Moltemplate Manual

Andrew Jewett, Shea Lab
University of California Santa Barbara
Santa Barbara, CA 93106
jewett.aij@gmail.com

October 22, 2012

Contents

1 Introduction

Moltemplate is a cross-platform text-based molecule builder for LAMMPS. It is typically used for building coarse-grained toy molecular models. Moltemplate users have access to (nearly) all of the standard and non-standard (custom, user-created) force-field and features available in LAMMPS.

(Although optimized for LAMMPS, moltemplate is a general text manipulation tool which, in principle, could be used to generate topology and force-field files for other simulation programs. Please email **jewett.aij@gmail.com** *if you want to attempt this.)*

A file format has been created to store molecule definitions (the LAMMPS-template format, "LT"). Typical ".LT" files contain atom coordinates, topology data (bonds), LAMMPS force-field data, and other LAMMPS settings (such as group definitions, fixes, and user-defined input files) for a type of molecule (or a molecular subunit). Molecules can be copied, combined, and linked together to define new molecules. (These can be used to define larger molecules. Unlimited levels of object composition, nesting, and inheritance are supported.) Once built, individual molecules and subunits can be customized (atoms and bonds, and subunits can be moved, deleted and replaced).

Moltemplate requires the Bourne-shell, and a recent version of python (2.7 or 3.0 or higher), and can run on OS X, linux, or windows (if a suitable shell environment has been installed).

1.1 Converting *LT files* to LAMMPS input/data files

The moltemplate.sh program converts LT-files (which contain molecule definitions) into complete LAMMPS input-scripts and data-files:

```
moltemplate.sh -atomstyle "full" system.lt
```

or

```
moltemplate.sh -xyz coords.xyz -atomstyle "full" system.lt
```

In the first example, the coordinates of the atoms in the system are built from commands inside the "system.lt" file. In the second example coordinates for the atoms are read from an XYZ-file (PDB-files are also supported). (The "full" atom style was used in this example, but other LAMMPS atom styles are supported, including hybrid styles.)

Either of these commands will construct a LAMMPS data file and a LAMMPS input script (and possibly one or more auxiliary input files), which can be directly run in LAMMPS with minimal editing.

1.2 Converting LAMMPS input/data files to *LT files*

Existing LAMMPS input/data files can be converted into ".LT" files using the "ltemplify.py" utility. (See appendix B.)

Additional tools

The VMD topotools plugin [1] is useful for converting PDB files into LAMMPS format. These files can then be converted to "LT" format using the "ltemplify.py" utility. VMD [2] and topotools are also useful for visualizing the data files created by moltemplate.sh. (Documentation for doing this is included in the *online examples* discussed below.)

The PACKMOL [3] program is useful for generating coordinates of dense heterogeneous mixtures of molecules, which can be read by moltemplate. (The VMD "solvate" plugin may also be helpful.)

Online examples

This manual explains how to use moltemplate.sh to build LAMMPS files from scratch, but it does not discuss how to run LAMMPS or how to visualize the results.

This manual assumes users have some basic familiarity with LAMMPS.

For users who are not familiar with LAMMPS, several complete, working examples (with images and readme files) are available online (at `http://moltemplate.org`) which can be downloaded and modified. These examples are a good starting point for learning LAMMPS and moltemplate.

License

2 Installation

If you don't already have moltemplate, the first step of the installation is to download and unpack the moltemplate archive. The most up-to-date version of moltemplate can always be downloaded at: `http://www.moltemplate.org` and unpacked using:

```
tar xzf moltemplate_2012-3-31.tar.gz
```

(The date will vary from version to version.)

Alternately, if you obtained moltemplate bundled with LAMMPS, then the *"moltemplate"* directory will probably be located in the *"tools"* subdirectory of your lammps installation.

If necessary, move the *"moltemplate"* directory to its desired location. (For the sake of this example, let's assume you move it to: *"$HOME/moltemplate"*.)

The *"moltemplate.sh"* script and the python scripts it invokes are located in the *"src"* subdirectory. You should update your PATH environment variable to include this directory. You also need to set your MOLTEMPLATE_PATH environment variable to point to the *"common"* subdirectory. (Force fields and commonly used molecules will eventually be located here.)

If you use the bash shell, typically you would edit your ∼/.bash_profile, ∼/.bashrc, or ∼/.profile files to contain the following lines:

```
export PATH="$PATH:$HOME/moltemplate/src"
export MOLTEMPLATE_PATH="$HOME/moltemplate/common"
```

If you use the tcsh shell, typically you would edit your ∼/.login, ∼/.cshrc, or ∼/.tcshrc files to contain the following lines:

```
setenv PATH "$PATH:$HOME/moltemplate/src"
setenv MOLTEMPLATE_PATH "$HOME/moltemplate/common"
```

3 Quick reference *(skip on first reading)*

Note: New users should skip to section 4

3.1 Moltemplate commands

command	meaning
MolType **{** *content ...* **}**	Define a new type of molecule (or namespace) named *MolType*. The text enclosed in curly brackets (*content*) typically contains multiple write(), write_once() commands to define Atoms, Bonds, Angles, Coeffs, etc... *(If that molecule type exists already, then this will append additional **content** to its definition.)* **new** and **delete** commands can be used to create or delete molecular subunits *within* this molecule. (See the *SPCE*, *2bead*, and *Butane* molecules, and the *TraPPE* namespace defined in sections 4.1, 6.1, 11.8, & 11.4.1.
mol_name = **new** *MolType*	Create (instantiate) a copy of a molecule of type *MolType* and name it *mol_name*. (See section 4.1.)
mol_name = **new** *MolType.xform()*	Create a copy of a molecule and apply coordinate transformation *xform()* to its coordinates. (See sections 4.2 and 3.3.)
molecules = **new** *MolType* [*N*].*xform()*	Create *N* copies of a molecule of type *MolType* and name them *molecules[0]*, *molecules[1]*, *molecules[2]*... Coordinates in each successive copy are cumulatively transformed according to *xform()*. (See sections 4.2, 7.1 and 3.3.) Multidimensional arrays are also allowed. (See section 9.)
molecules = **new** *MolType.xform1()* **[*N*]**.*xform2()*	Apply coordinate transformations (*xform1()* to *MolType*, before making *N* copies of it while cumulatiely applying *xform2()*. (See section 7.1 and 7.3.)
molecules = **new** **random**([*M1.xf1()*, *M2.xf2()*, *M3.xf2()*,...], [p_1, p_2, p_3,...], *seed*) **[*N*]**.*xform()*	Generate an array of *N* molecules randomly selected from *M1,M2,M3,...* with probabilities p_1, p_2, p_3..., using (optional) initial coordinate transformations *xf1(), xf2(), xf3, ...*, and applying transformation *xform()* cumulatively thereafter. This also works with multi-dimensional arrays. (See sections 7.4 and 9.2.)
NewMol = *OldMol*	Create a new molecule **type** based on an existing molecule type. Additional atoms (or bonds, etc...) can be added later to the new molecule using NewMol {*more content...*}. (See section 10.)
NewMol = *OldMol.xform()*	Create a new molecule **type** based on an existing molecule type, and apply coordinate transformation *xform()* to it. (See section 10. *This feature is experimental as of 2012-9-28.*)

NewMol **inherits** *Mol1 Mol2 ...* {
 	
additional content ...
 	
}
	Create a new molecule **type** based on multiple existing molecule types. Atom types, bond types, angle types (etc) which are defined in *Mol1*, or *Mol2*, ... are available inside the new molecule. *Additional content* (including more *write()* or *write_once()* or *new* commands) follows within the curly brackets. (See sections 10, 11.8, and 11.8.1)

MolType.xform()	Apply the coordinate transform *xform()* to the coordinates of the atoms in all molecules of type *MolType*. (See section 10. *This feature is experimental as of 2012-9-28.*)
molecule.xform()	Apply the coordinate transform *xform()* to the coordinates in *molecule*. (Here *molecule* refers to a specific instance or copy of a particular molecule type. See sections 8 and 4.2.)
molecules[*range*].*xform()*	Apply the coordinate transform *xform()* to the coordinates of molecules specified by *molecule*[*range*]. (This also works for multidimensional arrays. See sections 7.5 and 8.)
delete *molecule*	Delete the *molecule* copy. (This command can appear inside a molecule's definition to delete a specific molecular subunit within a molecule. In that case, it will be carried out in every copy of that molecule type. **delete** can also be used to delete specific atoms, bonds, angles, dihedrals, and improper interactions.) See section 8.3.
delete *molecules*[*range*]	Delete a range of molecules specified by *molecule*[*range*]. (This also works for multidimensional arrays. See sections 8.3 and 9.4.)

write_once('*file*') {
text ...
}
	Write the text enclosed in curly brackets {...} to file *file*. The *text* can contain @variables which are replaced by integers. (See sections 5.1 and 5.2.)

write('*file*') {
text ...
}
	Write the text enclosed in curly brackets {...} to file *file*. *This is done every time a new copy of this molecule is created using the "new" command.* The *text* can contain either @variables or $variables which will be replaced by integers. (See sections 5.1 and 5.2.)

Note: *file* names beginning with "Data " or "In " (such as "Data Atoms" or "In Settings") are inserted into the relevant section of the LAMMPS data file or input script. (See section 5.4.)	
include *file*	Insert the contents of file *file* here. (Quotes optional.)
import *file*	Insert the contents of file *file* here. This command prevents circular inclusions and is safer to use.

using namespace *X*	This enables you to refer to any of the molecule types, defined within a **namespace** object (*X* in this example), *without* needing to refer to these objects by their full path. (Unfortunately, atom types, or bond, angle, dihedral, or improper types must still be referred to explicitly, by their full path.) Note: a "**namespace** object" is any object which lacks any *write()* or *new* commands. (In other words, namespace objects define atom types and/or molecule types without creating any atoms or molecules. See section 11.7.)
category *$catname*($i_0$, Δ) or **category** *@catname*(i_0, Δ)	Create a new variable category. See section C.2 for details.
create_var { *variable* }	Create a variable locally (but do not write it to a file). (Typically *$mol* is the *variable*. See section 6.1.1.)

3.2 Common $ and @ variables

variable type	meaning
$atom:*name*	A unique ID number assigned to atom *name* in this molecule. (Note: The *:name* suffix can be omitted if the molecule in which this variable appears only contains a single atom.)
@atom:*type*	A number which indicates an atom's *type* (typically used to lookup pair interactions.)
$bond:*name*	A unique ID number assigned to bond *name* (Note: The *:name* suffix can be omitted if the molecule in which this variable appears only contains a single bond.)
@bond:*type*	A number which indicates a bond's *type*
$angle:*name*	A unique ID number assigned to angle *name* (Note: The *:name* suffix can be omitted if the molecule in which this variable appears only contains a single angle interaction.)
@angle:*type*	A number which indicates an angle's *type*
$dihedral:*name*	A unique ID number assigned to dihedral *name* (Note: The *:name* suffix can be omitted if the molecule in which this variable appears only contains a single dihedral-angle interaction.)
@dihedral:*type*	A number which indicates a dihedral's *type*
$improper:*name*	A unique ID number assigned to improper *name* (Note: The *:name* suffix can be omitted if the molecule in which this variable appears only contains a single improper interaction.)
@improper:*type*	A number which indicates an impropers's *type*
$*mol* or $*mol:.*	This variable refers to the ID number of *this* molecule object. (See section 4.1. Note: *"$mol"* is shorthand for *"$mol:."*)
$*mol:...*	The ID number assigned to the molecule to which this object belongs (if applicable). See sections 6.1.1, 11.6 and appendix F.
$*catname*:**query**()	Query the current value of the counter in category $*catname* (without incrementing it. This is useful for counting the number of atoms, bonds, angles, molecules, etc... created so far.)
@*catname*:**query**()	Query the current value of the counter in category @*catname* (without incrementing it. This is useful for counting the number of atom types, bond types, angle types, etc... declared so far.)

See section 5.2 for details.

3.3 Coordinate transformations

suffix	**meaning**
.move(x,y,z)	Add numbers *(x,y,z)* to the coordinates of every atom
.rot(θ, x, y, z)	Rotate atom coordinates by angle θ around axis *(x,y,z)* passing through the origin. (Dipole directions are also rotated.)
.rot($\theta, x, y, z, x_0, y_0, z_0$)	Rotate atom coordinates by angle θ around axis pointing in the direction *(x,y,z)*, passing through the point (x_0, y_0, z_0). (This point will be a *fixed point*. Dipole directions are also rotated.)
.scale(ratio)	Multiply all atomic coordinates by *ratio*. (***Important:*** *The scale() command does not update force-field parameters such as atomic radii or bond-lengths. Dipole magnitudes are affected.)*
.scale(x_r, y_r, z_r)	Multiply *x, y, z* coordinates by x_r, y_r, z_r, respectively
.scale(ratio,x_0, y_0, z_0) or *.scale($x_r, y_r, z_r, x_0, y_0, z_0$)*	You can supply 3 optional additional arguments x_0, y_0, z_0 which specify the point around which you want the scaling to occur. (This point will be a *fixed point*. Of omitted, the origin is used.)
Note: **Multiple transformations can be chained together into a compound operation.** (For example: "$.scale(2.0).rotate(45.2, 1, 0, 0).move(25.0, 0, 0)$") These are evaluated from left-to-right. (See section 7.1.)	

3.4 moltemplate.sh command line arguments:

argument	**meaning**
-atomstyle *style*	Inform moltemplate which atom_style you are using. (*style* is "full" by default). Other styles like "molecular" or "hybrid full dipole" are supported. For custom atom styles, you can also specify the list of column names manually (enclosed in quotes). For example: *-atomstyle "molid x y z atomid atomtype mux muy muz"*
-pdb coords.pdb	Read all of the atomic coordinates from an external PDB file (Periodic boundary conditions are also read, if present. See section 4.2.)
-xyz coords.xyz	Read all of the atomic coordinates from an external XYZ file (See section 4.2.)
-a '*variable value*'	Assign *variable* to *value*. (The *variable* should begin with either a @ or a $ character. Quotes and a space separator are required. See appendix C.1.)
-a bindings_file'	The variables in column 1 of *bindings_file* (which is a text file) will be assigned to the values in column 2 of that file. (This is useful when there are many variable assignments to make. See appendix C.1.)
-b '*variable value*' *or* -b *bindings_file*	Assign variables to values. Unlike assignments made with "-a", assignments made using "-b" are non-exclusive. (They may overlap with other variables in the same category. See appendix C.1.)
-overlay-bonds -overlay-angles -overlay-dihedrals -overlay-impropers	By default moltemplate overwrites duplicate bonded interactions which involve the same set of atoms. These flags disable that behavior. This can be useful when you want to superimpose multiple angular or dihedral forces on the same set of atoms (eg. to enable more complex force fields).
-nocheck	Do *not* check for common LAMMPS/moltemplate syntax errors. (This might be useful when using moltemplate with simulation software other than LAMMPS, *or* to build systems which need new non-standard LAMMPS features.)

