMMPS only allows for HEADERS of 256 characters in length so keep that in mind for managing very long headers. If your HEADER goes over 256 characters LAMMPS will likely exit with an “invalid file error”. All LUNAR codes enforce the 256-character limit, but manually typing in the headers may create headers larger than 256 characters.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

The code will write four output files based on the **newfile** name, where each of the output files extension is based on its purpose. The following file extensions will be written with the basename set by the **newfile** string:

* **.data** which is the LAMMPS datafile that is ready for simulation of the randomly mixed and rotated molecules.
* **.log.lunar** is a written log file of the printouts produced when running **cell\_builder.py** which contains the settings used as well as the table of molecule positions and orientations in case someone wants to analyze the spatial and angular distribution of molecules that **cell\_builder.py** creates.
* **.script** is a written LAMMPS script that is ready for simulation to reduce the simulation cell dimensions to create a perfect cube of a specific density to slowly transition the **.data** from a gas to a liquid. The force field section will be initialized from the style hints from the read-in files (if your read-in files do not have style hints the force field styles will be set as N/A. If you generated your initial datafiles with **all2lmp.py** your style hints in the files will be correct and consistent).

# Code: bond\_react\_merge\_prep.py

## Purpose

The purpose of **bond\_react\_merge\_prep.py** is to add comments to a LAMMPS datafile with the **all2lmp.py** styling of commenting such that any LAMMPS datafile can be compatible with **bond\_react\_merge.py**. Two use cases currently exist 1) you have an already simulated LAMMPS datafile that you want to process in **bond\_react\_merge.py** and LAMMPS has stripped away the comments or 2) you have a LAMMPS datafile generated via another tool (such as EMC, moltemplate, msi2lmp, etc) and you want to process the file in **bond\_react\_merge.py**. In either case, comments need to be added to the file in the **all2lmp.py** style of commenting and **bond\_react\_merge\_prep.py** can help with this. The comments will be assigned to the energy coefficients based on the atomID(s) in the Atoms, Bonds, Angles, Dihedrals, and Impropers section of the data file. This means if no atomID(s) in the Atoms, Bonds, Angles, Dihedrals, or Impropers section currently use the CoeffID, a comment cannot be supplied and a general “N/A” or “N/A N/A” etc, comment will be supplied. You may automatically remove these CoeffID(s) if desired using the **rm\_unused\_coeffs** Boolean. **bond\_react\_merge\_prep.py** will leave all parameters and charges as is and only add comments to make the datafile compatible with **bond\_react\_merge.py**.

## Code Variables and Execution

**bond\_react\_merge\_prep.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **bond\_react\_merge\_prep.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 18.

A screenshot of a computer program

Description automatically generated

Figure 18: IDE view of some Python variables for bond\_react\_merge\_prep.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 bond\_react\_merge\_prep.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 19.

A screenshot of a computer

Description automatically generated

Figure 19: Command line override manual printed by running “python3 bond\_react\_merge\_prep.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **bond\_react\_merge\_prep.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **bond\_react\_merge\_prep.py** file. Examples of command line override:

python3 bond\_react\_merge\_prep.py -topo test1.data -cta test1.cta -dir testing\_directory

python3 bond\_react\_merge\_prep.py -topo test2.data -cta test2.cta -rm-na-coeffs T

python3 bond\_react\_merge\_prep.py -gui

python3 bond\_react\_merge\_prep.py -man

python3 LUNAR.py

python3 LUNAR.py 100

It is worth noting that there is a **use\_GUI** Boolean flag in **bond\_react\_merge\_prep.py** to use the GUI or to not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 bond\_react\_merge\_prep.py -gui”.) Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 bond\_react\_mergep\_prep.py -topo test1.data -cta test1.cta”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 bond\_react\_merge\_prep.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **bond\_react\_merge\_prep.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **bond\_react\_merge\_prep.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **bond\_react\_merge\_prep.py** GUI are shown in Figure 20. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **bond\_react\_merge\_prep.py** file, except for the **topofile** and the **cta\_file** variables which will remain as the default this code ships with or whatever the users has manually typed in **bond\_react\_merge\_prep.py**.

A screenshot of a computer

Description automatically generated

Figure 20: LUNAR/ bond\_react\_merge\_prep.py GUI

The **LUNAR.py** and the **bond\_react\_merge\_prep.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **bond\_react\_merge\_prep.py** can be set to different values, where **GUI\_zoom** in **bond\_react\_merge\_prep.py** only affects the GUI that is launched from **bond\_react\_merge\_prep.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **bond\_react\_merge\_prep.py** GUI from **LUNAR.py** GUI**.** Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file must be a LAMMPS **.data** fileand the styling of coefficients does not matter. This file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **cta\_file** Override tags: **-cta** or **-c** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file contains the “**c**omment **t**ype **a**ssignment” of atom types to map onto the atoms in the **topofile**. The format of this file will be discussed below. The **.cta** file extension is important and if the provided file does not have the **.cta** extension, **bond\_react\_merge\_prep.py** will exist with an error about providing an unsupported file type.
* The format of the **.cta** will now be discussed, which is very similar to the **.nta** file of **all2lmp.py**. The **.cta** file supports comments where anything trailing the “#” character will be treated as a comment and ignored. The first line will always be treated as a header and ignored. The **.cta** file is like a LAMMPS **.data** where it is broken down into sections, with each section having a specific style. The only difference is that after declaring each section based on a header keyword there CAN NOT be any white space between the header of the section and the start of information. Within each line, the **.cta** file is interpreted based on whitespace delimiters, where any number of whitespaces may exist but must have at least one whitespace to delimit different groupings of characters. A pseudo **.cta** file is shown below to get a feel for the format of the **.cta** file.

HEADER, this line will always be ignored

section1 style1 # anything following the “#” character will be ignored

1 cp/C

2 cp/C

3 hc/H

:

* There only is one **required** section that must exist in the **.cta** file and it has two different styles: **“style id”** or **“style type”**, which sets that atom type to map onto the atoms that are in the **topofile**. The **“style id”** header implies that the atom type set in this section will be mapped onto the atomIDs found in the **topofile**. The **“style type”** header implies that the atom type set in this section will be mapped onto the atomTypeIDs found in the **topofile**, where an atomTypeID is only in an already built LAMMPS **.data** files and corresponding to the atom typeID (seen in the Masses and Pair Coeffs section of a **.data** file). **bond\_react\_merge\_prep.py** will check that every atom in the **topofile** has an atom type set in the **.cta** file and will exit with an error if not every atom in the **topofile** is supplied with an atom-type/element.

Example of “style id” format of setting atom types, which will be mapped onto atomIDs

style id # atomID atom-type

1 cp/C

2 cp/C

3 hc/H

:

Example of “style type” format of setting atom types, which will be mapped onto atomTypeIDs

style type # atomTypeID atom-type

1 cp/C

2 hc/H

3 op/O

:

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **bond\_react\_merge\_prep.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the file set by the **topofile** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **bond\_react\_merge\_prep.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_cta’** and **topofile == ‘detda.data’):**
        + basename = ‘detda\_cta’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘cta-:’** and **topofile == ‘detda.data’):**
        + basename = ‘cta-detda’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as ‘ANYTEXT’. For example:
      * **(newfile = ‘detda\_renamed’** and **topofile = ‘detda.data’):**
        + basename = ‘detda\_renamed’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.

**Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **bond\_react\_merge\_prep.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix-bond force field the ‘full’ atom style is the best choice. whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice.

Variable: **rm\_unused\_coeffs** Override tags: **-rm-na-coeffs** or **-rm** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to remove any CoeffIDs where atomID(s) do not currently use that coeffID. Typically, unused coeffIDs(s) occur in a *“fix bond/react”* simulation as the *“fix bond/react”* algorithm is changing which atoms are using which coeffIDs. These unused CoeffIDs will not harm while using **bond\_react\_merge.py** or **bond\_react\_merge\_prep.py** but they do make the LAMMPS datafile slightly more unorganized. Most of the LUNAR codes have been devised to keep the LAMMPS datafile organized and neat and this option allows us to keep that philosophy true. The default value of **rm\_unused\_coeffs** should be True.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

The code will write a single output file based on the **newfile** name. The following file extension will be written with the basename set by the **newfile** string:

* **.data** which is the LAMMPS datafile that has comments in it and is ready to be read into **bond\_react\_merge.py**.
* **.log.lunar** which contains the printouts of **bond\_react\_merge\_prep.py**.

# Code: add\_pi\_electrons.py

## Purpose

The purpose of **add\_pi\_electrons.py** is to have a tool to deal with the difficulties of using PCFF-IFF “cg1” and “cge” atom types. These atom types are meant to be used in graphitic type of carbon models such as graphene, graphite, and CNTs. The method that PCFF-IFF uses to add charge to models that are graphitic is to add another atom on each side of the planar carbon atoms to neutralize the charge of the central carbon atom. These two new atoms are described as virtual pi-electrons and are essentially a Drude Oscillator particle but are relatively fixed in place via bond stretch and bond bending parameters and do not allow the responsive charge polarizability as most Drude Oscillator type of particles would allow. The “fixed rigidly bonded pi-electrons” allow for the π-π stacking interaction present in graphite where a responsive charge polarizability is not needed to be modeled.

The **add\_pi\_electrons.py** code also allows users of PCFF-IFF to model all interactions with the native PCFF “cp” atom type and then convert that type to “cg1” and add the “cge” pi-electron. The significance of this is that models may now be built where bonding between the PCFF atom types (using the “cp” atom type) and IFF atom types may be modeled. The origin of PCFF-IFF was to only model interfaces between the PCFF atom types and the IFF atom types and thus no bonding parameters exist to model anything other than interfaces between the two force fields. Utilizing the native “cp” atom type during the initial model building phase using **atom\_typing.py** and **all2lmp.py** will allow the model to be completely PCFF atom types and then the cp Masses and Pair Coeffs can be converted to cg1, the cp-cp Bond Coeffs to be converted to cg1-cg1, the cp-cp-cp Angle Coeffs to be converted to cg1-cg1-cg1, the cp-cp-cp-cp Dihedral Coeffs to be converted to cg1-cg1-cg1-cg1 and the cp-cp-cp-cp Improper Coeffs to be converted to cg1-cg1-cg1-cg1. While all other cp-any atom type interactions stay as PCFF parameterized.