4 Introductory tutorial

Summary

Moltemplate is a very simple text generator (wrapper) which repetitively copies short text fragments into one (or more) files and keeps track of various kinds of counters. Moltemplate is (intentionally) ignorant about LAMMPS and molecular dynamics in general. It is the user's responsibility to understand LAMMPS syntax and write LT files which obey it. Again, for users who are new to LAMMPS, the easiest way to do this is to modify an existing example.

4.1 Simulating a box of water using moltemplate and LAMMPS

Figure 1: Coordinates of a single water molecule in our example. (Atomic radii not to scale.)

Here we show an example of a lammps-template file for water. (The settings shown here are borrowed from the simple-point-charge [4] SPC/E model.) In addition to coordinates, topology and force-field settings, "LT" files can optionally include any other kind of LAMMPS settings including SHAKE constraints, k-space settings, and even group definitions. *(Unicode is supported.)*

```
# file "spce_simple.lt"
#
#    H1     H2
#      \   /
#        O

SPCE {

  # LAMMPS supports a large number of force-field styles. We must select
  # which ones we need. This information belongs in the "In Init" section.

  write_once("In Init") {
    units        real
    atom_style   full
    pair_style   lj/charmm/coul/long 9.0 10.0 10.0
    bond_style   harmonic
    angle_style  harmonic
    kspace_style pppm 0.0001
    pair_modify  mix arithmetic
  }
```

```
# AtomID     MolID    AtomType  charge  coordX   coordY   coordZ

write("Data Atoms") {
  $atom:O     $mol    @atom:O  -0.8476  0.0000000 0.000000 0.000000
  $atom:H1    $mol    @atom:H   0.4238  0.8164904 0.5773590 0.00000
  $atom:H2    $mol    @atom:H   0.4238  -0.8164904 0.5773590 0.00000
}

# AtomType  Mass

write_once("Data Masses") {
  @atom:O 15.9994
  @atom:H 1.008
}

#  BondID    BondType  AtomID1  AtomID2

write("Data Bonds") {
  $bond:OH1  @bond:OH  $atom:O  $atom:H1
  $bond:OH2  @bond:OH  $atom:O  $atom:H2
}

# AngleID    AngleType   AtomID1 AtomID2 AtomID3

write("Data Angles") {
  $angle:HOH @angle:HOH $atom:H1 $atom:O $atom:H2
}

# AtomType  epsilon sigma

write_once("Data Pair Coeffs") {
  @atom:O   0.1553  3.166
  @atom:H   0.0     2.058
}

# BondType    k      r_0

write_once("Data Bond Coeffs") {
  @bond:OH    200.00 1.0
}

#  AngleType  k       theta_0

write_once("Data Angle Coeffs") {
  @angle:HOH  200.0   109.47
}

write_once("In Settings") {
```

```
    group spce type  @atom:O  @atom:H
    fix fSHAKE spce shake 0.0001 10 100 b @bond:OH a @angle:HOH
    # (Remember to "unfix" fSHAKE during minimization.)
  }
} # SPCE
```

Words which are preceded by "$" or "@" characters are counter variables and will be replaced by integers. (See section 5.2 for details.) Users can include SPCE water in their simulations using commands like these:

```
# -- file "system.lt" --
import "spce_simple.lt"
wat = new SPCE [1000]
```

You can now use "moltemplate.sh" to create simulation input files for LAMMPS

```
moltemplate.sh -pdb coords.pdb -atomstyle "full" system.lt
```

This command will create lammps input files for the molecular system described in "system.lt", using the desired atom style ("full" by default). In this example, moltemplate is relying on an external file ("coords.pdb") to supply the atomic coordinates of the water molecules, as well as the periodic boundary conditions. Coordinates in XYZ format are also supported using "-xyz coords.xyz".

Details

Note that since XYZ files lack boundary information, you must also include a "Boundary" section in your ".lt" file, as demonstrated in section 4.2. In both cases, the order of the atom types in a PDB or XYZ file (after sorting) should match the order they are created by moltemplate (which is determined by the order of the "new" commands in the LT file). Unfortunately this may require careful manual editing of the PDB or XYZ file.

4.2 Coordinate generation

It is not necessary to provide a separate file with atomic coordinates. It is more common to manually specify the location (and orientation) of the molecules in your system using the ".move()" and ".rot()" commands in the LT file itself (discussed in section 6). For example you can replace the line:

```
wat = new SPCE [1000]
```

from the example above with 1000 lines:

```
wat1    = new SPCE
wat2    = new SPCE.move(3.450, 0.0, 0.0)
wat3    = new SPCE.move(6.900, 0.0, 0.0)
wat4    = new SPCE.move(10.35, 0.0, 0.0)
  :           :
wat1000 = new SPCE.move(34.50, 34.50, 34.50)
```

Specifying geometry this way is tedious. Alternatively, moltemplate has simple commands for arranging multiple copies of a molecule in periodic, crystalline, toroidal, and helical 1-D, 2-D, and 3-D lattices. For example, you can generate a simple cubic lattice of 10×10×10 water molecules (with a 3.45 Angstrom spacing) using a single command (which in this example we split into multiple lines)

```
wat  = new SPCE [10].move(0,0,3.45)
                [10].move(0,3.45,0)
                [10].move(3.45,0,0)
```

(See section 6 for more details and examples.) This will create 1000 molecules with names like "wat[0][0][0]", "wat[0][0][1]",..., "wat[9][9][9]". You can always access individual atomIDs, molIDs, bondIDs, angleIDs, and dihedralIDs (if present), for any molecule elsewhere in your LT files using this notation: "$atom:wat[2][3][4]/H1", "$bond:wat[0][5][1]/OH1", "$angle:wat[2][8][3]/HOH", "$mol:wat[0][1][2]". This allows you to define interactions which link different molecules together (see section 6).

A list of available coordinate transformations is provided in section 3.3.

Boundary Conditions:

LAMMPS simulations have finite volume and are usually periodic. We must specify the dimensions of the simulation boundary using the "write_once("Data Boundary")" command.

```
write_once("Data Boundary") {
   0.0  34.5  xlo xhi
   0.0  34.5  ylo yhi
   0.0  34.5  zlo zhi
}
```

This is usually specified in the outermost LT file ("system.lt" in this example). *(Note: Boundary conditions do not have to be rectangular or even periodic. For triclinic cells, additional "xy", "xz", and "yz" tilt parameters can be added. For details, lookup the "read_data" and "boundary" commands in the official LAMMPS documentation.)*

This system is shown in figure 2a). After you have specified the geometry, then you can run moltemplate.sh this way:

```
moltemplate.sh -atomstyle "full" system.lt
```

4.3 Running a LAMMPS simulation (after using moltemplate)

To run a simulation of one or more molecules, LAMMPS requires an *input script* and a *data file*. Input scripts typically contain force field styles, parameters and run settings. (They sometimes also contain atom coordinates.) Data files typically contain atom coordinates and bonded topology data. (They sometimes also contain force-field parameters.)

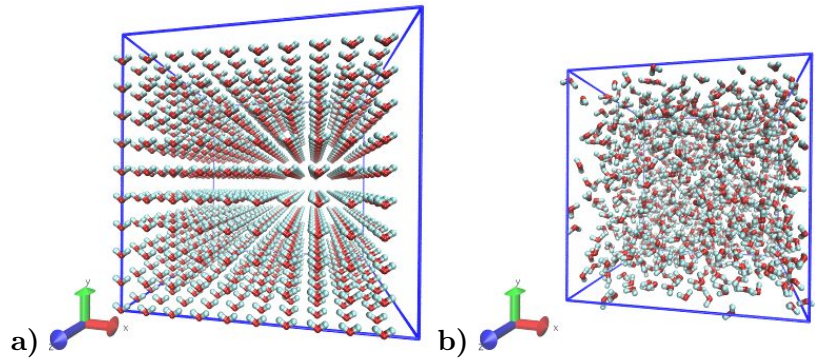


Figure 2: A box of 1000 water molecules (before and after pressure equilibration), generated by moltemplate and visualized by VMD with the topotools plugin. (The VMD console commands used for visualization were: "topo readlammpsdata system.data full", "animate write psf system.psf", "pbc wrap -compound res -all", and "pbc box". See the online examples for details.)

Moltemplate will create the following files: "system.data", "system.in", "system.in.init", "system.in.settings", (and possibly other files including "system.in.coords"). These are LAMMPS input/data files, and they can be run in LAMMPS with minimal modification (see below). The main input script file is named "system.in", and it usually contains just three lines:

```
include   "system.in.init"
read_data "system.data"
include   "system.in.settings"
```

To *run* a simulation, you will have to edit this file in order to add a couple of run commands. These commands tell LAMMPS about the simulation conditions you want to use (temperature, pressure), how long to run the simulation, how to integrate the equations of motion, and how to write the results to a file (file format, frequency, etc). Moltemplate.sh can not do this for you. Some simple examples (which you can paste into your input script) are provided in the *online examples* which can be downloaded from `http://moltemplate.org`. (These example input scripts typically have names like "run.in.nvt" and "run.in.npt".)

5 Overview

5.1 Basics: The *write()* and *write_once()* commands

Each LT file typically contains one or more "write" or "write_once" commands. These commands have the following syntax

```
write_once(filename) {text_block}
```

This creates a new file with the desired file name and fills it with the text enclosed in curly brackets {}. Text blocks usually span multiple lines and contain counter variables (beginning with "@" or "$"). which are replaced

with numbers. However the "write()" command will repeatedly append the same block of text to the file every time the molecule (in which the write command appears) is generated or copied (using the "new" command, after incrementing the appropriate counters, as explained in 5.2.2).

5.2 Basics: counter variables

Words following a "@" or a "$" character are *counter variables*. By default, *all counter variables are substituted with a numeric counter* before they are written to a file. These counters begin at 1 (by default), and are incremented as the system size and complexity grows (see below).

These words typically contain a colon (:) followed by more text. The text preceding this colon is the *category name*. (For example: "$atom:", "$bond:", "$angle:", "@atom:", "@bond:", "@angle:") Variables belonging to different categories are counted independently.

Users can override these assignment rules and create custom categories. (See appendices C.1 and C.2 for details.)

(Unicode is supported.)

5.2.1 Static counters begin with "@"

"@" variables generally correspond to *types*: such as atom types, bond types, angle types, dihedral types, improper types. These are simple variables and they assigned to unique integers in the order they are read from your LT files. Each uniquely named variable in each category is assigned to a different integer. For example, "@bond:" type variables are numbered from "1" to the number of *bond types*. (Pairs of bonded atoms are assigned a *bond type*. Later, LAMMPS will use this integer to lookup the bond-length and Hooke's-law elastic constant describing the force between these two atoms.)

5.2.2 Instance counters begin with "$"

On the other hand, "$" variables correspond to unique ID numbers: atom-IDs, bond-IDs, angle-IDs, dihedral-IDs, improper-IDs, and molecule-IDs. These variables are created whenever a copy of a molecule is created (using the "new" command). If you create 1000 copies of a water molecule using a command like

```
wat = new SPCE[10][10][10]
```

then moltemplate creates 3000 "$atom" variables with names like

```
$atom:wat[0][0][0]/O
$atom:wat[0][0][0]/H1
$atom:wat[0][0][0]/H2
$atom:wat[0][0][1]/O
$atom:wat[0][0][1]/H1
$atom:wat[0][0][1]/H2
```

⋮

```
$atom:wat[9][9][9]/O
$atom:wat[9][9][9]/H1
$atom:wat[9][9][9]/H2
```

5.2.3 Variable names: short-names *vs.* full-names

In the example above, the $ variables have full-names like "$atom:wat[8][3][7]/H1", not "$atom:H1". However inside the definition of the water molecule, you don't specify the full name. You can refer to this atom as "$atom:H1". Likewise, the full-name for the @atom variables is actually "@atom:SPCE/H", not "@atom:H". However inside the definition of the water molecule, you typically use the shorthand notation "@atom:H".

5.2.4 Numeric substitution

Before being written to a file, every variable (either $ or @) with a unique *full-name* will be assigned to a unique integer, starting at 1 by default.

The various $atom variables in the water example will be substituted with integers from 1 to 3000 (assuming no other molecules are present). But the "@atom:O" and "@atom:H" variables (which are shorthand for "@atom:SPCE/O" and "@atom:SPCE/H") will be assigned to to "1" and "2" (again, assuming no other molecule types are present).

So, in summary, @ variables increase with the *complexity* of your system (IE the number of molecule types or force-field parameters), but $ variables increase with the *size* of your system.

5.2.5 Variable scope

This effectively means that all variables are specific to local molecules they were defined in. In other words, an atom type named "@atom:H" inside the "SPCE" molecule, will be assigned to a different number than an atom named "@atom:H" in an "Arginine" molecule. This is because the two variables will have different *full* names ("@atom:SPCE/H", and "@atom:Arginine/H").

Sharing atom types or other variables between molecules

If you want to share atom types between molecules, you can place them outside the current molecule definition. Later you can use file-system-path-like syntax ("../", or "../../" or "/") to access atoms (or molecules) outside of the current molecule. For example, two different molecule types can share the same type of hydrogen atom by referring to it using this syntax: "@atom:../H". (More complex nested relationships are possible. For example see section 11.6.) *(Further details of variable syntax are discussed in appendix F.)*

5.3 Troubleshooting using the *output_ttree* directory

Users can see what numbers were assigned to each variable by inspecting the contents of the "output_ttree" subdirectory created by moltemplate. Unfor-

tunately, it is typical for LAMMPS to crash the first time you attempt to run it on a DATA file created by moltemplate. This often occurs if you failed to spell atom types and other variables consistently. The LAMMPS error message (located at the end of the "log.lammps" file created by LAMMPS) will help you determine what type of mistake you made. (For example, what type of variable was misspelled or placed in the wrong place?)

To help you, the "output_ttree" directory contains a file named "ttree_assignments.txt". This is a simple 2-column text file containing a list of *all* of the variables you have created in one column, and the numbers they were assigned to in the other column. This directory also contains all of the files that you created. The versions with a ".template" extension contain text interspersed with *full* variable names (before numeric substitution). (A spelling mistake, like using "$atom:H" when you meant to say "$atom:H1" or "@atom:H" will show up in these files if you inspect them carefully.) This can help you identify where the mistake occurred in your LT files.