Please note that it is possible to draw molecular models in software such as ChemDraw, Avogadro, BIOVIA Materials Studio, VMD, MolView.org, etc that already have the pi-electrons in them and then use **atom\_typing.py** and **all2lmp.py** to convert that file to a LAMMPS compatible file (or using tools like EMC, moltemplate, msi2lmp, etc). However, for sure **all2lmp.py** and **msi2lmp.exe** will identify the difference between an Improper set of atoms and an AngleAngle set of atoms based on the number of bonded neighbors on the central atom, where three bonded neighbors are used to imply the interaction as an Improper interaction and any more than three bonded neighbors are used to imply the interaction as an AngleAngle interaction. This means that if you supply a file to tools like **all2lmp.py** and **msi2lmp.exe** that already has pi-electrons in them they will assume that the cg1-cg1-cg1-cg1 is an AngleAngle set of atoms, and it is supposed to be an Improper set of atoms. Additionally, the cge virtual pi-electron only uses bonds and angles to hold the atom in its place, and tools like **all2lmp.py** and **msi2lmp.exe** will by default build cg1-cg1-cg1-cge Dihedrals and cg1-cg1-cge-cg1 AngleAngle interactions that are not needed, leading to a slower MD model with more energy contribution to go wrong and crash the simulation. Due to all the things listed above it is recommended to **NEVER** build a system with pi-electrons in them, but to add them later using **add\_pi\_electrons.py**.

## Code Variables and Execution

**add\_pi\_electrons.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **add\_pi\_electrons.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 21.

A screenshot of a computer

Description automatically generated

Figure 21: IDE view of some Python variables for add\_pi\_electrons.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 add\_pi\_electrons.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 22.

A screenshot of a computer screen

Description automatically generated

Figure 22: Command line override manual printed by running “python3 add\_pi\_electrons.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **add\_pi\_electrons.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **add\_pi\_electrons.py** file. Examples of command line override:

python3 add\_pi\_electrons.py -topo test1.data -rq T -dir testing\_directory

python3 add\_pi\_electrons.py -topo test2.data -cg1 T -pi-electrons T

python3 add\_pi\_electrons.py -gui

python3 add\_pi\_electrons.py -man

python3 LUNAR.py

python3 LUNAR.py 75

It is worth noting that there is a **use\_GUI** Boolean flag in **add\_pi\_electrons.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 add\_pi\_electrons.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 add\_pi\_electrons.py -topo test1.data”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 add\_pi\_electrons.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **add\_pi\_electrons.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **add\_pi\_electrons.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **add\_pi\_electrons.py** GUI are shown in Figure 23. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **add\_pi\_electrons.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the user has manually typed in **add\_pi\_electrons.py**.

A screenshot of a computer

Description automatically generated

Figure 23: LUNAR/ add\_pi\_electrons.py GUI

The **LUNAR.py** and the **add\_pi\_electrons.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **add\_pi\_electrons.py** can be set to different values, where **GUI\_zoom** in **add\_pi\_electrons.py** only affects the GUI that is launched from **add\_pi\_electrons.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **add\_pi\_electrons.py** GUI from **LUNAR.py** GUI**.** Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the file that is assigned by the variable or the override tag. This file must be a LAMMPS **.data** fileand the styling of coefficients must be class2 and should either be PCFF or IFF parameterized. The atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **types2convert** Override tags: **-types** or **-t2c** Usage: **REQUIRED**

* Is a Python list to add any atomTypeIDs to convert from either a “cp” and/or “c5” atom type to a “cg1” atom type. Additionally, the atomTypeIDs present in this list can have pi-electrons added to them by using the **add\_pi\_electrons** Boolean variable or charges set as “cg1” by using the **reset\_charges** Boolean variable. You may supply the atomTypeIDs as int values or if the read-in **topofile** has atom type comments in the Masses section you may supply a string value of the atomType characters (i.e., “cp” or “c5” may be used to identify the atomTypeIDs to convert to “cg1” or add pi electrons to).

Variable: **newfile** Override tags: **-newfile** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output filename(s). The following options exist for using the **newfile** string for setting the output file basenames:
  + **if newfile** starts with **‘:’** or ends with **‘:’:**
    - The output filename(s) will be the same as the input filename(s), but will have a suffix or prefix added to the file basename. The following are examples:
      * **Suffix (newfile == ‘:\_pi\_electron’** and **topofile == ‘cnt.data’):**
        + basename = ‘cnt\_ pi\_electron’, where the ‘:’character acts as a placeholder for the **topofile** basename.
      * **Prefix (newfile == ‘pi\_electron-:’** and **topofile == ‘cnt.data’):**
        + basename = ‘pi\_electron -cnt’, where the ‘:’character acts as a placeholder for the **topofile** basename.
    - **Recommended usage:** **common** and **safe** as this method is safe and output filename(s) carry similar names to input filename(s).
  + **if newfile == ‘ANYTEXT’:**
    - The output filename(s) will be set as ‘ANYTEXT’. For example:
      * **(newfile = ‘cnt\_renamed’** and **topofile = ‘cnt.data’):**
        + basename = ‘cnt\_renamed’
    - **Recommended usage:** **occasional** and **safe** as output filename(s) no longer carry similar names as output filename(s), but is safe as output filename(s) will not overwrite input filename(s)
  + **if newfile == ‘’:**
    - The output filename(s) will be the same as the input filename(s). This can be a **dangerous** option as you may inadvertently overwrite a file and then must assume the file to contain certain information, but it contains other information.
    - **Recommended usage:** **rare** and **dangerous** as this could lead to using incorrect files if not paying attention very carefully.

Variable: **convert2cg1** Override tags: **-convert2cg1** or **-cg1** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to have the atomTypeIDs in **types2convert** to be converted to cg1 parameters. For example, if only converting the PCFF cp atom type the cp Masses and Pair Coeffs will be converted to cg1, the cp-cp Bond Coeffs will be converted to cg1-cg1, the cp-cp-cp Angle Coeffs will be converted to cg1-cg1-cg1, the cp-cp-cp-cp Dihedral Coeffs will be converted to cg1-cg1-cg1-cg1 and the cp-cp-cp-cp Improper Coeffs will be converted to cg1-cg1-cg1-cg1. All the corresponding crossterms will be converted accordingly. Please note that when using **convert2cg1** the atomic charges will be left alone and must be updated using **add\_pi\_electrons** or **reset\_charges** Boolean variables.

Variable: **add\_pi\_electrons** Override tags: **-pi-electrons** or **-pie** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to add pi-electrons to the atomTypeIDs listed in **types2convert**. When the pi-electrons are added the bonding and angle topologies will be updated and the parameters will be assigned accordingly. Using **add\_pi\_electrons** will also add in the proper pi-electron charge and update the charge on all atomTypeIDs in **types2convert**, thus if **add\_pi\_electrons** is True there is no need to have **reset\_charges** as True.

Variable: **reset\_charges** Override tags: **-reset-charges** or **-rq** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to reset the charge of all atomTypeIDs listed in **types2convert** to the cg1 atom type charge. Note that this will make the system non-charge neutral if no pi-electrons are added to the system.

Variable: **net\_zero\_charge** Override tags: **-charge0** or **-q0** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) forcing the system net charge to zero by adding a fixed charge value to all atoms that do not belong to the atomTypeIDs listed in **types2convert** (i.e., every atom that has not been converted to cg1 or is not a cge atom type will have their charges scaled by a fixed value to force the net system charge to be zero). This option is useful when trying to model bonding between PCFF atom types and IFF atom types since the charging method used is inconsistent between the two and cannot guarantee a net zero-charged system. If your system is pure graphite/graphene or a pure CNT, this option can be set as False, but if you have functionalities of bonded molecules to the graphite/graphene or CNTs, this option should be True.

Variable: **neighbor\_charge\_constraint** Override tags: **-neigh\_charge** or **-nq** Usage: **REQUIRED**

* Is a Python string variable to set how to handle charge on “compounds” (materials that are not either pure graphite or CNT or fullerene, where there are other atom types bonded to the aromatic carbon atoms). The IFF charge model using the virtual pi-electrons was originally formulated only for pure graphitic systems and thus when trying to use the pi-electron charge with system that have different atom types other than aromatic carbons, the total system charge will become none-neutral. One method around this is to enforce the system charge to be neutral via the **net\_zero\_charge** option, however this method can have adverse effects such as completely removing charge from all none-aromatic atoms or arbitrarily scaling all none-aromatic atoms by the same charge value to achieve charge neutrality. This option provides a greater level of control to enforce charge neutrality. If your system is pure graphite or CNT or fullerene this option can be set to any supported options as none of the options effect this type of system. The following options exist:
  + ‘none’ which does not apply any constraint to how neighbor charges are handled. If there are any first neighbors bonded to the aromatic carbon atoms, this method may result in a none-charge neutral system.
  + ‘check-neighbors’ which checks that all neighbors are aromatic and only places the pi-electron if all neighbors are aromatic. This method is **recommended** if you are unsure what method to use as these type of atoms already have charge modeled and reducing the number of atoms in the simulation. Reducing the number of added pi-electrons results in the quickest simulation times.
  + ‘accumulate-carbon’ which will accumulate any residual charge into the carbon atom, to ensure the “local grouping of atoms” stays charge neutral and thus the entire system will remain charge neutral.
  + ‘accumulate-pi-electron’ which will accumulate any residual charge into the pi-electron atoms, to ensure the “local grouping of atoms” stays charge neutral and thus the entire system will remain charge neutral.
  + ‘accumulate-neighbor’ which will accumulate any residual charge into the first neighboring atom, to ensure the “local grouping of atoms” stays charge neutral and thus the entire system will remain charge neutral.

Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **add\_pi\_electrons.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix-bond force field the ‘full’ atom style is the best choice. whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **bond\_react\_merge\_prep.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the file set in the **topofile** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **add\_pi\_electrons.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file. LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and does help with the manual creation of the files. **add\_pi\_electrons.py** can add type labels into the written merged files even if the read-in files DO NOT have type labels defined since all of LUNAR has a consistent commenting scheme where **add\_pi\_electrons.py** can derive the type labels due to this consistent commenting scheme. If type labels are used and a LAMMPS simulation is run LAMMPS will strip all comments from the simulated datafiles, but not the type labels. Since **bond\_react\_merge.py** requires **all2lmp.py** style of comments to merge/unify the coeffs it means a LAMMPS datafile that has been simulated is not compatible with **bond\_react\_merge.py** and comments would either be added in by processing the file with **all2lmp.py** using a “style type” format in the **nta\_file** or comments would have to be added with **bond\_react\_merge\_prep.py** using a “style type” format in the **cta\_file**. Alternatively, if type labels are used LAMMPS will maintain the type labels and a LAMMPS datafile that has been simulated can then be read directly into **bond\_react\_merge.py** again since on the backend the type labels will be used to “rebuild” the comments that LAMMPS removes from the simulated datafiles. One limitation to this is that the code that rebuilds the comments from type labels assumes the **all2lmp.py** style of type labels (i.e., if you want to read in a LAMMPS datafile with type labels only and no comments into **bond\_react\_merge.py**, the type labels must be in the LUNAR style of type labels).

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

The code will write a single output file based on the **newfile** name, where the output file extension is based on its purpose. The following file extension will be written with the basename set by the **newfile** string:

* **.data** which is the LAMMPS datafile that has all the modifications done to it based on the **types2convert**, **convert2cg1**, **add\_pi\_electrons**, **reset\_charges**, and **net\_zero\_charge** variables. The placement of the pi-electrons in **add\_pi\_electrons.py** does not use an energy minimization technique, but a vector approach to make the bond length exactly as the cg1-cge bond length and the angles be exactly the cg1-cg1-cge and cge-cg1-cge angle, but it is still recommended to equilibrate and minimize the structure after pi-electrons have been added to ensure the lowest possible energy configuration has been found.
* **.log.lunar** which contains the printouts of **add\_pi\_electrons.py**.

# Code: atom\_removal.py

## Purpose

The purpose of **atom\_removal.py** is to have a tool to remove atoms from a LAMMPS datafile and then renumber the atoms, bonds, angles, dihedrals and impropers to be contiguous. The energy coefficients will not be adjusted after deleting atoms so there may be energy coefficients that are not used by any atom, bond, angle, or dihedral, or improper. Additionally, **atom\_removal.py** will write the energy coefficient comments and the style hints (if they exist in the read-in file) in the output LAMMPS datafile. The initial intended use case of **atom\_removal.py** was to fix templates that are built for a LAMMPS *“fix bond/react”* simulation where the template was accidentally built with atoms that should not be in the template. The most typical case is when individuals accidentally leave extra atoms on any atom that should be an “edge atom” in the template (“edge atoms” should only have one bonded neighbor, but it seems that even experienced *“fix bond/react”* often forget this when making the reaction templates). **atom\_removal.py** can be used to fix quickly and easily these types of *“fix bond/react”* templates where the energy coefficient comments are maintained in the output, such that the **atom\_removal.py** output LAMMPS datafille can be used immediately in **bond\_react\_merge.py** to create an entire set of files for a *“fix bond/react”* simulation.

Since the initial intended use case of **atom\_removal.py** is to remove a small number of atoms from a system it can be difficult to use it to remove many atoms from a system or determine which atoms to remove based on some grouping definition. If you need such a tool, **atom\_removal.py** may not be the correct tool to use. **atom\_removal.py** does allow the user to select which atomIDs or TypeIDs to remove, where if there is a certain TypeID(s) that the user wants to remove, many atoms can be removed by specifying a very small amount of information. The most obvious use case is to remove atoms by TypeID(s) when using PCFF-IFF cg1/cge virtual pi-electrons and it is desired to remove the virtual pi-electrons from the system.

## Code Variables and Execution

**atom\_removal.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **atom\_removal.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 24.

A screenshot of a computer program

Description automatically generated

Figure 24: IDE view of some Python variables for atom\_removal.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 atom\_removal.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 25.

A screenshot of a computer screen

Description automatically generated

Figure 25: Command line override manual printed by running “python3 atom\_removal.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **atom\_removal.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **atom\_removal.py** file. Examples of command line override:

python3 atom\_removal.py -topo test1.data -method atomIDs -atoms2remove 1,2,3 -dir .

python3 atom\_removal.py -topo test2.data -method TypeIDs -atoms2remove 1,2 -atomstyle full

python3 atom\_removal.py -gui

python3 atom\_removal.py -man

python3 LUNAR.py

python3 LUNAR.py 125

It is worth noting that there is a **use\_GUI** Boolean flag in **atom\_removal.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 atom\_removal.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 atom\_removal.py -topo test1.data”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 atom\_removal.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **atom\_removal.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **atom\_removal.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **atom\_removal.py** GUI are shown in Figure 26. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **atom\_removal.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the users has manually typed in **atom\_removal.py**.

A screenshot of a computer

Description automatically generated

Figure 26: LUNAR/ atom\_removal.py GUI

The **LUNAR.py** and the **atom\_removal.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **atom\_removal.py** can be set to different values, whereas **GUI\_zoom** in **atom\_removal.py** only affects the GUI that is launched from **atom\_removal.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **atom\_removal.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the LAMMPS datafile that is assigned by the variable or the override tag. This file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **method** Override tags: **-method** or **-m** Usage: **REQUIRED**

* Is a Python string to set the method to determine which atoms should be removed from the systems. The following strings/methods are supported:
  + ‘atomIDs’ which will identify atoms to remove based on their atomID that will be supplied in the **atoms2remove** list.
  + ‘typeIDs’ which will identify atoms to remove based on their atomTypeID that will be supplied in the **atoms2remove** list.

Variable: **atoms2remove** Override tags: **-atoms2remove** or **-a2r** Usage: **REQUIRED**

* Is a Python list to add any atomIDs or atomTypeIDs to remove based on the remove **method**. The atomIDs or typeIDs should be integer values with no limit to the number of them to add to the **atoms2remove** list. However, it is envisioned that users visualize their LAMMPS datafile and pick a small number of atoms to remove by specifying their atomIDs or typeIDs in the **atoms2remove** list.

Variable: **newfile** Override tags: **-ext** or **-e** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output file(s) “basename”. **atom\_removal.py** will find the “basename” of the file set via the **topofile** and then append the string assigned set by the **newfile** variable to the “basename” with an underscore between the two strings (i.e., **topofile** = ‘pre\_rxn1.data’; the basename = ‘pre\_rxn1’ and **newfile**=‘rm\_atoms’, the output files will carry the name ‘pre\_rxn1\_rm\_atoms’ “basename” with the file extension based on the type of output file).

Variable: **atom\_style** Override tags: **-atomstyle** or **-as** Usage: **REQUIRED**

* Is a Python string variable in which to set the style of the “Atoms” section of the written LAMMPS datafile. Currently, **atom\_removal.py** supports ‘full’, ‘charge’, and ‘molecular’. In general, if you are using a fix-bond force field the ‘full’ atom style is the best choice. whereas if you are using a bond order-based force field the ‘charge’ atom style is the best choice.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **atom\_removal.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the first file in the **files** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **atom\_removal.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **include\_type\_labels** Override tags: **-type-labels** or **-tl** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to include type labels in the written **.data** file. LAMMPS's new type labels option is intended to simplify the methods required to generate files for using the LAMMPS *“fix bond/react”* command and does help with the manual creation of the files. **atom\_removal.py** can add type labels into the written merged files even if the read-in files DO NOT have type labels defined since all of LUNAR has a consistent commenting scheme where **atom\_removal.py** can derive the type labels due to this consistent commenting scheme. If type labels are used and a LAMMPS simulation is run LAMMPS will strip all comments from the simulated datafiles, but not the type labels. Since **bond\_react\_merge.py** requires **all2lmp.py** style of comments to merge/unify the coeffs it means a LAMMPS datafile that has been simulated is not compatible with **bond\_react\_merge.py** and comments would either be added in by processing the file with **all2lmp.py** using a “style type” format in the **nta\_file** or comments would have to be added with **bond\_react\_merge\_prep.py** using a “style type” format in the **cta\_file**. Alternatively, if type labels are used LAMMPS will maintain the type labels and a LAMMPS datafile that has been simulated can then be read directly into **bond\_react\_merge.py** again since on the backend the type labels will be used to “rebuild” the comments that LAMMPS removes from the simulated datafiles. One limitation to this is that the code that rebuilds the comments from type labels assumes the **all2lmp.py** style of type labels (i.e., if you want to read in a LAMMPS datafile with type labels only and no comments into **bond\_react\_merge.py**, the type labels must be in the LUNAR style of type labels).

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

The code will write a single output file based on the **newfile** name, where the output file extension is based on its purpose. The following file extension will be written with the basename set by the **newfile** string:

* **.data** which is the LAMMPS datafile that has atoms removed and the atoms, bonds, angles, dihedrals, and impropers renumbered to be contiguous. If the read-in **topofile** had comments and/or style hints, they would be transferred to the written **.data** file.
* **.log.lunar** which contains the printouts of **atom\_removal.py**.

# Code: lmp2SYBYLmol2.py

## Purpose

The purpose of **lmp2SYBYLmol2.py** is to have a tool to convert any LAMMPS datafile to a SYBYL **.mol2** format such that chemical drawing or visualization tools such as ChemDraw, Avogadro, VMD, Material Studio, etc may be used open, view and/or modify a LAMMPS simulated molecule/system. The intended usage case is to either visualize a molecule/system or to be able to draw in or remove atoms using tools such as ChemDraw, Avogadro, VMD, Material Studio, etc. Creating a loop of processing within LUNAR where the sequence would be **atom\_typing.py** -> **all2lmp.py** -> LAMMPS simulation -> **lmp2SYBYLmol2.py** -> chemical drawing software -> **atom\_typing.py**, for an arbitrary number of cycles.

## Code Variables and Execution

**lmp2SYBYLmol2.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **lmp2SYBYLmol2.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 27.

A screenshot of a computer program

Description automatically generated

Figure 27: IDE view of some Python variables for lmp2SYBYLmol2.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 lmp2SYBYLmol2.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 28.

A screenshot of a computer program

Description automatically generated

Figure 28: Command line override manual printed by running “python3 lmp2SYBYLmol2.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **lmp2SYBYLmol2.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **lmp2SYBYLmol2.py** file. Examples of command line override:

python3 lmp2SYBYLmol2.py -topo test1.data -dir .

python3 lmp2SYBYLmol2.py -topo test2.data -rm-pbc-bonds T

python3 lmp2SYBYLmol2.py -gui

python3 lmp2SYBYLmol2.py -man

python3 LUNAR.py

python3 LUNAR.py 100

It is worth noting that there is a **use\_GUI** Boolean flag in **lmp2SYBYLmol2.py** to use the GUI or to not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 lmp2SYBYLmol2.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 lmp2SYBYLmol2.py -topo test1.data”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 lmp2SYBYLmol2.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **lmp2SYBYLmol2.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUIs, where **lmp2SYBYLmol2.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **lmp2SYBYLmol2.py** GUI are shown in Figure 29. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **lmp2SYBYLmol2.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the user has manually typed in **lmp2SYBYLmol2.py**.

A screenshot of a computer

Description automatically generated

Figure 29: LUNAR/ lmp2SYBYLmol2.py GUI

The **LUNAR.py** and the **lmp2SYBYLmol2.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **lmp2SYBYLmol2.py** can be set to different values, where **GUI\_zoom** in **lmp2SYBYLmol2.py** only affects the GUI that is launched from **lmp2SYBYLmol2.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **lmp2SYBYLmol2.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the LAMMPS datafile that is assigned by the variable or the override tag. This file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **lmp2SYBYLmol2.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the first file in the **files** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **lmp2SYBYLmol2.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **remove\_PBC\_bonds** Override tags: **-rm-pbc-bonds** or **-rm** Usage: **REQUIRED**

* Will remove periodic bonds, where these bonds are determined to be periodic via the minimum image convention. If the intended use is to visualize a molecule system in tools such as ChemDraw, Avogadro, VMD, Material Studio, etc it may be desired to remove the periodic bonds, which will appear as long bonds spanning the system.

Variable: **mass\_map** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as elemental symbols and the corresponding values being a list of masses associated with that element type. The purpose of the **mass\_map** dictionary is to set the element type of each atom type in the **.data** file, which will be identified via the mass of each atom type.

## Outputs

The code will write a single output files based on the **topofile** basename, where the output file extension is based will be **.mol2**.

# Code: cluster\_analysis.py

## Purpose

The purpose of **cluster\_analysis.py** is to have a tool analyze a polymer/system bonding connectivity/network. The main use case is to pair this analysis with a LAMMPS *“fix bond/react”* simulation output, however, **cluster\_analysis.py** can be used to analyze any polymer or systems network bonding connectivity. A bonded cluster is defined to be a group of atoms if each atom is bonded to the group by a minimally one covalent bond. The metrics that **cluster\_analysis.py** currently supports are:

* Clusters from 1-to-N clusters in the systems where the following metrics are computed for each bonded cluster:
  + Number of atoms (size)
  + Percent number of atoms (% size)
  + Mass (in AMU)
  + Percent Mass (% mass)
* Extent of reaction (p)
* Critical extent of reaction (pg)
* Degree of Polymerization (Xn)
* Weight-average molar mass (Mw)
* Number-average molar mass (Mn)
* Higher-average molar mass (Mz)
* Higher-average molar mass (Mz+1)
* Weight-average reduced molar mass (RMW)

**cluster\_anaylsis.py** will compute all these properties for a single LAMMPS datafile. If you have a “large batch” of LAMMPS datafiles that you want to compute these properties and log to a **.csv** file, investigate the **auto\_cluster\_analysis.py** code.

## Code Variables and Execution

**cluster\_analysis.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **cluster\_analysis.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 30.

A screenshot of a computer program

Description automatically generated

Figure 30: IDE view of some Python variables for cluster\_analysis.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 cluster\_analysis.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 31.

A screenshot of a computer program

Description automatically generated

Figure 31: Command line override manual printed by running “python3 cluster\_analysis.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **cluster\_analysis.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **cluster\_analysis.py** file. Examples of command line override:

python3 cluster\_analysis.py -topo test1.data -n0 100 -fav 2

python3 cluster\_analysis.py -topo test2.data -txt T

python3 cluster\_analysis.py -gui

python3 cluster\_analysis.py -man

python3 LUNAR.py

python3 LUNAR.py 100

It is worth noting that there is a **use\_GUI** Boolean flag in **cluster\_analysis.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 cluster\_analysis.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 cluster\_analysis.py -topo test1.data”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 cluster\_analysis.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **cluster\_analysis.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **cluster\_analysis.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **cluster\_analysis.py** GUI are shown in Figure 32. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **cluster\_analysis.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the users have manually typed in **cluster\_analysis.py**.

A screenshot of a computer

Description automatically generated

Figure 32: LUNAR/ cluster\_analysis.py GUI

The **LUNAR.py** and the **cluster\_analysis.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **cluster\_analysis.py** can be set to different values, whereas **GUI\_zoom** in **cluster\_analysis.py** only affects the GUI that is launched from **cluster\_analysis.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **cluster\_analysis.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the LAMMPS datafile that is assigned by the variable or the override tag. This file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **N0** Override tags: **-n0 &&** Variable: **fav** Override tags: **-fav** Usage: **REQUIRED**

* Both the **N0** and **fav** are used to compute the following metrics:
  + Extent of reaction ()
  + Critical extent of reaction ()
  + Degree of Polymerization ()
  + Where:
    - p is the extent of the reaction
    - pg is the critical extent of the reaction (gel point)
    - N0 Number of initial molecules before polymerization
    - N is the number of clusters in the current LAMMPS datafile
    - fav is the average number of functional groups present per monomer unit
    - Xn is the degree of polymerization
* **N0** is an integer value and **fav** is a float or integer value. Both metrics are system-dependent, and MUST BE set based on your system if you desire to use the extent or reaction and/or critical extent of reaction and/or degree of polymerization metrics to characterize your system.
* Example of computing **N0** and **fav** for an EPON-862 system (DETDA and DGEBF molecules):
  + 1 DETDA molecule (4-functional groups)
  + 2 DGEBF molecules (2 functional groups)
  + mix = 1\*DETDA + 2\*DGEBF (2 DETDA to every 1 DGEBF)
  + fav = (1-DETDA\*4-functional-groups + 2-DGEBF\*2-functional-groups)/3-molecules
  + fav = (1\*4 + 2\*2)/3 = 2.66
  + N0 = 3

Variable: **txtfile** Override tags: **-txt** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write a **.txt** file to log the printouts of the code.

## Outputs

The code will write a single output files based on the **topofile** basename, where the output file extension will be **.txt** if **txtfile** is True.

# Code: auto\_cluster\_analysis.py

## Purpose

The purpose of **auto\_cluster\_analysis.py** is to have a tool analyze a polymer/system bonding connectivity/network. The main use case is to pair this analysis with a LAMMPS *“fix bond/react”* simulation output, however, **auto\_cluster\_analysis.py** can be used to analyze any polymer or systems network bonding connectivity. A bonded cluster is defined to be a group of atoms if each atom is bonded to the group by a minimally one covalent bond. The metrics that **auto\_cluster\_analysis.py** currently supports are:

* Clusters from 1-to-N clusters in the systems where the following metrics are computed for each bonded cluster:
  + Number of atoms (size)
  + Percent number of atoms (% size)
  + Mass (in AMU)
  + Percent Mass (% mass)
* Extent of reaction (p)
* Critical extent of reaction (pg)
* Degree of Polymerization (Xn)
* Weight-average molar mass (Mw)
* Number-average molar mass (Mn)
* Higher-average molar mass (Mz)
* Higher-average molar mass (Mz+1)
* Weight-average reduced molar mass (RMW)

**auto\_cluster\_analysis.py** differs from **cluster\_analysis.py** in the fact that **auto\_cluster\_analysis.py** will load in a directory of LAMMPS datafiles and iterate through the directory automatically and then store the outputs (user-defined based on some small amount of Python coding) in a **.csv** file. This can reduce the tedious process of analyzing “large batches” of LAMMPS datafiles with **cluster\_analysis.py** however customizing the output **.csv** file format requires a small amount of Python code manipulation. The default values written to the **.csv** files are enough to compute the gel point of polymerization simulation, but others may be added if the users desire them.

## Code Variables and Execution

**auto\_cluster\_analysis.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **auto\_cluster\_analysis.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 33.

A screenshot of a computer code

Description automatically generated

Figure 33: IDE view of some Python variables for auto\_cluster\_analysis.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 auto\_cluster\_analysis.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 34.