Once a molecular system is debugged and working, users can ignore or discard the contents of this directory.

5.4 "Data" and "In"

If you are familiar with LAMMPS, you may have noticed the file names above (in the example from section 4.1) sound suspiciously like sections from LAMMPS DATA files or input scripts, such as "Data Boundary", "Data Atoms", "Data Bonds", "Data Masses", "Data Angles", "Data Dihedrals", "Data Impropers", "Data Pair Coeffs", "Data Bond Coeffs", "Data Angle Coeffs", "Data Dihedral Coeffs", "Data Improper Coeffs", "Data Angles By Type", "Data Dihedrals By Type", "Data Impropers By Type", "In Init", "In Settings"). All files whose names begin with "In " or "Data " are special. For the user's convenience, the moltemplate.sh script copies the contents of these files into the corresponding section of the DATA file or INPUT scripts generated by moltemplate ("system.data", "system.in.settings", etc). (Then the original files are moved to the "output_ttree/" directory, in an effort to clean things up and hide them from view.) Users can create their own custom sections to a LAMMPS data file. (See section 5.6. Note: It is unwise to add blank lines to a data file section. Moltemplate will add the section headers and blank lines needed to keep LAMMPS happy.)

However the "write()" and "write_once()" commands are not only used for generating sections from a DATA file or INPUT scripts. Any file can be created. Files whose names do not begin with "In " or "Data " can have any format (and are not moved or cleaned up). (See section 5.5 for an example.)

5.5 *(Advanced)* Using moltemplate to generate auxiliary files

The following excerpt from an LT file creates a file named "system.in.sw". (It contains parameters for the "sw" pair style. This exotic many-body pair style requires a large number of parameters, which are read from a separate file.) This "system.in.sw" file file will be read later when you run the simulation. (The pair_coeff command below tells LAMPS to read that file.)

```
write_once("system.in.sw") {
  mW mW mW 6.189 2.3925 1.8 23.15 1.2 -0.33333 7.04956 0.602224 4 0 0
}
write_once("In Settings") {
  pair_coeff * * sw system.in.sw mW NULL NULL NULL
}
```

As new force-field styles and/or fixes are added to LAMMPS, the files they depend on can be embedded in an LT file in this way.

5.6 *(Advanced)* Making custom DATA sections

Suppose that in the future, the format of the LAMMPS DATA file changes so that it now becomes necessary to supply a new section named "Foo Fee Fum", for example. You could do that using this command:

```
write_once("Data Foo Fee Fum") {
  File contents goes here. (These files can contain
  atom counters and/or other counter variables).
}
```

This way moltemplate copy this text into the "Foo Fee Fum" section at the end of the DATA file it is constructing. This allows users to adapt to future changes in the LAMMPS data file format.

Details/Comments

Note that it is okay for static and instance variables to share the same variable names (such as "@atom:O" and "$atom:O", or "@angle:HOH" and "$angle:HOH"). These are distinct and are counted separately.

Variable and molecule names can include unicode characters and whitespace such as "@{atom: CA }" or "@atom:\ CA\ ".

6 Object composition and coordinate generation

Each time a molecule is created using the "new" command, it can be rotated, moved, and scaled. (Individual atom positions can be customized later using the "write("Data Atoms")" command. Molecules are free to move during the simulation, of course.) The initial atomic coordinates of a large compound object is typically created by moving each individual component.

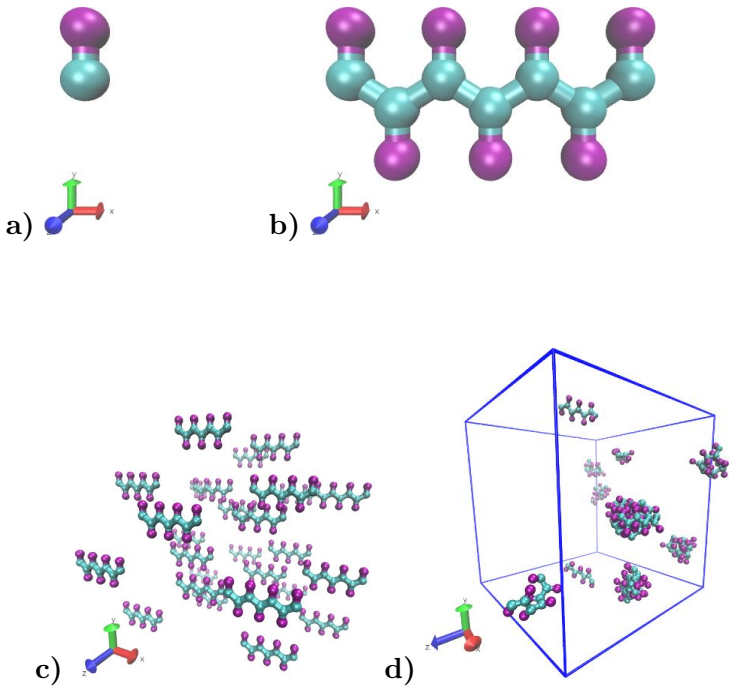


Figure 3: **a)-b)** *Building a complex system from small pieces:* Construction of a polymer (**b**) out of smaller (2-atom) subunits (**a**) using composition and rigid-body transformations. Bonds connecting different residues together (blue) must be declared explicitly, but angle and dihedral interactions will be generated automatically. See section 6.1 for details. **c)** An irregular lattice of short polymers. (See section 9.) **d)** The same system after 100000 time steps using Langevin dynamics. (The VMD console commands used for visualization were: "topo readlammpsdata system.data full", "animate write psf system.psf", "pbc wrap -compound res -all", and "pbc box". See online examples for details.)

6.1 Building a large molecule from smaller pieces

As an example, we define a small 2-atom molecule "2bead", construct a short polymer ("peptide") using it as a building block, and create a lattice of peptides. At each step, we define the relative orientation and position of each unit.

Consider the following simple molecule

```
# -- file "2bead.lt" --
```

```
2bead {

  write_once("In Init") {
    # -- Default styles for "2bead" --
    units           real
    atom_style      full
    bond_style      harmonic
    angle_style     harmonic
    dihedral_style  charmm
    pair_style      lj/cut/coul/debye 0.1 11.0
    pair_modify     mix arithmetic
    dielectric      80.0
    special_bonds   lj 0.0 0.0 0.0
  }

  # atom-id  mol-id   atom-type  charge  x      y        z

  write("Data Atoms") {
    $atom:CA $mol:... @atom:CA    0.0  0.000  1.0000   0.0000000
    $atom:R  $mol:... @atom:R     0.0  0.000  4.4000   0.0000000
  }

  #  Note: The "..." in "$mol:..." tells moltemplate that this molecule
  #        may be a part of a larger molecule, and to use the larger
  #        parent object's molecule id number as it's own, if present.


  # atom-type  mass

  write_once("Data Masses") {
    @atom:CA   13.0
    @atom:R    50.0
  }

  # atom-type   epsilon sigma

  write_once("Data Pair Coeffs") {
   @atom:CA        0.10 2.0
   @atom:R         0.50 3.6
  }

  # bond-id   bond-type        atom-id1  atom-id2

  write("Data Bonds") {
    $bond:CR  @bond:sidechain  $atom:CA  $atom:R
  }

  # bond-type          k     r0
```

```
  write_once("Data Bond Coeffs") {
    @bond:sidechain   30.0   3.4
    @bond:backbone    30.0   3.7
  }

  # Although there's no need to define angular interactions (because this
  # "molecule" only contains two atoms), we define the settings for angles
  # or dihedrals which might be present later when we build a polymer.

  # angle-type         k    theta0

  write_once("Data Angle Coeffs") {
    @angle:backbone   30.00  114
    @angle:sidechain  30.00  123
  }

  # dihedral-type      K  n   d  w

  write_once("Data Dihedral Coeffs") {
    @dihedral:back   -0.5 1 -180 0.0
    @dihedral:side   -1.5 1 -180 0.0
  }

  #  Rules for determining 3 and 4-body bonded interactions by type

  # angle-type       atomType1 atomType2 atomType3  bondType1 bondType2

  write_once("Data Angles By Type") {
    @angle:backbone  @atom:CA @atom:CA  @atom:CA   @bond:*   @bond:*
    @angle:sidechain @atom:CA @atom:CA  @atom:R    @bond:*   @bond:*
  }

  # dihedral-type AtomType1 AtomType2 AtomType3 AtomType4 bondType1 btyp2 btyp3

  write_once("Data Dihedrals By Type") {
    @dihedral:back @atom:CA  @atom:CA @atom:CA @atom:CA @bond:* @bond:* @bond:*
    @dihedral:side @atom:R   @atom:CA @atom:CA @atom:R  @bond:* @bond:* @bond:*
  }

} # 2bead
```

6.1.1 Building a simple polymer

We construct a short polymer by making 7 copies of "2bead", rotating and moving each copy:

```
# -- file "peptide.lt" --
```

```
import "2bead.lt"

Peptide {
  res1 = new 2bead
  res2 = new 2bead.rot(180.0, 1,0,0).move(3.2,0,0)
  res3 = new 2bead.rot(360.0, 1,0,0).move(6.4,0,0)
  res4 = new 2bead.rot(540.0, 1,0,0).move(9.6,0,0)
  res5 = new 2bead.rot(720.0, 1,0,0).move(12.8,0,0)
  res6 = new 2bead.rot(900.0, 1,0,0).move(16.0,0,0)
  res7 = new 2bead.rot(1080.0, 1,0,0).move(19.2,0,0)

  # Now, link the residues together this way:
  write("Data Bonds") {
    $bond:backbone1  @bond:2bead/backbone  $atom:res1/CA  $atom:res2/CA
    $bond:backbone2  @bond:2bead/backbone  $atom:res2/CA  $atom:res3/CA
    $bond:backbone3  @bond:2bead/backbone  $atom:res3/CA  $atom:res4/CA
    $bond:backbone4  @bond:2bead/backbone  $atom:res4/CA  $atom:res5/CA
    $bond:backbone5  @bond:2bead/backbone  $atom:res5/CA  $atom:res6/CA
    $bond:backbone6  @bond:2bead/backbone  $atom:res6/CA  $atom:res7/CA
  }

  create_var { $mol:. }  #<-creates a molecule ID number for this (".") object

  # This causes atoms in res1,res2,res3,...,res7 to share the same molecule ID
  # because in the 2bead.lt file, the "..." in "$mol:..." preferentially looks
  # for a counter of that type appearing in a "write()" statement (or a
  # "create_var" statement) in a parent molecule or earlier ancestor.
}
```

The position and orientation of each copy of "2bead" is specified after the "new" statement. Each "new" statement is typically followed by a chain of move/rotate/scale functions separated by dots, evaluated left-to-right (optionally followed by square brackets and then more dots). For example, "res2" is a copy of "2bead" which is first rotated 180 degrees around the X axis (denoted by "1,0,0"), and **then** moved in the (3.2,0,0) direction. (The last three arguments to the "rot()" command denote the axis of rotation, which does not have to be normalized.) (A list of available coordinate transformations is provided in section 3.3.)

(Note: Although we did not do this here, it is sometimes convenient to represent polymers as 1-dimensional arrays. See sections 7 and 7.4 for examples.)

To bond atoms in different molecules or molecular subunits together, we used the write("Data Bonds") command to append additional bonds to the system. (Because we are outside the definition of these molecules, we must be careful to refer to the atom-IDs and bond-types by their *full-names*. Here I'm using the "@bond:backbone" settings which were defined in the "2bead" molecule, so I refer to the bond type as "@bond:2bead/backbone". See section 5.2.3.)

6.1.2 Sharing atom, bond and angle types

Normally you must separately define the parameters for all of the atoms types, and bond types, angle types etc... in every type of molecule. However different kinds of monomers in a heteropolymer typically will share some common backbone atom types and other properties. You must be careful to indicate which atom and bond types are shared between different monomers by referring them using a "../" prefix. (See sections 5.2.5, 11.6, and 11.5 for details and examples.) *Note: There is a heteropolymer example in the the "2bead_heteropolymer/" directory in the online examples. This example demonstrates how to share backbone atoms, bonds, and angles. You can also define specific angle or dihedral interactions which are specific to the atom types in different residues.*

6.2 Bonded interactions *by type*

In this example we did *not* provide a list of all 3-body and 4-body forces between bonded atoms in the polymer. (for example using the "write_once("Data Angles")" command from section 4.1, *or* the "write_once("Data Dihedrals")", or "write_once("Data Impropers")" commands.) Instead we provided moltemplate.sh with instructions to help it figure out which atoms participate in 3-body and 4-body bonded interactions. Moltemplate can detect consecutively bonded atoms and determine the forces between them based on atom type. (Bond type can also be used as a criteria.) We did this in "2bead.lt" using the "write_once("Angles By Type")" and "write_once("Dihedrals By Type")" commands. You can also generate improper interactions between any 4-atoms bonded together in a T-shaped topology using the "write_once("Impropers By Type")" command. See appendix A for more details. *(More general interactions are possible. See appendix E.2.)*

7 Arrays and coordinate transformations

Moltemplate supports 1-dimensional, and multi-dimensional arrays. These can be used to create straight (or helical) polymers sheets, tubes, torii. They are also to fill solid 3-dimensional volumes with molecules or atoms. (See sections 4.2 and 9.)

Here we show an easier way to create the short polymer shown in section 6.1.1. You can make 7 copies of the *2bead* molecule this way:

```
res = new 2bead [7]
```

This creates 7 new *2bead* molecules (named *res[0]*, *res[1]*, *res[2]*, *res[3]*, ... *res[6]*). Unfortunately, by default, the coordinates of each molecule are identical. To prevent the atom coordinates from overlapping, you have several choices:

7.1 Transformations following brackets [] in a new statement

After every square-bracket [] in a new command, you can specify a list of transformations to apply. For example, we could have generated atomic

coordinates for the the short polymer in section 6.1.1 using this command:

```
res = new 2bead [7].rot(180, 1,0,0).move(3.2,0,0)
```

This will create 7 molecules. The coordinates of the first molecule *res[0]* are will be unmodified. However each successive molecule will have its coordinates cumulatively modified by the commands "rot(180, 1,0,0)" followed by "move(3.2,0,0)".

optional: initial customizations (preceding [] brackets)

You can also make adjustments to the initial coordinates of the molecule before it is copied, and before any of the array transformations are applied. For example:

```
res = new 2bead.scale(1.5) [7].rot(180, 1,0,0).move(3.2,0,0)
```

In this example, the "scale(1.5)" transformation is applied once to enlarge every *2bead* monomer initially. This will happen before any of the rotation and move commands are applied to build the polymer (so the 3.2 Angstrom spacings between each monomer will not be effected).

7.2 Transformations following instantiation

Alternately you apply transformations to a molecule after they have been created (even if they are part of an array).

```
res = new 2bead [7]

# Again, the first line creates the molecules named
# "res[0]", "res[1]", "res[2]", "res[3]", ... "res[6]".
# The following lines move them into position.
res[1].rot(180.0, 1,0,0).move(3.2,0,0)
res[2].rot(360.0, 1,0,0).move(6.4,0,0)
res[3].rot(540.0, 1,0,0).move(9.6,0,0)
res[4].rot(720.0, 1,0,0).move(12.8,0,0)
res[5].rot(900.0, 1,0,0).move(16.0,0,0)
res[6].rot(1080.0, 1,0,0).move(19.2,0,0)
```

7.3 Transformation order (general case)

A typical array of molecules might be instantiated this way:

```
mols = new Molecule.XFORMS1() [N].XFORMS2()
mols[*].XFORMS3()
```

The list of transformations denoted by "XFORMS1" in this example are applied to the molecule first. Then the transformations in "XFORMS2" are then applied to each copy of the molecule multiple times. (For the molecule with index "i", named "Molecule[i]", XFORMS2 will be applied i times.) Finally after all the molecules have been created, the list of transformations in XFORMS3 will be applied. For example, to create a ring of 10 peptides of radius 30.0, centered at position (0,25,0), use this notation:

```
peptide_ring = new Peptide.move(0,30,0) [10].rot(36,1,0,0)
  # After creating it, we can move the entire ring
  # (These commands are applied last.)
peptide_ring[*].move(0,25,0)
```

7.4 Random arrays

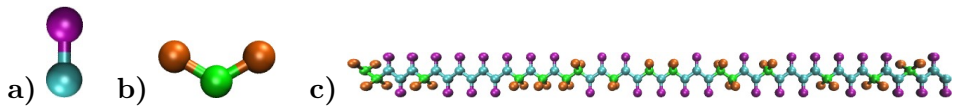


Figure 4: A random heteropolymer (**c**), composed of of *2bead* and *3bead* monomers (**a** and **b**) in a 3:2 ratio.

Arrays of random molecules can be generated using the *new random() []* syntax. For example, below we define a random polymer composed of 50 *2bead* and *3bead* monomers. (See figure 4.)

```
RandPoly50 {
  # Make a chain of randomly chosen monomers:

  monomers = new random([2bead, 3bead], [0.6, 0.4], 123456)
                 [50].rot(180,1,0,0).move(2.95, 0, 0)

  # Now, link the monomers together this way:
  write("Data Bonds") {
    $bond:bb1 @bond:backbone $atom:monomers[0]/CA $atom:monomers[1]/CA
    $bond:bb2 @bond:backbone $atom:monomers[1]/CA $atom:monomers[2]/CA
    $bond:bb3 @bond:backbone $atom:monomers[2]/CA $atom:monomers[3]/CA
    $bond:bb4 @bond:backbone $atom:monomers[3]/CA $atom:monomers[4]/CA
      ⋮
    $bond:bb50 @bond:backbone $atom:monomers[48]/CA $atom:monomers[49]/CA
  }
  #(Note: Both the "2bead" and "3bead" molecules contain atoms
  #       named "$atom:CA".  The atom types are different, however.)
} #RandPoly50
```

It is also possible to fill a 2 or 3-dimensional volume with molecules randomly. This is discussed in section 9.2.

The *new random()* function takes 2 or 3 arguments: a list of molecule types (*2bead* and *3bead* in this example), and a list of probabilities (*0.6* and *0.4*) both enclosed in square-brackets []. There is no limit to the number of molecule types which appear in these lists. (These lists can also contain vacancies/blanks. See section 9.3.) (An optional random-seed argument can also be included. For example the *"123456"* shown above. If you omit this number, then you will get different results each time you run moltemplate.) Note that once a molecule containing random monomers is defined, (*"RandPoly50"* in this example), each copy of that molecule (created using the *new* command) is identical.

optional: initial customizations (within *random()*)

As before, you may apply an initial transformation to each monomer type immediately after its name. For example to move the two monomer types closer or further away from the polymer axis, you can use:

```
monomers = new random([2bead.move(0,0.01,0),
                       3bead.move(0,-0.01,0)],
                      ...
```

These *move(0,0.01,0)* and *move(0,-0.01,0)* commands will be applied *before* the other rotate and move commands are applied which generate the polymer.

7.5 [*] and [i-j] notation

You can move the entire array of molecules using "[*]" notation:

```
res[*].move(0,0,40)
```

(Note that "res.move(0,0,40)" does not work. You must include the "[*]".) You can also use range limits to move only some of the residues:

```
res[2-4].move(0,0,40)
```

This will move only the third, fourth, and fifth residues.

Of course, as mentioned earlier, you can also always load atom coordinates from an external PDB or XYZ file. Such files can be generated by PACKMOL, or a variety of advanced graphical molecular modeling programs. For complex systems, this may be the best choice.

8 Customizing molecule position and topology

By default, each copy of a molecule created using the *new* command is identical. This need not be the case.

As discussed in section 7.2, individual molecules which were recently created can be moved, rotated, and scaled. You can also overwrite or delete individual atoms, bonds, and other interactions within a molecule, or their subunits. (See sections 8.3.2, 8.1, and 8.2.) You make any of these modifications to *some* copies of the molecule without effecting other copies. Furthermore, if those molecules are compound objects (if they contain individual molecular subunits within them), then you can rearrange the positions of their subunits as well. And all of this can be done from anywhere else in the LT file.

For example, suppose we used the "Peptide" molecule we defined above to create a larger, more complex "Dimer" molecule.

```
Dimer {
  peptides = new Peptide [2].rot(180,1,0,0).move(0, 12.4, 0)
}
dimer = new Dimer
```

The *Dimer* molecule is shown in figure 7a). *(Note: The rot() and move() commands are only applied to the the second peptide, as explained in section 7.1.)* We can customize the position of the 3rd residue of the second peptide this way:

```
dimer/peptide[1]/res[2].move(0,0.2,0.6)
```

This does not effect the position of *res[2]* in *peptide[0]* (or in any other *"Peptide"* molecule). If you want to move them both, you could use a wildcard character "*"

```
dimer/peptide[*]/res[2].move(0,0.2,0.6)
```

(You an also use ranged notation, such as "peptide[0-1]", as an alternative to "peptide[*]". See section 7.5. You could also modify the definition of the "Peptide" molecule. See section 10.)

8.1 Customizing individual atom locations

To customize the positions of *individual atoms*, don't use the "move" or "rot" commands. Instead simply overwrite their coordinates this way:

```
write("Data Atoms") {
  $atom:dimer/peptide[0]/res[2]/CA $mol:dimer/peptide[1] @atom:2bead/R 0 6.4 8.2 0.6
}
```

Note that because you are outside the definition of the 2bead molecule, you must be careful to refer to the atom-ID, molecule-ID, and atom-Type variables using their *full* names, as shown above. (Don't use the abbreviated names you normally use. See section 5.2.3.)

8.2 Adding bonds and angles to individual molecules

Adding additional bonds within a molecule can be accomplished by writing additional lines of text to the "Data Bonds" section. (This is what we did when we added bonds between residues to create a polymer in section 6.1.1.) Again, bonds and atom names must be referred to by their *full* names. Bonds and bonded interactions can be deleted using the "delete" command. (See section 8.3.)

8.3 The delete command

8.3.1 Deleting molecules or molecular subunits

Molecules can be further customized by deleting individual atoms, bonds, bonded-interactions, and entire subunits. We can **delete** the 3rd residue of the second peptide, use the "delete" command:

```
delete dimer/peptide[1]/res[2]
```

8.3.2 Deleting atoms, bonds, angles, dihedrals, and impropers

Individual atoms or bonds can be deleted in a similar way:

```
delete dimer/peptide[1]/res[3]/CA         #<-- deletes the "CA" atom
delete dimer/peptide[1]/res[4]/sidechain  #<-- deletes the "sidechain" bond
```

Whenever an atom or a molecule is deleted, the bonds, angles, dihedrals, and improper interactions involving those atoms are deleted as well. *(In fact, any lines of text in any "write()" statement containing references to deleted atoms are omitted.)*

When a bond is deleted, any angular, dihedral, or improper interactions are automatically generated by moltemplate are removed as well. (However other bonded interactions explicitly listed by the user in their "Data Angles", "Data Dihedrals", or "Data Impropers" sections are not removed. These need to be deleted manually.)

Multiple molecules can moved or deleted in a single command. For example, the following command deletes the third, fourth, fifth residues from both peptide[0] and peptide[1]:

```
delete dimer/peptide[*]/res[2-4]
```

See section 7.5 for an explanation of ranged ("[2-4]") array notation, and wildcard characters ("*").

9 Multidimensional arrays

The same techniques work with multidimensional arrays. Coordinate transformations can be applied to each layer in a multi-dimensional array. For example, to create a cubic lattice of 3x3x3 peptides: you would use this syntax:

```
peptides = new Peptide [3].move(0, 0, 30.0)
                       [3].move(0, 30.0, 0)
                       [3].move(30.0, 0, 0)
```

(Similar commands can be used with rotations to generate objects with cylindrical, helical, conical, or toroidal symmetry.)

9.1 Customizing individual rows, columns, or layers

Similarly, you can customize the position of individual peptides, or layers or columns using the methods above:

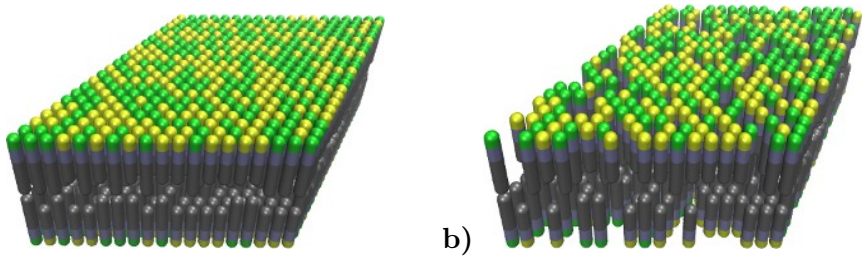
```
peptides[1][*][*].move(20,0,0)
peptides[*][1][*].move(0,0,20)
peptides[*][*][1].move(0,20,0)
```

(See figure 3c))

9.2 Creating random mixtures using multidimensional arrays

You can use *"new random()"* to fill space with a random mixture of molecules. The following 2-dimensional example creates a lipid bilayer (shown in figure 5) composed of an equal mixture of DPPC and DLPC lipids. (...Whose definition we omit here. See the online examples for details.)

```
import "lipids"                                          # define DPPC & DLPC
lipids = new random([DPPC,DLPC], [0.5,0.5], 123)         # "123"=random_seed
                    [19].move(7.5,    0,     0)          # lattice spacing 7.5
                    [22].move(3.75, 6.49519, 0)          # hexagonal lattice
                     [2].rot(180, 1, 0, 0)               # 2 monolayers
```



a) **b)**

Figure 5: A lipid bilayer membrane composed of a random equal mixture of two different lipid types in a 1:1 ratio. (See section 9.2.) In **b)** one of the molecule types was left blank leaving vacancies behind. (See section 9.3.)

9.3 Inserting random vacancies

The list of molecule types passed to the *random()* function may contain blanks. In the next example, 30% of the lipids are missing:

```
lipids = new random([DPPC, ,DLPC], [0.35,0.3,0.35], 123) # 2nd element is blank
                    [19].move(7.5,    0,     0)
                    [22].move(3.75, 6.49519, 0)
                     [2].rot(180, 1, 0, 0)
```

The results are shown in figure 5b). *(Note: When this happens, the array will contain missing elements. Any attempt to access the atoms inside these missing molecules will generate an error message, however moving or deleting array elements is always safe.)*

9.4 Cutting rectangular holes using delete

The delete command can be used to cut large holes in 1, 2, and 3-dimensional objects. For example, consider a simple 3-dimensional array of molecules:

```
molecules = new OneAtomMolecule [12].move(3.0,0,0)
                                [12].move(0,3.0,0)
                                [12].move(0,0,3.0)
```

```
delete molecules[*][*][2]
delete molecules[*][*][8]
delete molecules[6-7][0-8][5-6]
```

The result of these operations is shown in figure 6. *(Note: You may move or delete previously deleted array elements more than once, and/or deleting overlapping rectangular regions without error.)*

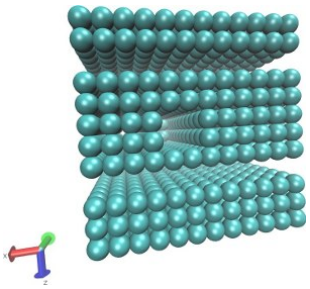


Figure 6: Rectangular holes can be carved out of an array of molecules (represented here by blue spheres) using the "delete" command. Three delete commands were used to remove the two planar regions and the rectangular hole in the center.

10 Customizing molecule *types*

You can create modified versions of existing molecule *types*, without having to redefine the entire molecule. For example:

```
Dimer0 = Dimer.move(-9.6,-6.2, 0).scale(0.3125)
```

or equivalently:

```
Dimer0 = Dimer
Dimer0.move(-9.6,-6.2, 0).scale(0.3125)
```

This creates a new type of molecule named "Dimer0" whose coordinates have been centered and rescaled. (Note that the "scale()" command only effects the atomic coordinates. (You will have to override earlier force field settings, such as atomic radii and bond-lengths in order for this to work properly.) If we want to make additional customizations (such as adding atoms, bonds, or molecular subunits), we could use this syntax:

```
Dimer0 = Dimer

# Add some new atoms connecting the two peptides in the dimer

Dimer0 {
  write("Data Atoms") {
    $atom:t1 $mol:. @atom:2bead/CA   0.0   23.0   0.0     0.0
    $atom:t2 $mol:. @atom:2bead/CA   0.0   24.7   4.0     0.0
    $atom:t3 $mol:. @atom:2bead/CA   0.0   24.7   8.4     0.0
```

```
    $atom:t4 $mol:. @atom:2bead/CA   0.0   23.0   12.4    0.0
  }
  write("Data Bonds") {
    $bond:b1 @bond:2bead/backbone $atom:peptides[0]/res7/CA $atom:t1
    $bond:b2 @bond:2bead/backbone $atom:t1 $atom:t2
    $bond:b3 @bond:2bead/backbone $atom:t2 $atom:t3
    $bond:b4 @bond:2bead/backbone $atom:t3 $atom:t4
    $bond:b5 @bond:2bead/backbone $atom:t4 $atom:peptides[1]/res7/CA
  }
}

# Center and rescale the atoms in all "Dimer0"
Dimer0.move(-9.6,-6.2, 0).scale(0.3125)
```

The result of these modifications is shown in figure 7b).