A screenshot of a computer screen

Description automatically generated

Figure 34: Command line override manual printed by running “python3 auto\_cluster\_analysis.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **auto\_cluster\_analysis.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **auto\_cluster\_analysis.py** file. Examples of command line override:

python3 auto\_cluster\_analysis.py -dir replicate1\_rxn\_files -n0 100 -fav 2

python3 auto\_cluster\_analysis.py -dir replicate2\_rxn\_files -txt T

python3 auto\_cluster\_analysis.py -gui

python3 auto\_cluster\_analysis.py -man

python3 LUNAR.py

python3 LUNAR.py 75

It is worth noting that there is a **use\_GUI** Boolean flag in **auto\_cluster\_analysis.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 auto\_cluster\_analysis.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 auto\_cluster\_analysis.py -dir replicate1\_rxn\_files”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 auto\_cluster\_analysis.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **auto\_cluster\_analysis.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **auto\_cluster\_analysis.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **auto\_cluster\_analysis.py** GUI are shown in Figure 35. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **auto\_cluster\_analysis.py** file, except for the **files\_directory** variable which will remain as the default this code ships with or whatever the user has manually typed in **auto\_cluster\_analysis.py**.

A screenshot of a computer

Description automatically generated

Figure 35: LUNAR/ auto\_ cluster\_analysis.py GUI

The **LUNAR.py** and the **auto\_cluster\_analysis.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **auto\_cluster\_analysis.py** can be set to different values, where **GUI\_zoom** in **auto\_cluster\_analysis.py** only affects the GUI that is launched from **auto\_cluster\_analysis.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **auto\_cluster\_analysis.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **files\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will read all the LAMMPS datafile that are in the files directory. The files can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section.

Variable: **newfile** Override tags: **-new-file** or **-nf** Usage: **REQUIRED**

* Is a Python string variable in which to set the new output file(s) “basename”. Unlike most of the LUNAR codes, the “basename” set by **newfile**, **auto\_cluster\_analysis.py** is the actual filename output, whereas, in the rest of the LUNAR codes, the **newfile** string would get append to the **topofile** “basename”. The output file extension will be a **.csv** file with the info logged based on the Python “logger” dictionary located around line 215 of the code as shown below:

A computer screen shot of a computer code

Description automatically generated

* Where the output .csv file format then looks like this:

A screenshot of a table

Description automatically generated

* The “logger” dictionary may be modified as desired where the available attributes and logging options are provided in the comment block around lines 165-195:

A computer screen with text and numbers

Description automatically generated

* The available attributes will not be documented here since they are documented in the code and adjusting the output **.csv** file format requires users to be able to do a minimum amount of coding.

Variable: **N0** Override tags: **-n0 &&** Variable: **fav** Override tags: **-fav** Usage: **REQUIRED**

* Both the **N0** and **fav** are used to compute the following metrics:
  + Extent of reaction ()
  + Critical extent of reaction ()
  + Degree of polymerization ()
  + Where:
    - p is the extent of the reaction
    - pg is the critical extent of the reaction (gel point)
    - N0 Number of initial molecules before polymerization
    - N is the number of clusters in the current LAMMPS datafile
    - fav is the average number of functional groups present per monomer unit
    - Xn is the degree of polymerization
* **N0** is an integer value and **fav** is a float or integer value. Both metrics are system-dependent, and MUST BE set based on your system if you desire to use the extent or reaction and/or critical extent of reaction and/or degree of polymerization metrics to characterize your system.
* Example of computing **N0** and **fav** for an EPON-862 system (DETDA and DGEBF molecules):
  + 1 DETDA molecule (4-functional groups)
  + 2 DGEBF molecules (2 functional groups)
  + mix = 1\*DETDA + 2\*DGEBF (2 DETDA to every 1 DGEBF)
  + fav = (1-DETDA\*4-functional-groups + 2-DGEBF\*2-functional-groups)/3-molecules
  + fav = (1\*4 + 2\*2)/3 = 2.66
  + N0 = 3

Variable: **txtfile** Override tags: **-txt** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to write a **.txt** file to log the printouts of the code.

## Outputs

The code will write a single output file based on each **topofile** basename that the code has iterated through, where the output file extension will be **.txt** if **txtfile** is True and a **.csv** file with the basename set as the **newfile** string.

# Code: free\_volume.py

## Purpose

The purpose of **free\_volume.py** is to have a tool to estimate the free volume, percent free volume, free volume spatial distribution, and free volume connectivity in a LAMMPS simulated system. The free volume is computed via:

* [Free volume] = [Simulation cell volume] – [Atom occupancy volume]

Where the [Simulation cell volume] is defined via the LAMMPS simulation cell box dimensions and the [Atom occupancy volume] is defined via the sum of the volume encapsulated by the vdw radii of each atom in the system. One of the most difficult issues in determining the sum encapsulated volumes from the vdw radii of all atoms in the system is the overlapped vdw radii of bonded atoms. **free\_volume.py**, therefore, computes the [Atom occupancy volume], by dividing the simulation cell box into small elemental voxels and then assigns each voxel to an atomID, where once the voxel is “owned” by an atomID it cannot be “owned” by any other atomID and then just sums up the elemental volume of the “owned” voxels.

## Code Variables and Execution

**free\_volume.py** has three methods to run/interact with the code. One method is to run in an IDE (integrated design environment), where all Python variables can be changed in the IDE or a text editor. Another method is to run at the command line, where the Python variables can be manipulated via command line overrides. Lastly, GUI’s have been built where the user can run the code from a GUI, where all GUI defaults are initialized from the code file (thus you can change the defaults which the GUI will load with by adjusting the code file and the GUI has a save button to automatically update the code file from the GUI settings). To run the code in IDE mode, open **free\_volume.py** in your favorite IDE (I like Anaconda’s Spyder IDE), where you will see all the Python variables as shown in Figure 36.

A screenshot of a computer program

Description automatically generated

Figure 36: IDE view of some Python variables for free\_volume.py (NOTE examples provided in the boxed-in comments).

Once opened in an IDE or text editor you can manipulate the Python variables as needed (discussion of what the variables do will be discussed below) to adjust the read-in files and the code operations, then run based on the IDE run method. To run the code at the command line type “python3 free\_volume.py -man” and run to get access to the in-built command line override manual and how to adjust the Python variables at the command line shown in Figure 37.

A screenshot of a computer program

Description automatically generated

Figure 37: Command line override manual printed by running “python3 free\_volume.py -man”

The command line override manual should provide enough guidance on how to override the hard-coded values assigned to the Python variables found in **free\_volume.py**. Please note that when using the command line override if not all arguments are given via the “-tag tag-input”, the code will default to what is written in the **free\_volume.py** file. Examples of command line override:

python3 free\_volume.py -topo test1.data -pbc p-p-p -voxel 0.1

python3 free\_volume.py -topo test2.data -run-mode numba-p -dir .

python3 free\_volume.py -gui

python3 free\_volume.py -man

python3 LUNAR.py

python3 LUNAR.py 125

It is worth noting that there is a **use\_GUI** Boolean flag in **free\_volume.py** to use the GUI or not use the GUI. The command line interface offers a “-gui” command line option to launch the GUI from the command line no matter the Boolean given to the **use\_GUI** variable (i.e., if use\_GUI = False you can still launch the GUI from the command line by typing “python3 free\_volume.py -gui”). Alternatively, if the **use\_GUI** Boolean is True and you supply command line arguments other than the “-gui” option such as “python3 free\_volume.py -topo test1.data”, the GUI will NOT launch since the code was provided command line arguments signifying the desire to use the command line overrides. Lastly, if the **use\_GUI** Boolean is True and “python3 free\_volume.py” is typed at the command line, the GUI will be launched.

The code can also be run from a GUI directly by running **free\_volume.py** in an IDE and by setting the **use\_GUI** Boolean as True. Also, LUNAR comes with a **LUNAR.py** code, that when run allows the user to click buttons to open any of the LUNAR GUI’s, where **free\_volume.py** can be initialized from the LUNAR GUI itself. The LUNAR GUI and **free\_volume.py** GUI are shown in Figure 38. Please note that when clicking the “Save the current GUI settings as the default GUI settings” all settings will be adjusted in the **free\_volume.py** file, except for the **topofile** variable which will remain as the default this code ships with or whatever the users have manually typed in **free\_volume.py**.

A screenshot of a computer

Description automatically generated

Figure 38: LUNAR/ free\_volume.py GUI

The **LUNAR.py** and the **free\_volume.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **free\_volume.py** can be set to different values, whereas **GUI\_zoom** in **free\_volume.py** only affects the GUI that is launched from **free\_volume.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **free\_volume.py** GUI from **LUNAR.py** GUI.Additionally, that **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

Each variable will now be described one by one to explain its purpose and what changing it does. Context will be given when needed, but it is assumed that you are familiar with some things surrounding computational chemistry and MD.

Variable: **topofile** Override tags: **-topo** or **-t** Usage: **REQUIRED**

* Will read in the LAMMPS datafile that is assigned by the variable or the override tag. This file can be in any LAMMPS format, but the atom style must either be full, charge, or molecular with the style hint comment provided in the Atoms section. The only supported simulation cell is orthogonal currently.

Variable: **max\_voxel\_size** Override tags: **-voxel** or **-v** Usage: **REQUIRED**

* Is a Python float or int value that will set the maximum voxel size to discretize the simulation cell into voxels. The **max\_voxel\_size** is the maximum length of any edge on the discretized voxel.

Variable: **probe\_diameter** Override tags: **-probe-diameter** or **-pd** Usage: **REQUIRED**

* Is a Python float or int value that will set the probe size in insert it into the voxelated grid. If the probe diameter is zero the atom volume calculation is exactly the vdw volume of the atoms. Whereas if the probe diameter is greater than zero the calculation starts to approach the Positron annihilation lifetime spectroscopy (PALS) method and the probe diameter supplied to **free\_volume.py** can be thought of as the probe size of Positron annihilation lifetime spectroscopy (PALS) method.
* Additionally, if **probe\_diameter** is set to‘min-voxel’ the probe diameter will be updated to the minimum dimension of a voxel found in your system and dependent on the **max\_voxel\_size** variable.