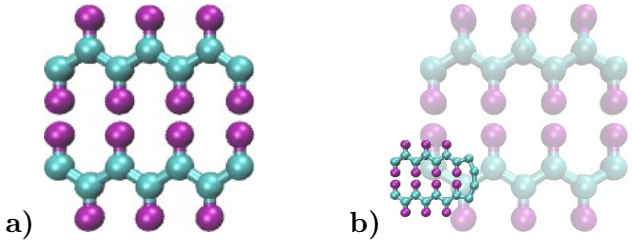


Figure 7: **a)** The "Dimer" molecule. This is a contrived example consisting of two "Peptides". See section 6.1.1 **b)** A customized version of the "Dimer" molecule. (The original "Dimer" is shown faded in the background for comparison.)

Note1: Coordinate transformations applied to entire molecule types are an experimental feature as of 2012-9-28. This feature has not been rigorously tested.

Note2: These coordinate transformations will be applied ***after*** *the molecule is constructed. Consequently, to make things clear, I recommend placing the coordinate transforms applied to an entire molecule type* ***after*** *all of its internal details (bonds, atoms, subunits) have been declared, as we did here.*

Note3: You may also want all of the atoms in "Dimer0" to share the same molecule-ID counter ("$mol"), so that LAMMPS realizes they belong to the same molecule. To do that you should delete the "create_var $mol:." line from the definition of the Peptide molecule, and add it to Dimer0.

(Advanced) Inheritance

The *Dimer0* molecule is a type of *Dimer* molecule. For those who are familiar with programming, relationships like this are analogous to the relationship between parent and child objects in an object-oriented programming language. More general kinds of inheritance are supported by moltemplate and are discussed in section 11.8.

(Advanced) Multiple Inheritance

If we wanted, we could have created a new molecule type (like *"Dimer0"*) which includes atom types and features from *multiple* different types of molecules. Section 11.8 mentions one way to do this and section 11.8.3 discusses alternate approaches.

Advanced moltemplate usage

11 Portability: Using *LT files* for force-field storage

The ".LT" format is a flexible file format for storing force field parameters in LAMMPS. If you want to share your ".LT" file with others, it's not safe to assume that all interactions use the same standard formula.

11.1 Mixing molecule types

LAMMPS has the ability to combine molecules using multiple different force-field styles together using. In section 4.1, we provided an example of an SPCE water molecule model. This example was simple to understand. However, as written, it would be impossible to combine this definition of water with other molecules which don't share the same simple bond or angle styles. For example, we used harmonic restoring forces to preserve the water angle at \$109.47, but other users may want to mix this SPCE water with a small number of molecules which use a more complicated angular potential formula, or tabular angle potentials. Using the "hybrid" keyword, you can avoid this limitation. A more robust example is included below.

```
# file "spce.lt"
#
#    H1     H2
#      \   /
#        O

SPCE {

  write_once("In Init") {
    # -- Default styles (for solo "SPCE" water) --
    units        real
    atom_style   full
    pair_style   hybrid lj/charmm/coul/long 9.0 10.0 10.0
    bond_style   hybrid harmonic
    angle_style  hybrid harmonic
    kspace_style pppm 0.0001
    pair_modify  mix arithmetic
  }
```

```
# AtomID  MolID("."=this)  AtomType  charge  coordX   coordY   coordZ
write("Data Atoms") {
  $atom:O  $mol:. @atom:O -0.8476  0.0000000 0.00000 0.000000
  $atom:H1 $mol:. @atom:H  0.4238  0.8164904 0.00000  0.5773590
  $atom:H2 $mol:. @atom:H  0.4238  -0.8164904 0.00000 0.5773590
}

# atom-type  Mass
write_once("Data Masses") {
  @atom:O    15.9994
  @atom:H     1.008
}

# -- Forces between atoms (non-bonded) --

#           atomTypeI atomTypeJ    pair-style-name    parameter-list
write("In Settings") {
  pair_coeff   @atom:O @atom:O  lj/charmm/coul/long   0.1553  3.166
  pair_coeff   @atom:H @atom:H  lj/charmm/coul/long   0.0     2.058
}

# -- Forces between atoms (bonded) --

#  bond-id   bond-type  atom-id1 atom-id2
write("Data Bonds") {
  $bond:OH1  @bond:OH   $atom:O  $atom:H1
  $bond:OH2  @bond:OH   $atom:O  $atom:H2
}

#             bond-type   bond-style-name  parameter-list
write("In Settings") {
  bond_coeff   @bond:OH      harmonic      200.0   1.0
}

# angle-id   angle-type  atom-id1 atom-id2 atom-id3
write("Data Angles") {
  $angle:HOH @angle:HOH  $atom:H1 $atom:O $atom:H2
}

#              angle-type  angle-style-name  parameter-list
write("In Settings) {
  angle_coeff  @angle:HOH    harmonic        200.0   109.47
}

# miscellaneous
write_once("In Settings") {
  group spce type  @atom:O  @atom:H
  fix fSHAKE spce shake 0.0001 10 100 b @bond:OH a @angle:HOH
```

```
    # (Remember to "unfix" fSHAKE during minimization.)
  }

} # SPCE
```

There are two differences between this molecule definition and the "spce_simple.lt" example from section 4.1:

Hybrid force field styles

To experienced LAMMPS users, it may seem strange that in this example that we have chosen "hybrid" styles followed by only one force-field style ("harmonic"). However this will make your molecule easier to share with others. When other people use your LT file, they can override these styles as explained in section 11.2.

Force-field parameters are specified in "In Settings" instead of the "Data" file

We replaced the "write_once("Data Pair Coeffs")", "write_once("Data Bond Coeffs")", and "write_once("Data Angle Coeffs")" (which use lammps DATA file syntax) with "pair_coeff", "bond_coeff" and "angle_coeff" commands (which use lammps INPUT SCRIPT file syntax). (Of course, more complicated molecules would require dihedral and improper styles as well.) Again, moltemplate.sh copies all text located in a "write_once("In Settings")" statement into the LAMMPS input script. The various "coeff" commands above are LAMMPS input script commands, and therefore they must appear inside a "write_once("In Settings")" statement.

11.2 Combining molecules with different force field styles

Later on, if a user wants to combine the SPCE water molecule with another molecule which uses a tabular pair_style (for example), they would have to specify the complete hybrid pair_style in the "Init" section of their LT file. For example:

```
import "spce.lt"
import "other_molecule.lt"

write_once("In Init") {
  pair_style hybrid lj/charmm/coul/long 9 10 10 table spline 1000
}
```

Note: By placing the "write_once("In Init"){ }" statement *after* "import "spce.lt"", this insures that the pair_style commands issued here will override the pair_style commands issued earlier "spce.lt". This allows moltemplate users users to combine their molecules "spce.lt" file shown here with other template files without modification (assuming the atom styles match).

11.3 Nesting

Molecule names such as "Solvent" (or even "Water") are short and easy to type, but are vague and are not portable. If you use common, generic molecule names, you will not be able to combine your molecule templates with templates written by others (without carefully checking for naming conflicts). LT files were meant to be used for storing and exchanging libraries of different molecule types.

Suppose, for example, that you want to run a simulation consisting of different molecule types, each of which belong to different LT files. Suppose two of the LT files both happen to contain definitions for "Water". Moltemplate does not detect these name clashes automatically and instead attempts to merge the two versions of "Water" together, (most likely creating a molecule with 6 atoms instead of 3). This is presumably not what you want.

As the number of molecule types grows, the possibility of naming clashes increases. As the behavior of the same molecule can be approximated using many different force fields, one has to be careful to avoid clashing molecule names.

To alleviate the problem, you can "nest" your molecules inside the definition of other molecules or objects. This reduces the scope in which your molecule is defined. See section 11.5 for an example.

11.4 A simple force-field example

Force-field parameters can be shared by groups of related molecules. In the example below, we create an object named "TraPPE". Later we use it to define a new molecule named "Cyclopentane".

The following example defines a coarse-grained (united-atom) version of a "cyclopentane" molecule. (Hydrogen atoms have been omitted.) In this example, only the atom types (and positions) and the bonds connecting them need to be specified. The interactions between them are determined automatically by the settings in the force-field file "trappe1998.lt".

```
import "trappe1998.lt"

cyclopentane {

  # AtomID  MolID('.'=this) AtomType  charge coordX  coordY   coordZ
  write("Data Atoms") {
    $atom:C1 $mol:. @atom:TraPPE/CH2 0.0 0.0000 0.000000000 1.0000000
    $atom:C2 $mol:. @atom:TraPPE/CH2 0.0 0.0000 0.951056516 0.3090170
    $atom:C3 $mol:. @atom:TraPPE/CH2 0.0 0.0000 0.587785252 -0.809017
    $atom:C4 $mol:. @atom:TraPPE/CH2 0.0 0.0000 -0.587785252 -0.809017
    $atom:C5 $mol:. @atom:TraPPE/CH2 0.0 0.0000 -0.951056516 0.3090170
  }

  write("Data Bonds") {
    $bond:bond1 @bond:TraPPE/CC $atom:C1 $atom:C2
```

```
    $bond:bond2 @bond:TraPPE/CC $atom:C2 $atom:C3
    $bond:bond3 @bond:TraPPE/CC $atom:C3 $atom:C4
    $bond:bond4 @bond:TraPPE/CC $atom:C4 $atom:C5
    $bond:bond5 @bond:TraPPE/CC $atom:C5 $atom:C1
  }
}
```

(The "TraPPE/" is explained below.) We can create copies of this molecule in the same way we did with SPCE:

```
# A cubic lattice of 125 cyclopentane molecules (12-angstrom spacing)
mols = new Cyclopentane [5].move(0,0,12) [5].move(0,12,0) [5].move(12,0,0)
```

Unlike the SPCE example, we don't have to specify all of the interactions between these atoms because the atom and bond types (CH2, CC). match the type-names defined in the "trappe1998.lt" file. This file contains a collection of atom types and force-field parameters for coarse-grained hydrocarbon chains. (See [5] for details.) This way, the "CH2" atoms in cyclopentane will interact with, and behave identically to any "CH2" atom from any other molecule which uses the TraPPE force field. (The same is true for other atom types, and interaction-types which are specific to "TraPPE", such as "@atom:TraPPE/CH3", "@bond:TraPPE/CC", etc... Another molecule which uses the TraPPE force field is discussed later in section 11.5.) The important parts of the "trappe1998.lt" file are shown below:

11.4.1 Namespace example

```
# -- file "trappe1998.lt" --

TraPPE {
  write_once("Data Masses") {
    @atom:CH2 14.1707
    @atom:CH3 15.2507
  }
  write_once("In Settings") {
    bond_coeff     @bond:CC      harmonic   120.0   1.54
    angle_coeff    @angle:CCC    harmonic   62.0022 114
    dihedral_coeff @dihedral:CCCC opls 1.411036 -0.271016 3.145034 0.0
    pair_coeff @atom:CH2 @atom:CH2 lj/charmm/coul/charmm 0.091411522 3.95
    pair_coeff @atom:CH3 @atom:CH3 lj/charmm/coul/charmm 0.194746286 3.75
    # (Interactions between different atom types use mixing rules.)
    # (Hybrid styles were used for portability.)
  }
  write_once("Data Angles By Type") {
    @angle:CCC @atom:C* @atom:C* @atom:C* @bond:CC @bond:CC
  }
  write_once("Data Dihedrals By Type") {
   @dihedral:CCCC @atom:C* @atom:C* @atom:C* @atom:C* @bond:CC @bond:CC @bond:CC
  }
}
```

In addition to the atom-type names and masses, this file stores the force-field parameters (coeffs) for the interactions between them.

Bonded interactions *by type*

Again, the "Angles By Type" and "Dihedrals By Type" sections tell moltemplate.sh that bonded 3-body and 4-body interactions exist between any 3 or 4 consecutively bonded carbon atoms (of type CH2, CH3, or CH4) assuming they are bonded using "CC" (saturated) bonds. The "*" character is a wild-card. "C*" matches "CH2", "CH3", and "CH4". (Bond-types can be omitted or replaced with wild-cards "@bond:*".)

Namespaces and nesting:

Names like "CH2" and "CC" are extremely common. To avoid confusing them with similarly named atoms and bonds in other molecules, we enclose them ("nest" them) within a *namespace* ("TraPPE", in this example). Unlike "SPCE" and "Cyclopentane", "TraPPE" is not a molecule. It is just a container of atom types, bond-types and force-field parameters shared by other molecules. We do this to distinguish them from other atoms and bonds which have the same name, but mean something else. Elsewhere we can refer to these atom/bond types as "@atom:TraPPE/CH2" and "@bond:TraPPE/CC". (You can also avoid repeating the cumbersome "TraPPE/" prefix for molecules defined within the TraPPE namespace. For example, see section 11.5.)

11.5 Nested molecules

Earlier in section 11.4.1, we created an object named "TraPPE" and used it to create a molecule named "Cyclopentane". Here we use it to demonstrate nesting. Suppose we define a new molecule "Butane" consisting of 4 coarse-grained (united-atom) carbon-like beads, whose types are named "CH2" and "CH3".

```
# -- file "trappe_butane.lt" --

import "trappe1998.lt"

Butane {
  write("Data Atoms"){
    $atom:C1 $mol:. @atom:TraPPE/CH3 0.0  0.419372  0.000 -1.937329
    $atom:C2 $mol:. @atom:TraPPE/CH2 0.0  -0.419372  0.000 -0.645776
    $atom:C3 $mol:. @atom:TraPPE/CH2 0.0  0.419372  0.000  0.645776
    $atom:C4 $mol:. @atom:TraPPE/CH3 0.0  -0.419372  0.0000 1.937329
  }
  write("Data Bonds"){
    $bond:b1 @bond:TraPPE/CC $atom:C1 $atom:C2
    $bond:b2 @bond:TraPPE/CC $atom:C2 $atom:C3
    $bond:b3 @bond:TraPPE/CC $atom:C3 $atom:C4
```

```
  }
}
```

Alternately, as mentioned above, it may be simpler to nest our "Butane" within "TraPPE", so that so that it does not get confused with other (perhaps all-atom) representations of butane. In that case, we would use:

```
# -- file "trappe_butane.lt" --

import "trappe1998.lt"

TraPPE {
  Butane {
    write("Data Atoms"){
      $atom:C1 $mol:. @atom:../CH3 0.0  0.419372  0.000 -1.937329
      $atom:C2 $mol:. @atom:../CH2 0.0  -0.419372  0.000 -0.645776
      $atom:C3 $mol:. @atom:../CH2 0.0  0.419372  0.000  0.645776
      $atom:C4 $mol:. @atom:../CH3 0.0  -0.419372  0.0000 1.937329
    }
    write("Data Bonds"){
      $bond:b1 @bond:../CC $atom:C1 $atom:C2
      $bond:b2 @bond:../CC $atom:C2 $atom:C3
      $bond:b3 @bond:../CC $atom:C3 $atom:C4
    }
  }
}
```

Note: Wrapping Butane within "TraPPE{ }" clause merely appends additional content to be added to the "TraPPE" object defined in the "trappe1998.lt" file (which was included earlier). It does not overwrite it. Again "../" tells moltemplate use the "CH2" atom defined in the context of the TraPPE environment (IE. one level up). This insures that moltemplate does not create a new "CH2" atom type which is local to the Butane molecule. (Again, by default all atom types and other variables are local. See section 5.2.5.)

To use this butane molecule in a simulation, you would import the file containing the butane definition, and use a "new" command to create one or more butane molecules.

```
import "trappe_butane.lt"
new butane = TraPPE/Butane
```

(You don't need to import "trappe1998.lt" in this example because it was imported within "trappe_butane.lt".) The "TraPPE/" prefix before "Butane" lets moltemplate/ttree know that butane was defined *locally* within TraPPE.

Note: An alternative procedure using ***inheritance*** *exists which may be a cleaner way to handle these kinds of relationships. See sections 11.8 and 11.8.1.*

11.6 Path syntax: "../", ".../", and "$mol:."

Generally, multiple slashes ("/") as well as ("../") can be used build a path that indicates the (relative) location of any other molecule in the object hierarchy. (The ".", "/" and ".." symbols are used here in the same way they are used to specify a path in a unix-like file-system. For example, the "." in "$mol:." refers to the current molecule (instance), in the same way that "./" refers to the current directory. (Note: *"$mol"* is shorthand for *"$mol:."*)

A slash by itself, "/", refers to the *global environment*. This is the outermost environment in which all molecules are defined/created.

(Advanced) Ellipsis notation ".../"

If you are using multiple levels of nesting, and if you don't know (or if you don't want to specify) where a particular molecule type or atom type (such as "CH2") was defined, you can refer to it using ".../CH2" instead of "../CH2". The "..." ellipsis syntax searches up the tree of nested molecules to find the target (the text following the "/" slash).

(Advanced) $mol:... notation

Molecules can contain multiple layers of hierarchy, however all the atoms share the same molecule ID. To refer to the ID of the molecule to which you belong, use "$mol:...". (If none of the molecules which instantiate the current molecule define a variable in the $mol category, then a new local $mol variable will be created automatically.

The "..." syntax is explained more formally in appendix F.)

11.7 *using namespace* syntax

Because the *Butane* molecule was defined within the *TraPPE* environment, you normally have to indicate this when you refer to it later. For example, to create a copy of a *Butane* molecule, you would normally use:

```
import "trappe_butane.lt"

butane = new TraPPE/Butane
```

However for convenience, you can use the "**using namespace**" declaration so that, in the future, you can quickly refer to any of the molecule types defined within *TraPPE* directly, without having to specify their path.

```
import "trappe_butane.lt"

using namespace TraPPE

butane = new Butane
```

This only works for molecule types, not atom types

Unfortunately, you still *must* always **refer to** atom types, bond types, and any other **primitive types explicitly** (by their full path). For example, the second line in the *"Data Atoms"* in the example below does not refer to the *CH2* atom type defined in *TraPPE*. (Instead it creates a *new* atom type, which is probably not what you want.)

```
import "trappe_butane.lt"
using namespace TraPPE
butane = new Butane
write("Data Atoms") {
  $atom:C1 $mol @atom:TraPPE/CH2 0.0   0.41937 0.00 1.9373  # <-- yes
  $atom:C2 $mol @atom:CH2        0.0  -0.41937 0.00 -0.6457 # <-- no, bad idea
}
```

If, for example, you want to leave out the "TraPPE/" prefix when accessing the atom, bond, and angle types defined in TraPPE, then instead you can define a new molecule which *inherits* from TraPPE. (See section 11.8.)

11.8 Inheritance

We could have defined *Butane* this way:

```
import "trappe1998.lt"

Butane inherits TraPPE {
  write("Data Atoms"){
    $atom:C1 $mol:. @atom:CH3 0.0  0.419372  0.000 -1.937329
    $atom:C2 $mol:. @atom:CH2 0.0  -0.419372  0.000 -0.645776
    $atom:C3 $mol:. @atom:CH2 0.0  0.419372  0.000  0.645776
    $atom:C4 $mol:. @atom:CH3 0.0  -0.419372  0.0000 1.937329
  }
  write("Data Bonds"){
    $bond:b1 @bond:CC $atom:C1 $atom:C2
    $bond:b2 @bond:CC $atom:C2 $atom:C3
    $bond:b3 @bond:CC $atom:C3 $atom:C4
  }
}
```

A molecule which *inherits* from another molecule (or namespace) *is* a particular type of that molecule (or namespace). Defining *Butane* this way allows it to access all of molecule types, atom types, and bond types, etc... defined within *TraPPE* as if they were defined locally. (I did not have to refer to the CH3 atom types as "@atom:TraPPE/CH3", for example.)

11.8.1 Multiple inheritance:

A molecule can inherit from multiple parents. This is one way you can allow the *Butane* molecule to borrow atom, bond, angle, dihedral, and improper types from *multiple* different force-field parents:

```
import "trappe1998.lt"
import "dreiding1990.lt"

Butane inherits TraPPE Dreiding {
  ...
}
```

Details:Moltemplate attempts to resolve duplicate atom types or molecule types if they are found in both parents, giving priority to the first parent in the list of parents following the "inherits" keyword. ("TraPPE" in this example. Note: This feature has not been rigorously tested as of 2012-9-28.)

11.8.2 Inheritance *vs.* Nesting

If two molecules are related to each other this way: *"A **is a** particular type of B"*, then consider using inheritance instead of nesting (or object composition). In this example (with *Butane* and *TraPPE*) either nesting or inheritance would work.

Again, one very minor advantage to nesting *Butane* inside *TraPPE*, is that it prevents the name *Butane* from being confused with or conflicting with any other versions of the *Butane* molecule defined elsewhere. (Usually this is not a consideration.)

11.8.3 Inheritance *vs.* Object Composition

On the other hand, if two molecules are related to each other this way: *"A is **comprised of** B and C"*, then you might consider using object composition instead of inheritance. For example:

```
import "B.lt"  # <-- defines the molecule type "B"

import "C.lt"  # <-- defines the molecule type "C"

A {
  b = new B
  c = new C
}
```

12 Known bugs and limitations

Please report any bugs you find by email to **jewett.aij@gmail.com**, or to the lammps-users mailing list.

1) Moltemplate requires a large amount of memory (RAM)

For example, setting up a system of 300000 atoms using moltemplate currently requires 5GB of free memory (as of 2012-10-01). (Memory usage appears to scale linearly with system size.) *I am working to reducing these requirements.*

Meanwhile this problem may be alleviated by using other python interpreters with a lower memory footprint. Also, computers with a moderate amount of RAM can be rented very cheaply. (For example, see `http://cloud.google.com/products/compute-engine.html`.)

When setting up large simulations consider using the "ulimit" command before running moltemplate to prevent system crashes. (If you are on a shared computer, ask an administrator to do this.) If these options are not available, you can always run a resource monitor (like "top") before starting moltemplate and kill the process if it's memory usage exceeds 80%.

2) Limited support for non-point-like atoms:

As of 2012-9-27, only the "full", "angle", "atomic", "charge", and "molecular" styles have been tested. The "dipole" atom style *is* supported but has not been tested. Non-point-like atoms like "ellipsoid", "tri", "line", are not rotated correctly by the ".rot()" command, or scaled correctly by the ".scale()" command. More exotic exotic atom styles, such as "wavepacket", "election", "sphere" and "peri" have not been tested. Feel free to contact the author to request support for new atom styles.

3) Triclinic boundary conditions have not been tested:

As of 2012-9-27, support for PDB files with triclinic cells is experimental. Please let me know if it is not working.

4) Inconsistent support for wildcard characters ("*" and "?")

As of 2012-9-27, wildcard characters ("*" and "?") are interpreted differently in different parts of an LT file. Wildcard characters work reliably and are used for *string* pattern matching when inside any of the *"By Type"* sections in an LT file (such as *"Angles By Type"*, *"Dihedrals By Type"*, and *"Impropers By Type"*). However these characters are interpreted differently when they appear in *pair_coeff*, *bond_coeff*, *angle_coeff dihedral_coeff*, and *improper_coeff* commands (and their corresponding *"Coeff"* sections of a data file). LAMMPS interprets "*" characters appearing in *coeff* commands as *numeric* wildcard characters. This can lead to unintended side-effects and is discouraged. Currently, please avoid "*" characters in *coeff* commands. They can be safely used in array brackets, *[*]*, or in the *"By Type"* sections. (See section 7.5 and appendix A.)

Appendices

A Bonded interactions "By Type"

Interactions between atoms in LAMMPS are normally specified *by atom type*, unless they are directly bonded together. However, as of 2012-3-07, all *bonded interactions*, including 3-body angle, and 4-body dihedral and improper interactions, are specified by unique*by atom ID number*. (There are typically a large number of angles in a typical molecule, and the majority of lines in a typical LAMMPS data file are used to keep track of them.)

This has changed in moltemplate.sh. moltemplate.sh contains a utility which can generate angles, dihedrals, and impropers automatically by atom and bond *type*. (This utility is described in section E.) moltemplate.sh will inspect the network of bonds present in your system, detect all 3-body, and 4-body interactions, and determine their type. (Higher n-body interactions can also be defined by the user.) Specifying interactions this way can eliminate significant redundancy since many atoms share the same type.

To make use of this feature, you would create a new section named "Angles By Type", "Dihedrals By Type", or "Impropers By Type" whose syntax mimics the "Angles", "Dihedrals", and "Impropers" sections of a LAMMPS data file. The syntax is best explained by example:

```
write("Data Angles By Type") {
  @angle:XCXgeneral     * *C* *
  @angle:CCCgeneral   @atom:C @atom:C @atom:C *   *
  @angle:CCCsaturated @atom:C @atom:C @atom:C @bond:sp3 @bond:sp3
}
```

The first line will generate a 3-body angle interaction (of type "@angle:XCXgeneral") between any 3 consecutively bonded atoms as long as the second atom's type-name contains the letter "C". (Atom and bond type-names can contain wildcard characters *)

The second line will generate a 3-body interaction of type "@angle:CCCgeneral" between any 3 atoms of type "@atom:C", regardless of the type of bonds connecting them. (The last two columns, which are both wildcard characters, *, tell moltemplate.sh to ignore the two bond types. Since this is the default behavior these two columns are optional and can be omitted.)

The third line will generate a 3-body interaction of type "@angle:CCCsaturated" between any 3 atoms of type "@atom:C", if they are connected by bonds of type "@bond:sp3".

Note: The 2nd and 3rd lines in this example will generate new interactions which may override any angle interactions assigned earlier.

Regular expressions

Regular-expressions can also be used to match potential atom and bond types. (To use regular expressions, surround the atom and bond types on either side by slashes. For example: @atom:C[1-5]/, should match @atom:C1 through @atom:C6.) *Note: This feature has not been tested as of 2012-9-27.*

In a similar way, one can define "Dihedrals By Type" and "Impropers By Type".

B Using ltemplify.py to create an *LT file*

The "ltemplify.py" script can be used to convert existing simple LAMMPS input script and data files into a single ".lt" file. (Note: As of 2012-2-13, *ltemplify.py is experimental software*, and does not work for every LAMMPS DATA/INPUT file. Known limitations of ltemplify are listed below.)

Example 1

```
ltemplify.py -name Mol file.in file.data > mol.lt
```

This creates a template for a new type of molecule (named "Mol"), consisting of all the atoms in the lammps files you included, and saves this data in a single LT file ("mol.lt"). This file can be used with moltemplate.sh (and/or ttree.py) to define large systems containing this molecule.

Note: The input script ("file.in" in this example) should appear before the data file ("file.data") in the argument list.

In many cases, a LAMMPS data file may contain many copies of the same molecule. In order to select one of these molecules you must manually indicate the atoms which belong to that molecule. To do that, use the following syntax:

Example 2

```
ltemplify.py -name Mol -molid "1" file.in file.data > mol.lt
```

In this example, only atoms belonging to molecule 1 are extracted.

This only works if you are using one of the "molecular" atom_styles. If you are using a different atom_style, you can select the atoms you want either by type or by id number. To do that use the following syntax:

Example 3

```
ltemplify.py -name Mol -atomtype "1 2 3" lammpsfile.in lammpsfile.data > mol.lt
```

In this example, only atoms whose type is 1, 2, or 3 are included.

Example 4

```
ltemplify.py -name Mol -atomid "13 14 15 61*69" \
             lammpsfile.in lammpsfile.data > mol.lt
```

In this example, only atoms whose ids are 13, 14, 15, and 61 through 69 are included.

Limitations:

Limitations: Wildcard characters and ltemplify.py

Again ***coeff*** commands containing "*" characters are risky, especially when processed by ltemplify.py. This practice is discouraged. For example:

```
pair_coeff 1   * 0.15 3.2
pair_coeff 2*3 3 0.05 3.5
```

The only problem here is that, in principle, it is unlikely but possible that once this file has been converted to LAMMPS template format (LT), moltemplate may assign different numbers to these atom types. Although the atom types in each expression will be correctly and uniquely identified, the range of atoms in between may be incorrect. For example, the range from "2*3" in the example above could in principle be replaced with "2*12", if the third atom type in the original file get's assigned a "12". (This only happens if the user makes additional manual changes to the LT file after it was generated.) To be on the safe side, try to avoid using "*" in any of the "_coeff" commands in the input scripts that you pass to ltemplify.py (if possible). Instead represent each interaction explicitly.

```
pair_coeff 1   1 0.15 3.2
pair_coeff 1   2 0.15 3.2
pair_coeff 1   3 0.15 3.2
pair_coeff 2   3 0.05 3.5
pair_coeff 3   3 0.05 3.5
```

(It is a good idea to do this in LT files as well.)