Variable: **vdw\_method** Override tags: **-vdw-method** or **-vdw** Usage: **REQUIRED**

* Is a Python string value to tell **free\_volume.py** which vdw radii to use for setting the occupied vdw volume. The following methods are supported:
  + ‘dict’ where the used vdw radii will come from the **vdw\_radius** dictionary.
  + ‘class1’ where the used vdw radii will come from the read-in **topofile** and the **topofile** uses the 12-6 LJ potential and parameter ordering in file is **[epsilon sigma]**.
  + ‘class2’ where the used vdw radii will come from the read-in **topofile** and the **topofile** uses the 9-6 LJ potential and parameter ordering in file is **[epsilon sigma]**.

Variable: **parent\_directory** Override tags: **-dir** or **-d** Usage: **REQUIRED**

* Will set the directory where all the written files will be stored. Setting it as an empty string as such: **parent\_directory** = ‘’ or **parent\_directory** = ‘.’, will write all files to the location of **free\_volume.py**. Setting it as any path will create a path from the LUNAR directory and store the written files there. Setting **parent\_directory** to ‘topofile’ will use the path of the first file in the **files** variable to write the files to that location on your machine. The ‘topofile’ shortcut string should be the default usage for 1st-time users. Once the code finishes running a printout will show the path of where the files are written to help guide the user to that location on their machine. Directories will be made via **free\_volume.py** if they do not already exist on your machine. Additionally, another shortcut dealing with the ‘topofile’ shortcut is that relative directories can be made from the location of the **topofile** on your machine via ‘topofile/NEWDIR’, where ‘NEWDIR’ will build relative directories to the topofile path. Examples of using ‘topofile/NEWDIR’
  + **parent\_directory** = ‘topofile/NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/NEWDIR’ will be built and all outputs will be stored there.

* + **parent\_directory** = ‘topofile/../NEWDIR’ and **topofile** = ‘Furan\_Resin/files/furan.data’, where the base directory from **topofile** is ‘Furan\_Resin/files’ so a directory with the path

‘Furan\_Resin/files/../NEWDIR’ will be built and all outputs will be stored there. Where the ../ characters traverse one directory level backward.

Variable: **boundary** Override tags: **-pbc** or **-p** Usage: **REQUIRED**

* This option sets the boundary of the simulation cell for assigning voxels to atomIDs, where the vdw radii leave one side of the simulation box and enter on the opposite side of the simulation box.
* The data type assigned to the **boundary** is a string with three boundary flags with the same meaning that LAMMPS uses for boundary flags. The string must contain 3 characters each separated by whitespace. Each character can either be ‘p’ for periodic or ‘f’ for non-periodic. Each location in the string sets the x, y, or z faces of the simulation cell to that boundary.

Variable: **run\_mode** Override tags: **-run-mode** or **-run** Usage: **REQUIRED**

* Is a Python string to set the run mode and dependencies of the free volume calculation. Depending on the chosen **max\_voxel\_size** and the number of atoms and the size of the simulation cell, **free\_volume.py** can become very slow to compute the free volume. Different run modes have been provided to try to optimize the run time, but also control the number of additional Python packages needed to run the code. The following run modes are supported with a description and their dependencies:
  + ‘stl’ which runs on the standard Python library and requires: tqdm. This is the slowest-performing run mode but has the least amount of dependencies.
  + ‘numpy’ which runs on the standard Python library uses numpy vectorization and requires tqdm and numpy. This is the medium-performing run mode and has a medium amount of dependencies.
  + ‘numba’ which runs on the standard Python library and uses numba to compile numpy arrays down to machine code and runs in serial and requires: tqdm, numpy, and numba. This is one of the fastest-performing run modes and has the most amount of dependencies.
  + ‘numba-p’ which runs on the standard Python library and uses numba to compile numpy arrays down to machine code runs in parallel and requires: tqdm, numpy, and numba. This is one of the fastest-performing run modes and has the most amount of dependencies. The numba documentation does not say how it determines the number of processors it will use to parallelize the code, so it is best to assume that it will use all available processors to run the code in parallel, which means you should be cautious about running **free\_volume.py** on an HPC infrastructure in ‘numba-p’ mode, since it may try to run on all the HPC processors (this also depends on the queuing system and architecture or the HPC clusters).
  + ‘stl-dd’ which has the same meaning as ‘stl’, but uses a domain decomposition and linked cell list algorithm instead of the default algorithm. The domain decomposition and linked cell list algorithm may be quicker for all models over the ‘stl’ **run\_mode**.
  + ‘numba-dd’ which has the same meaning as ‘numba’, but uses a domain decomposition and linked cell list algorithm instead of the default algorithm. The domain decomposition and linked cell list algorithm may be quicker for some models over the ‘numba’ **run\_mode**.
  + ‘numba-ddp’ which has the same meaning as ‘numba-p’, but uses a domain decomposition and linked cell list algorithm instead of the default algorithm. The domain decomposition and linked cell list algorithm may be quicker for some models over the ‘numba-p’ run\_mode.
  + ‘CUDA’ which runs on the standard Python library and uses numba to compile numpy arrays down to machine code runs in parallel on the GPU and requires: tqdm, numpy, and numba. This is one of the fastest-performing run modes and has the most amount of dependencies. This mode requires that you have a CUDA-enabled GPU (i.e. a NVIDIA GPU) and the number of threads per block for atom operations and voxel operations are defined via the Python variables **CUDA\_threads\_per\_block\_atoms** and **CUDA\_threads\_per\_block\_voxels**, respectively.
  + ‘CUDA-dd’ which has the same meaning as ‘CUDA’, but uses a domain decomposition and linked cell list algorithm instead of the default algorithm. The domain decomposition and linked cell list algorithm may be quicker for some models over the ‘CUDA’ **run\_mode**.
  + ‘dd’ variants define the subdomain size from the following equation:
    - subdomain = round(2.1\*max(system\_vdw\_radii), 2) + 2\*max\_voxel\_size + probe\_diameter
    - restrictions that the simulation cell must at least be 3\*subdomain in each direction to use any of the ‘dd’ variants.
* A test system using a mixed and equilibrated EPON 862 system with 21,060 atoms, , a p p p boundary, a probe diameter of 1.06, and a voxel size of 0.5 and 0.25 was used to develop an understanding of the run times for each run mode to get a feel for the relative performance of each mode:

A graph and diagram of a graph

Description automatically generated with medium confidence

* These test where performed on an 8-core Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz and an NVIDIA GeForce RTX 2070 Super. The ‘numba-p’ and ‘numba-ddp’ run modes utilized all 8-cores and the ‘CUDA’ and ‘CUDA-dd’ utilized 128 threads per block for atom operations and 256 threads per block for voxel operations.
* In general, the numba and CUDA run modes are the quickest, where on a larger 0.5 voxel size ‘numba-ddp’ is the quickest, and on the smaller voxel size ‘CUDA-dd’ is the quickest. This is due to the overhead cost of copying data back and forth from the CPU to the GPU and the difference in concurrent operations between the CPU and GPU, where it makes most sense when many calculations are needed. If your CPU has less than 8-cores you may find the CUDA modes quicker for most cases, or if your CPU has more than 8-cores you may find the numba modes to be quicker. A few performance benchmark tests must be performed on your machine to determine the quickest run mode. Additionally, for the CUDA modes, the threads per block for atom operations and the threads per block for voxel operations can significantly change the performance.

Variable: **CUDA\_threads\_per\_block\_atoms** Override tags: **-cuda-atoms** or **-ca** Usage: **REQUIRED**

* This option is only meant for either the ‘CUDA’ or ‘CUDA-dd’ **run\_mode** and sets the number of threads per block for atom operations for the GPU parallelization. The blocks per grid are then computed by:
  + blocks\_per\_grid\_atoms = math.ceil( natoms / CUDA\_threads\_per\_block\_atoms )
* The threads per block should be in doubling multiples of 8 (i.e. 8, 16, 32, 64, 128, 256, 512, and 1024), where some GPUs may not be able to go up to 1024.
* On the developer's GPU and a few test systems was noted that a good starting number for **CUDA\_threads\_per\_block\_atoms** is natoms/150 for example:
  + System: 1200 atoms
  + CUDA\_threads\_per\_block\_atoms = 1200/150 = 8

Variable: **CUDA\_threads\_per\_block\_voxels** Override tags: **-cuda-voxels** or **-cv** Usage: **REQUIRED**

* This option is only meant for either the ‘CUDA’ or ‘CUDA-dd’ **run\_mode** and sets the number of threads per block for voxel operations such as generating voxels and computing the free volume voxel connectivity for the GPU parallelization. The blocks per grid are then computed by:
  + blocks\_per\_grid\_voxels = math.ceil( natoms / CUDA\_threads\_per\_block\_voxels )
* The threads per block should be in doubling multiples of 8 (i.e. 8, 16, 32, 64, 128, 256, 512, and 1024), where some GPUs may not be able to go up to 1024. Additionally, for voxel operations if

**CUDA\_threads\_per\_block\_voxels** is set to zero it will tell **free\_volume.py** to run thevoxel operations in the CPU instead of the GPU.

* On the developer's GPU and a few test systems was noted that a good starting number for **CUDA\_threads\_per\_block\_voxels** is about double that of **CUDA\_threads\_per\_block\_atoms** if the **max\_voxel\_size** is less than 0.5 Å.

Variable: **files2write** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as representative names of the different files that **free\_volume.py** can write and the values are Booleans (True or False), to control whether the code will write that file or not. The following are the supported keys and what type of file it controls:
  + ‘write\_atoms\_free’ which is a **\*.data** file with the voxels that are assigned to the atoms and to the free volume, where different atomTypeIDs are provided to visualize the atom volume and free volume together in software like OVITO.
  + ‘write\_bonds\_free’ which is a **\*.data** file with the voxels that are assigned to the free volume and the atoms and bonds are transferred from the read in **topofile**, where different atomTypeIDs are provided to visualize the atoms/bonds and free volume together in software like OVITO.
  + ‘write\_atoms\_only’ which is a **\*.data** file with the voxels that are assigned to the atoms only, where different atomTypeIDs are provided for the element types in the system to visualize the atom volume in software like OVITO.
  + ‘write\_free\_only’ which is a **\*.data** file with the voxels that are assigned to the free volume only to visualize the free volume in software like OVITO.
  + ‘write\_all\_voxels’ which is a **\*.data** file with all the unassigned voxels to get an understanding of how discretized the system was based on the **max\_voxel\_size** variable.
  + ‘write\_spat\_dis-x’ which is a **\*spatial\_distribution\_direction\_x.csv** file of the free volume grouped spatial distribution in the X-directions.
  + ‘write\_spat\_dis-y’ which is a **\*spatial\_distribution\_direction\_y.csv** file of the free volume grouped spatial distribution in the Y-directions.
  + ‘write\_spat\_dis-z’ which is a **\*spatial\_distribution\_direction\_z.csv** file of the free volume grouped spatial distribution in the Z-directions.

Variable: **compute\_free\_volume\_distributions** Override tags: **-free-vol-dist** or **-fve** Usage: **REQUIRED**

* Is a Python Boolean variable (True or False) to compute the free volume voxel connectivity based on a graph theory approach. For a voxel to be connected to another it must be less than a **max\_voxel\_size** distance away in either the X, Y, or Z-directions, which means a voxel that is “fully buried” in other voxels will have 26 first neighbors. Once the first neighbor connectivity is determined the voxels are determined to be connected to a larger number of neighbors using a breadth-first search traversal. This analysis works decently well but tends to be slow and the results usually don’t further the understanding of the tested systems. Due to this, it is often recommended that this option is set as False and used only if a need seems to arise to understand the free volume connectivity. In most cases, the free volume is usually one large, connected volume, which is why **compute\_free\_volume\_distributions** should usually be False since it is slow, and that information can almost already be assumed to be the case.

Variable: **mass\_map** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as elemental symbols and the corresponding values being a list of masses associated with that element type. The purpose of the **mass\_map** dictionary is to set the element type of each atom type in the **.data** file, which will be identified via the mass of each atom type.

Variable: **vdw\_radius** Override tags: N/AUsage: **REQUIRED**

* Is a Python dictionary with keys set as elemental symbols and the corresponding values the vdw radii of that elemental type. The **vdw\_radius** dictionary is used to set the element to vdw radii mapping such that voxels can be assigned to atomIDs using the vdw radii as a maximum distance cut-off.

Once the code is run there will be a variety of printouts, which are important to look through. The printouts highlight key information about your molecular system and will tell you if there are any WARNINGS or ERRORS.

## Outputs

Please refer to the **files2write** section to see the type of outputs **free\_volume.py** generates and how to control whether they are written or not. Additionally, each time the code is run the following outputs will be generated using the basename from the **topofile**:

* **\*.log.lunar** which has stored the printouts of **free\_volume.py**

# Code: log\_analysis.py

## Purpose

The purpose of **log\_analysis.py** is to have a tool to analyze the LAMMPS output log file (will just be discussed as “the log file”) which can be customized using the LAMMPS *“thermo\_style custom”* and LAMMPS *“log”* log commands. Additionally, dynamic LAMMPS variables, computes, and fixes can have information logged to the LAMMPS log file. Typically, the data in the log file needs to be plotted and analyzed to compute averaged properties or slopes of portions of the logged data. This can be a tedious task because parsing the log file can be challenging, especially when there are more than one section in the log file. Typically, custom scripts are built by users of LAMMPS to read and plot the logged data, however they usually only can be applied to a limited number of analysis methods. The goal of **log\_analysis.py** is to have a general purpose LAMMPS log file plotting tool with enough analysis methods to make it useful for a large variety of analysis types.

## Code Variables and Execution

Unlike the rest of the LUNAR codes the **log\_analysis.py** code only supports a GUI run method and that is because plotting and analysis is an interactive process where the most user friendly way for interaction seems to be a GUI. A log file can be loaded, data plotted, and an arbitrary number of analysis’s can be performed all through the GUI. Additionally, **log\_analysis.py** script can have “modes” defined that will automatically update the GUI settings to quickly “load” different analysis methods to help speed up large batch processing. The **log\_analysis.py** code can read in all sections in the log file or certain portions of the log file at a time. The **LUNAR.py** GUI, **log\_analysis.py** GUI, and an example plot are shown in Figure 39.

The **log\_analysis.py** GUI can be initialized by running the **log\_analysis.py** script at the command line or in an IDE. Alternatively, you a run the **LUNAR.py** GUI from the command line or IDE and then click the “log\_analysis” button to load the **log\_analysis.py** GUI. Anaconda users utilizing the Spyder IDE may have to run *“%matplotlib qt”* at the console to activate the interactive plotting options. Otherwise, the plots will show up in the “Plots” tab of the IDE.

A screenshot of a computer

Description automatically generated

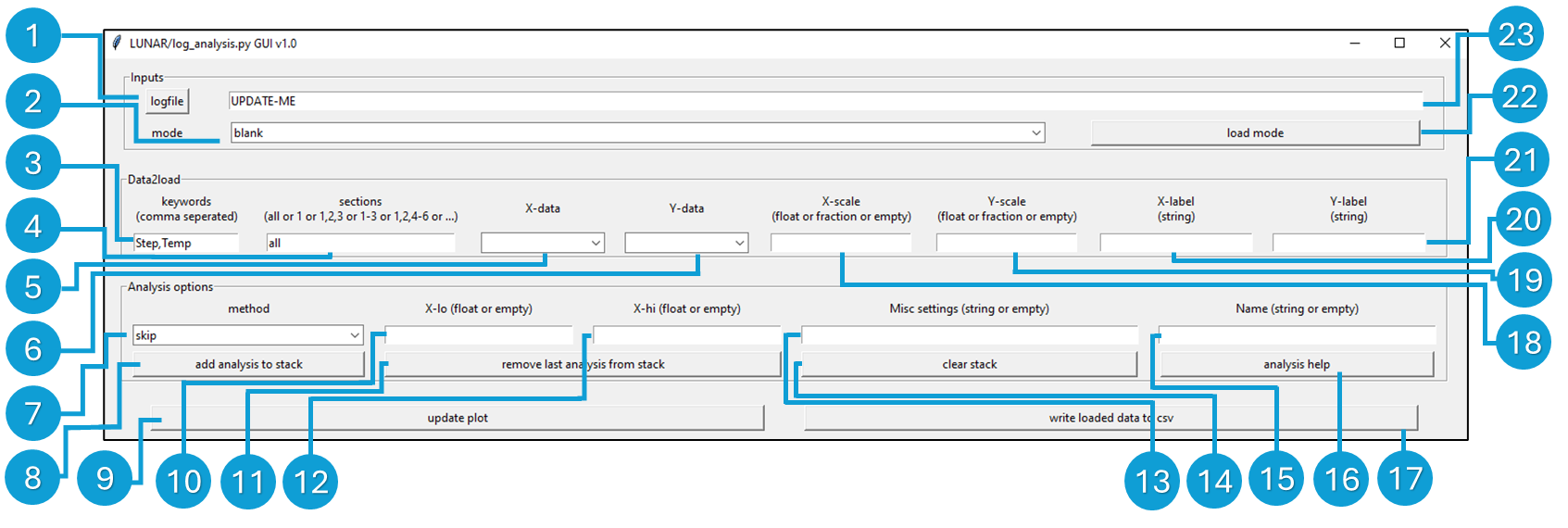
A screenshot of a computer

Description automatically generated

Figure 39: LUNAR/ log\_analysis.py GUI

The **LUNAR.py** and the **log\_analysis.py** files both have a **GUI\_zoom** variable to control the size of the font and the spacing of the widgets. The purpose of **GUI\_zoom** is to allow users to control the GUI size for two reasons 1) to manage screen space which is dependent on how the user has their OS display settings set and 2) to provide a level of accessibility for those in need of large fonts. All GUI’s are initialized with the default font type and font size, such that a **GUI\_zoom = 100** means use default GUI font size, where **GUI\_zoom = 75** or **GUI\_zoom = 125** means decrease font size and widget spacing by 25% or increase the default font size and widget spacing by 25% respectively. The **GUI\_zoom** variable found in both **LUNAR.py** and **log\_analysis.py** can be set to different values, whereas **GUI\_zoom** in **log\_analysis.py** only affects the GUI that is launched from **log\_analysis.py**. Whereasthe **GUI\_zoom** variable found in **LUNAR.py** affects the master LUNAR GUI but is also used when launching **log\_analysis.py** GUI from **LUNAR.py** GUI.Additionally, the **LUNAR.py** code can be run from the command line with a command line integer argument of 0 to 200 to change the **GUI\_zoom** variable, where if there is no additional information is provided at the command line, the **GUI\_zoom** variable found in **LUNAR.py** will be enforced.

The use of the GUI will now be discussed below:



**Inputs frame of GUI**

#1 Button: **logfile**

* Will open a file dialog box to select a log file to load. After the file is clicked the GUI will read the logfile based on the **keywords** (#3 entry) and **sections** (#4 entry) entries discussed below, to find the columns in the log file so that **X-data** (#5 Combo box) and **Y-data** (#6 Combo box) can be updated with the columns that are available to plot.
* The loaded filename will appear in entry #23. Please note that you can manually type the filename in entry #23 as well, but this will not automatically update the **X-data** (#5 Combo box) and **Y-data** (#6 Combo box). Thus, the only way you should load a file is to use #1 button to click on a logfile to load.