C Advanced moltemplate.sh Usage

moltemplate.sh has several optional command line arguments. These are explained in below:

```
Usage:

moltemplate.sh [-atomstyle style] \
               [-pdb/-xyz coord_file] \
               [-a assignments.txt] file.lt

Optional arguments:

-atomstyle style  By default, moltemplate.sh assumes you are using the "full"
                  atom style in LAMMPS.  You can change the atom style to "dipole"
                  using -atomstyle dipole.  If you are using a hybrid style,
                  you must enclose the list of styles in quotes.  For example:
                  -atomstyle "hybrid full dipole"
                  For custom atom styles, you can also specify the
                  list of column names manually (enclosed in quotes):
```

```
                -atomstyle "molid x y z atomid atomtype mux muy muz"

-xyz xyz_file   An optional xyz_file argument can be supplied as an argument
                following "-xyz".
                This file should contain the atomic coordinates in xyz format.
                (The atoms must appear in the same order in the data file.)

-pdb pdb_file   An optional pdb_file argument can be supplied as an argument
                following "-pdb".

                This should be a PDB file (with ATOM or HETATM records) with
                the coordinates you wish to appear in the LAMMPS data file.
                (The atoms must appear in the same order in the data file.)

                If the PDB file contains periodic boundary box information
                (IE., a "CRYST1" record), this information is also copied
                to the LAMMPS data file.
                (Support for triclinic cells is experimental as of 2012-2-13.
                 Other molecular structure formats may be supported later.)
-a "@atom:x 1"
-a assignments.txt
                The user can customize the numbers assigned to atom, bond,
                angle, dihedral, and improper types or id numbers by using
                   -a "VARIABLE_NAME VALUE"
                for each variable you want to modify.  If there are many
                variables you want to modify, you can save them in a file
                (one variable per line).  For an example of the file format
                run moltemplate.sh once and search for a file named
                "ttree_assignments.txt".  (This file is often located in
                the "output_ttree/" directory.) Once assigned, the remaining
                variables in the same category will be automatically assigned
                to values which do not overlap with your chosen values.
-b assignments.txt
                "-b" is similar to "-a". However, in this case, no attempt
                is made to assign exclusive (unique) values to each variable.
-nocheck
               Normally moltemplate.sh checks for common errors and typos and
               halts if it thinks it has found one.  This forces the variables
               and categories as well as write(file) and write_once(file)
               commands to obey standard naming conventions.  The "-nocheck"
               argument bypasses these checks and eliminates these restrictions.
```

C.1 Manual variables assignment ("-a" or "-b")

It is possible to manually customize the values assigned to the atom types (or to any other ttree-style variables). For example, consider the the "spce.lt" file shown earlier. This file defines a single water molecule with two atom types (hydrogen and oxygen). Typically the "O" atom type is normally

assigned to the integer "1", and "H" would be assigned to "2". This is because "O" appears before "H" in that file. If you wanted to swap the order, you could swap the order in which they first appear.

Alternately you can specify the atom assignments directly using one or more "-a" flags followed by a quoted assignment string:

```
moltemplate.sh -a "@atom:SPCE/O 2" system.lt
```

This assigns the oxygen atom type to "2". Note that quotes are necessary around the '@atom:SPCE/O 2' string, which is a single argument. (Also note that it is necessary to include SPCE/ before the O, because in that example, this atom appeared (and was thus defined) inside the SPCE molecule's environment. Alternately, if it had been defined outside, globally, then you could refer to it using "@atom:O")

Variables need not be assigned to numbers. If for some reason, you want to substitute "a string" everywhere this atom type appears, you would do it this way:

```
moltemplate.sh -a '@atom:SPCE/O "a string"' system.lt
```

Multiple assignments can be made by using multiple "-a" flags:

```
moltemplate.sh -a '@atom:SPCE/O 2'  -a '@atom:SPCE/H 1' system.lt
```

However if you have a large number of assignments to make, it may be more convenient to store them in a file. You can create a two-column text file (for example "new_assignments.txt") and run moltemplate this way:

```
moltemplate.sh -a new_assignments.txt system.lt
```

The contents of the "new_assignments.txt" file in this example would be:

```
@atom:SPCE/O  2
@atom:SPCE/H  1
```

The order of lines in this file does not matter.

Using "-pdb" and "-a" together

If you are using the "-pdb" or "-xyz" flags, these must appear first. The the "-a" (and "-b") flags must appear *at the end* of the argument list (but before the ".lt" file). For example:

```
moltemplate.sh -pdb file.pdb -a '@atom:SPCE/O 2' system.lt
```

Assigning $angle, $dihedral, $improper variables

In general any kind of variable can be assigned this way (not only atom types), including $mol, $bond, @bond, @angle, $angle, ... as well as user-defined variable type. *Caveat: The only occasional exceptions are the $angle, $dihedral, $improper variables.* (When "Angles By Type" interactions are selected by the user, and mixed with regular "Angles", all of the $angle variables are automatically generated. The same is true for "Dihedrals By

Type" and "Impropers By Type". See section E for an explanation of "By Type" interactions.)

Angles, dihedrals, and impropers interactions are automatically generated, and in this case the user does not have the freedom to assign these variables.

The "-b" flag

Note that when using the "-a" flag above, care will be taken to insure that the assignment(s) are exclusive. None of the atom types (other than @atom:SPCE/O) will be assigned "2". (For this reason, using the "-a" flag to change the atom type assignments can, in principle, alter the numbers assigned other atom types, or variables.) This usually the desired behavior. However suppose, for some reason, that you wanted to force a variable assignment, so that other variables in the same category are not effected. In that case, you can use the "-b" flag:

```
moltemplate.sh -b '@atom:SPCE/O 2' system.lt
```

Keep in mind, that in this example, this could cause other atom-types (for example "@atom:SPCE/H") to be assigned to overlapping numbers.

The "ttree_assignments.txt" file

Generally, after running moltemplate.sh, a "ttree_assignments.txt" file will be created (or updated if it is already present) to reflect any changes you made. (This file is usually located in the "output_ttree/" directory. It can also be located the current directory "./".) You can always check this to make sure that the atom types (or any other ttree variables) were assigned correctly.

The "ttree_assignments.txt" file has the same format as the "new_assignments.txt" file example above.

Note: In both files, an optional slash, "/", may follow the "@" or "$" characters, as in "@/atom:SPCE/O". (This slash is optional and indicates the environment in which the counter is defined. The "@atom" counter is defined globally. The "$resid" counter example described in section C.2 is not.)

lttree.py and ttree.py also accept "-a" and "-b" flags

If for some reason, you are using "lttree.py" or "ttree.py" instead of "moltemplate.sh", then the "-a" and "-b" flags explained here also work with these scripts. They are not specific to moltemplate.sh.

C.2 Customizing the counting method using *category*

Variables in ".lt" files are assigned to integers by default, starting with 1, and incrementing by 1. This can be overridden using the "category" command. For example, to create a new variable category named "distance" which starts at 0 and increments by 0.5, you would include this command in your LT file:

```
category $distance(0.0, 0.5)
```

C.3 Creating local independent counters

By default variables in a given category are always assigned to unique integers. This can be overridden using the "category" command. For example, you might have a variable that keeps track of the position index of each residue in each protein chain. The first residue in a protein (N-terminus) is assigned "1", the second residue, "2", etc, *regardless* of the number of protein chains in your system.

To do this, we can create a new variable category named "resid" which is defined within the scope of each instance of the "Protein" molecule:

```
Residue {
  write("Data Atoms") {
    $atom:CA @atom:CA $resid:.  0.0  0.0 0.0 0.0
    $atom:CB @atom:CB $resid:.  0.0  1.53 0.0 0.0
  }
}

Protein {
  category $resid(1,1)
  residues = Residue[100]
}

proteins = Protein[10]
```

In this example, there are 10 proteins containing 100 residues each. The "$resid" counters will be replaced with integers in the range $1 \ldots 100$, (not $1 \ldots 1000$, as you might expect). Because the "$resid" counter is local to the protein it is defined within, "$resid" variables in other proteins do not share the same counter, and can overlap.

C.4 Changing the variable counting order ("-order")

Most variables are assigned automatically. By default static variables (@) are assigned in the order they appear in the file (or files, if multiple LT files are included). Subsequently, instance variables ($) are assigned in the order they are created during instantiation. However you can customize the order in which they are assigned.

Ordering

LT files are parsed by moltemplate.sh/lttree.py in multiple stages. The "write_once()" and "write()" commands are carried out in the static and instance phases respectively, as explained below.

The *static* phase

In the "static" phase, "write_once()" statements are carried out in the order they are read from the user's input file(s) (regardless of whether or not they

appear in nested classes). Any "include" commands will effect this order. After processing the class definitions, and carrying out the "write_once()" commands, lttree.py begins the instantiation phase.

The *instantiation* phase

During this phase, lttree.py makes copies of (instantiates) classes which were requested by the user using the "new" command. During this stage, lttree.py also appends data to files using the "write" command. (In this manual, the "write()" and "new" are called instance commands.) The sequence of alternating "write()" and "new" commands in the order that they appear in the user's input file(s). "new" commands recursively invoke any instance commands for each copy of the class they create.

Static variable ordering (@)

By default, static @ variables are assigned in the order that they appear in the user's input file (after any "include" commands have been carried out). This is true regardless of whether they appear in "write()" or "write_once()" commands, and whether they appear in nested classes. If "-order-dfs" is selected, then static @ variables are defined in the order they appear in the tree, with variables defined in the outermost nested class, (the global class named "/") define first. If this option is selected then static variables defined in "write_once()" commands are assigned to numbers first before any variables in "write()" command are processed. (Position in the input file is used as a secondary sort criteria.) On the other hand, the "-order-file" command line option (described above) does not modify the numeric ordering of static variables (because they are ordered according to file position by default).

Again, the counting of instance variables (prefixed by "$") does not interfere with static variable assignment. For example "@atom:x" and "$atom:x" correspond to different variables and belong to different variable categories ("@atom" and "$atom") and they are assigned to numerical values independently.

D Using *lttree.py* or *ttree.py* directly

(bypassing moltemplate.sh)

"moltemplate.sh" is only a simple script which invokes "lttree.py", and then combines the various output files generated by lttree.py into a single LAMMPS input script and a data file, along with coordinate data. "lttree.py" then invokes "ttree.py". "ttree.py" lacks the ability to read or generate coordinates, but is otherwise nearly identical to "lttree.py" and "moltemplate.sh".

If in the future moltemplate.sh no longer works with some new, recently added LAMMPS feature, you can bypass moltemplate.sh and run lttree.py or ttree.py directly. Everything moltemplate.sh does can essentially be done by hand with a unix shell and a text editor. This procedure is outlined below.

D.1 First run ttree.py

The syntax for running "ttree.py" is identical to the syntax for running moltemplate.sh. The moltemplate.sh syntax is explained above.

Unfortunately, ttree.py does not understand the -pdb or -xyz arguments for processing coordinate data. If you run "ttree.py" directly, then you must extract the coordinate data from these files yourself and insert it into your lammps input files manually. This is explained below.

Example: Go to the examples/waterSPCE/ directory and run:

ttree.py system.lt

This will prepare LAMMPS input files for a system of 32 water molecules. (In this example, we are using the "SPCE" water model.)

Running the command above will probably create the following files: "Data Atoms" (The "Atoms" section of a LAMMPS data file, w/o coordinates) "Data Bonds" (The "Bonds" section of a LAMMPS data file) "Data Angles" (The "Angles" section of a LAMMPS data file) "Data Masses" (The "Masses" section of a LAMMPS data file) "In Init" (The "Initialization" section of a LAMMPS input script.) "In Settings" (The "Settings" section of a LAMMPS input script, which typically contains force-field parameters, group defs, and constraints) "Data Boundary" (The "Periodic Boundary Conditions" section of a LAMMPS data file.) "ttree_assignments.txt" (Variable assignments. See "customization" section.)

This data can be easily combined into a single LAMMPS data file and a single lammps input script later on, using a text editor, or the unix "cat" and "paste" commands.

It may also create these files: "Data Angles By Type", "Data Dihedrals By Type", "Data Impropers By Type". These files tell moltemplate how to automatically generate bonded-interactions by atom and bond type. They must be converted to lists of angles, dihedrals, and impropers, using the "nbody_by_type.py" utility (as explained in appendix A).

D.2 Then create a LAMMPS data file

Create a new file ("system.data" in this example), and paste the following text into it:

Create the "header" section

Example:

```
LAMMPS Description

     96  atoms
     64  bonds
     32  angles
     0  dihedrals

     2  atom types
     1  bond types
```

```
     1  angle types
     0  dihedral types

  0.000000   9.043    xlo xhi
  0.000000   15.663   ylo yhi
  0.000000   7.361    zlo zhi
```

If you use ttree.py, will have to count the number of atoms, bonds, and atom types, bond types etc. yourself.

Note: the numbers in the "xlo xhi" "ylo yhi" "zlo zhi" lines determine the simulation box size, and will vary from system to system. If ttree created a file named "Data Boundary", you can copy this information from there. (Triclinic cells have a fourth line containing the "xy xz yz" parameters.) (If you have a .PDB file, these boundary box numbers are in the "CRYST1" line near the beginning of the file.)

Once you've created the "header" section of the data file, paste the other sections to the end of your LAMMPS data file (with the appropriate section headings and blank lines).

```
echo "" >> system.data
echo "Atoms" >> system.data
echo "" >> system.data
cat "Data Atoms" >> system.data
echo "" >> system.data
echo "Bonds" >> system.data
echo "" >> system.data
cat "Data Bonds" >> system.data
echo "" >> system.data
echo "Angles" >> system.data
echo "" >> system.data
cat "Data Angles" >> system.data
echo "" >> system.data
echo "Masses" >> system.data
echo "" >> system.data
cat "Data Masses" >> system.data
echo "" >> system.data
```

Depending on your system, you may also have these files as well: "Data Dihedrals" "Data Impropers" "Data Bond Coeffs" "Data Angle Coeffs" "Data Dihedral Coeffs" "Data Improper Coeffs". If so, then then append them to the end of your data file as well. (There are numerous other optional sections for "class2" force-fields. Exotic atom styles also require their own sections such as "lines" "ellipsoids" and "triangles". Consult the LAMMPS documentation for details on these as well.)