#2 Combo box: **modes**

* The GUI is dynamic where you can manually adjust each option to generate an analysis method. However, **log\_analysis.py** also allows users to build pre-defined settings to automatically load and update the GUI. This makes performing analysis’s much quicker, since you can automatically adjust all settings with a click of a button.
* The #2 Combo box: **modes** allows users to select a pre-defined set of options. To load the predefined set of options you must click #22 Botton: **load mode**, which will then load the predefined set of options into the GUI.
* The pre-defined options are setup in the **log\_analysis.py** file via Python dictionaries, where the **\*\_mode = {}** naming convention has been established to setup the options to automatically load (as shown below).
* A lot of different modes have been built and can be used as examples to define more loadable modes. The **blank\_mode = {}** dictionary is commented in the **log\_analysis.py** file to provide guidance on building your own pre-defined mode.
* NOTE that some of the defined modes load **X-data** (#5 Combo box) and **Y-data** (#6 Combo box) names that are unique to certain LAMMPS run’s since they load in LAMMPS variable names that are custom to certain LAMMPS scripts. This means that you may need to update those names to fit your LAMMPS simulation to utilize the mode directly.

A screen shot of a computer

Description automatically generated

#22 Combo box: **load mode**

* Will load the current mode defined in the **mode** Combo Box, by setting all options in the GUI according to the mode settings setup in the **\*\_mode = {}** dictionary found in **log\_analysis.py**.

**Data2load frame of GUI**

#3 entry: **keywords**

* Names of a column or columns in the logfile. This information is used to parse the logfile. Minimally one keyword must be specified to properly parse the logfile, however more can be added by comma separating the keywords.

#4 entry: **sections**

* Depending on the LAMMPS simulation the logfile maybe written in “sections” of output data. The logfile reader will allow users to read in whatever desired section(s) they wish. The function that reads the logfile will label the first section as 1, the second section as 2, all the way to the Nth section.
* The inputs to the sections entry is based on the Windows print dialog boxes for selecting which pages to print. Where the “-“ and “,” characters can be used to string together different combinations of sections. The “-“ character will set the sections to load as a linear series starting from the left value to the right value and the “,” character will string together arbitrary sections. The following are some examples of valid inputs to the **sections** entry:
  + all (will read all section of the logfile)
  + 1 (will only read the first section)
  + 1,2,3 (will read the first, second, and third sections)
  + 1-3 (will read the first, second, and third sections)
  + 1,2,4-6 (will read the first, second, fourth, fifth, and sixth sections)

# 5 Combo Box: **X-data**

* After the logfile has been loaded via the **logfile** button, where the sections of the logfile have been defined from the **sections** entry, the **X-data** Combo box will be populated with all available columns from the loaded data. You can select the data you with to plot as the x-data.

# 6 Combo Box: **Y-data**

* After the logfile has been loaded via the **logfile** button, where the sections of the logfile have been defined from the **sections** entry, the **Y-data** Combo box will be populated with all available columns from the loaded data. You can select the data you with to plot as the y-data.

# 18 entry: **X-scale**

* There are cases where it is desired to “scale” the x-data by some value to convert from one units to another. The **X-scale** entry allows users to add a value that is multiplied to every data point in the data defined by **X-data** Combo box. If the **X-scale** entry is left empty the **X-data** will be left as is. However, if a float or fraction is supplied every data point in the **X-data** series will be multiplied by the float or fraction. The following are a few examples:
  + 0.5 (every data point in the **X-data** series will be multiplied by 0.5)
  + ½ (every data point in the **X-data** series will be multiplied by ½)

# 19 entry: **Y-scale**

* There are cases where it is desired to “scale” the y-data by some value to convert from one units to another. The **Y-scale** entry allows users to add a value that is multiplied to every data point in the data defined by **Y-data** Combo box. If the **Y-scale** entry is left empty the **Y-data** will be left as is. However, if a float or fraction is supplied every data point in the **Y-data** series will be multiplied by the float or fraction. The following are a few examples:
  + 0.5 (every data point in the **Y-data** series will be multiplied by 0.5)
  + ½ (every data point in the **Y-data** series will be multiplied by ½)

# 20 entry: **X-label**

* Will set the x-label in the plot. If left empty the column name from the logfile that is loaded in the **X-data** Combo box will be used to provide an x-label in the plot.

# 21 entry: **Y-label**

* Will set the x-label in the plot. If left empty the column name from the logfile that is loaded in the **Y-data** Combo box will be used to provide an x-label in the plot.

**Analysis options frame of GUI**

The **analysis options** frame entries and Combo Box are interpreted in rows, where each row defines aspects of the analysis method defined in the **method** column.

# 7 Combo Box: **method** column

* Sets the **method** of analysis or the row. The following methods are available:
  + **average** (will find the average y-value of the data in a given x-range)
  + **moving average** (will find the moving average of data in a given x-range)
  + **linear regression** (will find the slope of section of data in a given x-range)
  + **cursor** (will place a data point in the plot)
  + **skip** (the **X-lo**, **X-hi**, **Misc settings**, and **Name** columns will be ignored)
* A visualization of the different methods is provided below.

A screenshot of a computer

Description automatically generated

# 10 Entry: **X-lo** column

* Sets the lower range of the X-data series for which the analysis will be applied.

# 12 Entry: **X-hi** column

* Sets the higher range of the X-data series for which the analysis will be applied.

# 13 Entry: **Misc settings** column

* Inputs to override defaults of certain methods. If you click #16 Button **analysis help** a help page will pop-up defining the miscellaneous options that can be applied for each **method**.

A close-up of a computer screen

Description automatically generated

# 15 Entry: **Name** column

* Will set the name of the analysis in the legend of the plot. If left empty a default name is given as analysis-N, where N is the index of the row.

# 8 Button: **add analysis to stack**

* If clicked a new row will be added to the stack to define another analysis.

# 11 Button: **remove last analysis from stack**

* If clicked the last analysis row added will be cleared and the **method** column will be set to **skip**. Effectively removing the analysis from the stack.

# 14 Button: **clear stack**

* If clicked every analysis row added will be cleared and the **method** column will be set to **skip**. Effectively removing all defined analysis’s from the stack.

# 16 Button: **analysis help**

* Will open a pop-up describe the analysis **method**s usage of the **Misc settings** column.

**Bottom buttons of GUI**

# 9 Button: **update plot**

* Will apply every option in the GUI starting with re-reading the logfile based on the **keywords** and **sections** entries.
* Will set the x and y-data based on the **X-data** and **Y-data** Combo Boxes.
* Will plot the **X-data** and **Y-data** with the **X-scale** and **Y-scale** applied and the **X-label** and **Y-label** defining the axis titles.
* Will apply every analysis in the **Analysis options** frame and plot the analysis.
* Will save a figure based on the basename of the logfile using the **\*.jpeg** extension.
* Will write a text file based on the basename of the logfile using the **\*.log.lunar** extension, which details the analysis results.

# 17 Button: **write loaded data to csv**

* Will write all loaded data to a comma separate variable file based on the basename of the logfile with the **\*.csv** extension. This will allow users to plot data in excel if they wish to further their analysis or generate plots for publication purposes.

## Outputs

The code will write files described above in the **Bottom buttons of GUI** section.

# References

[1] A. P. Thompson, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications,* vol. 271, no. 2022, pp. 108-171, 2022, doi: 10.1016/j.cpc.2021.108171.

[2] A. I. Jewett, D. Stelter, J. Lambert, S. M. Saladi, O. M. Roscioni, M. Ricci, L. Autin, M. Maritan, S. M. Bashusqeh, T. Keyes, R. T. Dame, J. E. Shea, G. J. Jensen, and D. S. Goodsell, "Moltemplate: A Tool for Coarse-Grained Modeling of Complex Biological Matter and Soft Condensed Matter Physics," *J Mol Biol,* vol. 433, no. 11, p. 166841, May 28 2021, doi: 10.1016/j.jmb.2021.166841.

[3] M. A. Bone, B. J. Howlin, I. Hamerton, and T. Macquart, "AutoMapper: A python tool for accelerating the polymer bonding workflow in LAMMPS," *Computational Materials Science,* vol. 205, 2022, doi: 10.1016/j.commatsci.2022.111204.

[4] "Open force field." <https://openforcefield.org/software/> (accessed.

[5] "LigParGen." [Online]. Available: <http://zarbi.chem.yale.edu/ligpargen/>.

[6] "msi2lmp." [Online]. Available: <https://www.mankier.com/1/msi2lmp>.

[7] "Molecular Simulation Design Framework." [Online]. Available: <https://mbuild.mosdef.org/en/stable/>.

[8] "MedeA Software." <https://www.materialsdesign.com/medea-software> (accessed 2023).

[9] P. J. i. t. Veld, "Temperature-dependent elasticity of a semicrystalline interphase composed of freely rotating chains," *Macromolecules,* vol. 36, no. 19, pp. 7358-7365, 2003.

[10] "msi2lmp - Man Page Converter for Materials Studio files to LAMMPS." <https://www.mankier.com/1/msi2lmp> (accessed.

[11] M. Bone. "Moldyn Atlas." <https://www.molydyn.com/atlas> (accessed.

[12] J. Gasteiger, "Gasteiger, Johann, and Mario Marsili. "Iterative partial equalization of orbital electronegativity—a rapid access to atomic charges," *Tetrahedron,* vol. 36, no. 22, pp. 3219-3228, 1980.

[13] J. R. Gissinger, B. D. Jensen, and K. E. Wise, "Chemical reactions in classical molecular dynamics," *Polymer,* vol. 128, pp. 211-217, Oct 16 2017, doi: 10.1016/j.polymer.2017.09.038.

[14] J. R. Gissinger, B. D. Jensen, and K. E. Wise, "REACTER: A heuristic method for reactive molecular dynamics," *Macromolecules,* vol. 53, no. 22, pp. 9953-9961, 2020, doi: 10.1021/acs.macromol.0c02012.

[15] J. M. Martinez and L. Martinez, "Packing optimization for automated generation of complex system's initial configurations for molecular dynamics and docking," *J Comput Chem,* vol. 24, no. 7, pp. 819-25, May 2003, doi: 10.1002/jcc.10216.

[16] L. Martinez, R. Andrade, E. G. Birgin, and J. M. Martinez, "PACKMOL: a package for building initial configurations for molecular dynamics simulations," *J Comput Chem,* vol. 30, no. 13, pp. 2157-64, Oct 2009, doi: 10.1002/jcc.21224.