D.3 Now create the LAMMPS input script

```
echo "include \"In Init\"" > system.in
echo "read_data system.data" >> system.in
echo "include \"In Settings\"" >> system.in
```

Lastly, you have to worry about supplying the atomic coordinates. (Unlike moltemplate, ttree.py does not handle atom coordinates.)

The following commands are useful for extracting coordinates from PDB or XYZ files and converting them to LAMMPS input script commands:

D.4 Extract coordinates

To extract coordinates from a .PDB file ("file.pdb"), use:

```
awk '/^ATOM  |^HETATM/{print substr($0,31,8) \
                          " "substr($0,39,8) \
                          " "substr($0,47,8)}' \
    < file.pdb \
    > tmp_atom_coords.dat
```

(Note: There should be two spaces following the word "ATOM" above.)

To extract coordinates from an XYZ file ("file.xyz"), use:

```
awk 'function isnum(x){return(x==x+0)} \
     BEGIN{targetframe=1;framecount=0} \
     {if (isnum($0)) {framecount++} else \
      {if (framecount==targetframe) { \
       if (NF>0) { \
        if ((NF==3) && isnum($1)) { \
         print $1" "$2" "$3} \
        else if ((NF==4) && isnum($2)) { \
         print $2" "$3" "$4} }}}}' \
    < file.xyz \
    > tmp_atom_coords.dat
```

D.5 Convert the coordinate file to LAMMPS input script format

```
awk '{if (NF>=3) { \
      natom++; print "set atom "natom" x "$1" y "$2" z "$3" "}}' \
    < tmp_atom_coords.dat \
   >> system.in.coords
```

Finally import "system.in.coords" in your lammps input script using:

```
echo "include \"system.in.coords\"" >> system.in
```

E Using the *nbody_by_type.py* utility

(bypassing moltemplate.sh)

moltemplate.sh uses the "nbody_by_type.py" utility to generate many-body interactions between bonded atoms by atom type. In the event that moltemplate.sh crashes or is not up-to-date with LAMMPS, you can assign interactions by type by manually invoking nbody_by_type.py yourself.

As an example, the following command will generate a file "Angles" containing lines of text which should eventually be pasted into the "Angles" section of a LAMMPS data file:

```
nbody_by_type Angles \
    -atoms "Data Atoms" \
    -bonds "Data Bonds" \
    -nbodybytype "Data Angles By Type" \
    > "Data Angles"
```

For dihedral or improper interactions, repeat the command above, and replace "Angles" with "Dihedrals", or "Impropers" everywhere.

Note: The above instructions work assuming that you do not use any wildcard characters ("" or "?") or regular expressions in your "Angles By Type" section. If you use wildcards or regular expressions, then you must run the program this way:*

```
nbody_by_type Angles \
    -atoms "Data Atoms.template" \
    -bonds "Data Bonds.template" \
    -nbodybytype "Data Angles By Type.template" \
    > "Data Angles.template"
```

Afterwards, you must then replace each variable in the "Angles.template" file with the appropriate integer before you copy the contents into the LAMMPS data file. (The ttree_render.py program may be useful for this. Open the moltemplate.sh file with a text editor to see how this was done.)

Note that "Data Atoms", and "Data Bonds" refer to files which are normally created by "ttree.py" or "lttree.py" which contain atom and bond data in LAMMPS data file format, respectively. Similarly "Data Angles By Type" refers to a file containing instructions for how to automatically generate angles by atom type. (Again, this would typically be generated by running "ttree.py" or "lttree.py" on an LT file containing a block of text wrapped inside a "write_once('Data Angles By Type')" command.)

Note: if you already have existing "Data Angles", you can add them to the list of angle interactions created by nbody_by_type.py.

```
nbody_by_type Angles \
    -atoms "Data Atoms" \
    -bonds "Data Bonds" \
    -nbodyfile "Data Angles" \
    -nbodybytype "Data Angles By Type" \
    > extra_Angles.tmp
cat extra_Angles.tmp "Data Angles" > new_Angles
mv -f new_Angles "Data Angles"
rm -f extra_Angles.tmp
```

E.1 Usage

For reference, the complete man page for the "nbody_by_type.py" command is included below.

```
nbody_by_type.py reads a LAMMPS data file (or an excerpt of a LAMMPS)
data file containing bonded many-body interactions by atom type
(and bond type), and generates a list of additional interactions
in LAMMPS format consistent with those type (to the standard out).

Typical Usage:

nbody_by_type.py X < old.data > new.data

--or--

nbody_by_type.py X \
                 -atoms atoms.data \
                 -bonds bonds.data \
                 -nbody X.data \
                 -nbodybytype X_by_type.data
                 > new_X.data

In both cases "X" denotes the interaction type, which
is either "Angles", "Dihedrals", or "Impropers".
(Support for other interaction types can be added by the user. See below.)

-------- Example 1 -------

nbody_by_type.py X < old.data > new.data

In this example, nbody_by_type.py reads a LAMMPS data file
"orig.data", and extracts the relevant section ("Angles",
"Dihedrals", or "Impropers").  It also looks a section named "X By Type",
   (eg. "Angles By type", "Impropers By type", "Impropers By type")
which contains a list of criteria for automatically defining additional
interactions of that type.  For example, this file might contain:

Angle By Type

7 1 2 1 * *
8 2 2 * * *
9 3 4 3 * *

The first column is an interaction type ID.
The next 3 columns are atom type identifiers.
The final 2 columns are bond type identifiers.
The * is a wildcard symbol indicating there is no preference for bond types
in this example.  (Optionally, regular expressions can also be used to
define a type match, by enclosing the atom or bond type in / slashes.)

    The first line tells us to that there should be a 3-body "Angle"
interaction of type "7" whenever an atom of type 1 is bonded to an atom
```

```
of type "2", which is bonded to another atom of type "1" again.
The second line tells us that an angle is defined whenever three atoms
are bonded together and the first two are of type "2".
(Redundant angle interactions are filtered.)

    New interactions are created for every group of bonded
atoms which match these criteria if they are bonded together
in the relevant way for that interaction type (as determined by
nbody_X.py), and printed to the standard output.  For example,
suppose you are automatically generating 3-body "Angle" interactions using:

nbody_by_type Angles < old.data > new.data

The file "new.data" will be identical to "old.data", however the
"Angles By Type" section will be deleted, and the following lines of
text will be added to the "Angles" section:

394 7 5983 5894 5895
395 7 5984 5895 5896
396 7 5985 5896 5897
 :  :   :    :    :
847 9 14827 14848 14849

The numbers in the first column are counters which assign a ID to
every interaction of that type, and start where the original "Angles"
data left off (New angle ID numbers do not overlap with old ID numbers).
The text in the second column ("7", "9", ...) matches the text from the
first column of the "Angle By Type" section of the input file.

-------- Example 2 -------

nbody_by_type.py X \
                 -atoms atoms.data \
                 -bonds bonds.data \
                 -nbody X.data \
                 -nbodybytype X_by_type.data \
                 > new_X.data

In particular, for Angle interactions:

nbody_by_type.py Angles \
                 -atoms atoms.data \
                 -bonds bonds.data \
                 -nbody angles.data \
                 -nbodybytype angles_by_type.data \
                 > new_Angles.data

When run this way, nbody_by_type.py behaves exactly the same way
```

```
    as in Example 1, however only the lines of text corresponding to
    the new generated interactions are printed, (not the entire data file).
    Also note, that when run this way, nbody_by_type.py does not read the
    LAMMPS data from the standard input.  Instead, it reads each section of
    the data file from a different file indicated by the arguments following
    the "-atoms", "-bonds", "-nbody", and "-nbodybytype" flags.

    "Angles" is a 3-body interaction style.  So when run this way,
    nbody_by_type.py will create a 5 (=3+2) column file (new_Angles.data).

Note: the atom, bond and other IDs/types in need not be integers.

Note: This program must be distributed with several python modules, including:
        nbody_Angles.py, nbody_Dihedrals.py, and nbody_Impropers.py.  These
      contain bond definitions for angular, dihedral, and improper interactions.
```

E.2 Custom bond topologies

Currently nbody_by_type.py can detect and generate "Angle" and "Dihedral" interactions between 3 and 4 consecutively bonded atoms. It can also generate "Improper" interactions between 4 atoms bonded with a T-shaped topology (one central atom with 3 branches). The nbody_by_type.py script imports external modules named "nbody_Angles.py", "nbody_Dihedrals.py", and "nbody_Impropers.py" to help it detect angles, dihedrals, and improper interactions automatically. In case any new interaction types are ever added to LAMMPS, it is easy to define new bonded interaction types by supplying a new "nbody_X.py" python modules. These python files are usually only a few lines long. Copy one of the existing modules "nbody_Angles.py", "nbody_Dihedrals.py", or "nbody_Impropers.py") and modify it to the subgraph inside to match the bonded network that you want to search for.

F Variable syntax details

Counter variables have names like $CPATH/CATNAME:LPATH

All counter variables have 3 parts:

CPATH, the category scope object (which is usually omitted)

CATNAME, the category name

LPATH, the "leaf path". This includes the variable's name and (optionally) the location of that variable in the object tree relative to the object in which the variable is referenced (the current-context object)

Typically the CPATH is omitted as in the examples "@atom" "$atom", "$mol". However the CPATH can be specified explicitly as in "$/atom:" ("/" denotes explicitly that the counter has global scope). Another example with an explicit CPATH is the custom local counter variable named "$/proteins[5]/resid:." (See section C.3.) In that example, the CPATH is

"$/proteins[5]", the CATNAME is "resid", and the LPATH is ".". (Even though the CPATH was omitted in section C.3, the meaning is the same. Usually the CPATH does not need to be stated explicitly. See section F.2 below.)

F.1 General variable syntax

The ellipsis ("...") commonly appears in counter variables (or it is implied). The most complex and general variable syntax is:

$CPATH/.../CATNAME:LPATH

This means: find the closest ancestor of the CPATH object containing a category named "CATNAME". This ancestor determines the category's scope. Counter variables in this category are local to ancestors of that object. In this case, LPATH identifies the location of the variable's corresponding "leaf" object relative to the category scope object. On the other hand, if the the category's scope (CPATH) was not explicitly stated by the user, then the LPATH identifies the location of the leaf object relative to the object in which the variable was referenced (the current-context).

F.2 Variable shorthand equivalents

$CATNAME:LPATH is equivalent to "$.../CATNAME:LPATH"

This means: find the closest direct ancestor of the current object containing a category whose name matches CATNAME. If not found, create a new category (at the global level). *This is the syntax used most frequently in LT files.*

If the colon is omitted, as in $LPATH/CATNAME, then it is equivalent to: $CATNAME:LPATH. Again, in these cases, LPATH is a path which is relative to the object in which the variable was referenced.

If $LPATH is omitted, then this is equivalent to $CATNAME:. In other words, the the leaf node is the current node, ".". (This syntax is often used to count keep track of molecule ID numbers. You can use the counter variable "$mol" to keep track of the current molecule id number, because it counts the molecular objects in which this variable was defined. In this case the name of the category is "mol". As in most examples, the category object, CPATH, is not specified. This means the category object is automatically global. A global category object means that every molecule object is given a unique ID number which is unique for the entire system, not just unique within some local molecule. As a counter-example, consider amino acid residue counters. Each amino acid in a protein can be assigned a residue ID number which identifies it within a single protein chain. However because their category was defined locally at the protein level, these residue ID numbers are not global, and are not uniquely defined if there are multiple protein chains present.)

$CPATH/CATNAME:LPATH/...

(SHORTHAND equivalent)

Find the category name and object corresponding to "$CPATH/CATNAME:" (see above) If $CPATH/ is blank, then search for an ancestor with a category whose name matches CATNAME, as described above. To find the variable's corresponding "leaf object", start from the CURRENT object (not the category object). If LPATH is not empty, follow LPATH to a new position in the tree. Otherwise, start at the current object. (An empty LPATH corresponds to the current object.) From this position in the object tree search for a direct ancestor which happens to also be "leaf object" for some other variable which belongs to the desired category. If no such variable is found, then ttree creates a new variable whose leaf object is the object at the LPATH position, and put it in the desired category.

$LPATH/.../CATNAME is equivalent to $CATNAME:LPATH/...

(SHORTHAND equivalent)

If LPATH is omitted, then start from the current node. (In the molecular examples, "$.../mol" is a variable whose category name is "mol". The "leaf object" for the variable is either the current object in which this variable was defined, OR a direct ancestor of this object which has been assigned to a variable belonging to the category named "mol". In this way large objects (large molecules) can be comprised of smaller objects, without corrupting the "mol" counter which keeps track of which molecule we belong to. In other words, "$.../mol" unambiguously refers to the ID# of the large molecule to which this sub-molecule belongs (regardless of however many layers up that may be).)

$CPATH/CATNAME:LPATH

Variables in the output_ttree/ttree_assignments.txt file use the this syntax.

Finally, if the user explicitly specifies the path leading up to the cat node, and avoids using "...", then LPATH is interpreted relative to the category object, not the current object (however CPATH is interpreted relative to the current object). This happens to be the format used in the "ttree_assignments.txt" file (although you can use it anywhere else in an ".LT" file). In "ttree_assignments.txt" file, CPATH is defined relative to the global object. The variables in that file always begin with "$/" or "@/". The slash at the beginning takes us to the global environment object (to which all the other objects belong). (Since the variables in the "ttree_assignments.txt" always begin with "$/" or "@/", this distinction is usually not important because the category object for most variables usually is the "global" root object.)

References

[1] Axel Kohlmeyer. The TopoTools VMD plugin. `http://sites.google.com/site/akohlmey/software/topotools/`.

[2] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14:33–38, 1996.

[3] L. Martnez, R. Andrade, E. G. Brigin, and J. M. Martnez. Packmol: A package for building initial configurations for molecular dynamics simulations. *J. Comp. Chem.*, 30(13):2157–2164, 2009. `http://www.ime.unicamp.br/~martinez/packmol/`.

[4] H. J. C. Berendsen, J. R. Grigera, and T. P. Straatsma. The missing term in effective pair potentials. *J. Phys. Chem.*, 91(24):6269–6271, 1987.

[5] Marcus G. Martin and J. Ilja Siepmann. Transferable potentials for phase equilibria. 1. united-atom description of n-alkanes. *J. Phys. Chem. B*, 102(14):2569–2577, 1998.