Emoltemplate Manual

Andrew Jewett,
Jensen Lab (Caltech), Shea Lab (UCSB)
jewett.aij@gmail.com

February 2, 2017

Emoltemplate is based on Moltemplate, a molecule-building tool for LAMMPS. Similarly, this manual is based on the manual for moltemplate. As such, there may be mistakes. Please report inaccuracies to the author (via email or git).

Contents

1 Introduction

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

A file format has been created to store molecule definitions (the ESPResSo-template format, "ET"). Typical ".ET" files contain atom coordinates, topology data (bonds), ESPResSo force-field data, and other ESPResSo settings (such as 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).

Emoltemplate 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). Memory requirements are discussed in sec:limitations.

1.1 Converting *ET files* to ESPResSo TCL commands

The emoltemplate.sh program converts ET-files (which contain molecule definitions) into a file containing ESPResSo TCL commands:

```
emoltemplate.sh system.et
```

or

```
emoltemplate.sh -xyz coords.xyz system.et
```

In the first example, the coordinates of the atoms in the system are built from commands inside the "system.et" 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 ESPResSo atom styles are supported, including hybrid styles.)

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

Additional tools

The PACKMOL [1] program is useful for generating coordinates of dense heterogeneous mixtures of molecules, which can be read by emoltemplate. (The VMD "solvate" plugin may also be helpful.)

VMD [2] is very useful for visualization. VMD can interactively visualize ESPResSo simulations while they are in progress [3]. VMD also has a number of plugins for generating molecular geometry.

Online examples

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

This manual assumes users have some basic familiarity with ESPResSo.

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

License

Emoltemplate.sh is publicly available at `http://moltemplate.org/espresso` under the terms of the open-source 3-clause BSD license. `http://www.opensource.org/licenses/BSD-3-Clause`

2 Installation

Obtaining Emoltemplate

The source code for emoltemplate is now included with *moltemplate*. To install emoltemplate, you must install moltemplate. However the examples and documentation which come with moltemplate are specific to LAMMPS, not ESPResSo. ESPResSo-specific documentation and examples are now a separate download. Links to *both* the documentation/examples, *as well as* the moltemplate source code can be found at: `http://www.moltemplate.org/espresso/download.html` If you obtained moltemplate or emoltemplate as a .tar.gz file, (as opposed to github), then you can unpack it using:

```
tar -xzvf emoltemplate_2016-12-08.tar.gz
```

(The date will vary from version to version.) Sometimes this archive includes the moltemplate source code. In that case, the directory will contain a subdirectory named "moltemplate" containing the moltemplate source code and python packaging files.

If necessary, move the (outermost) *"moltemplate"* directory to its desired location. (For the sake of this example, let's assume it is located in your home directory. ***Note:*** *This directory should contain the file "**setup.py**"*)

There are two ways to install moltemplate/emoltemplate:

Installation Method 1 (pip)

If you are familiar with pip, then run this command from within the *moltemplate* directory:

```
pip install .
```

Make sure that your default pip install bin directory is in your PATH. (This is usually something like /.local/bin/ or /anaconda3/bin/. If you have installed anaconda, your PATH should have been updated for you automatically.) Later, you can uninstall both moltemplate/emoltemplate using:

```
pip uninstall moltemplate
```

Instructions for updating your PATH are included below.

Installation Method 2

Alternatively, you can edit your PATH variable manually to include the subdirectory where the emoltemplate.sh script is located (typically "moltemplate/moltemplate/scripts/"), as well as the directory containing the most of the python scripts ("moltemplate/"). Suppose the project directory with the README file is named "emoltemplate", and suppose it is located in your home directory:

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

```
export PATH="$PATH:$HOME/moltemplate/moltemplate"
export PATH="$PATH:$HOME/moltemplate/moltemplate/scripts"
```

If instead you use the **tcsh** shell, typically you would edit your ∼/.login, ∼/.cshrc, or ∼/.tcshrc files and append the following lines:

```
seetnv PATH "$PATH:$HOME/moltemplate/moltemplate"
setenv PATH "$PATH:$HOME/moltemplate/moltemplate/scripts"
```

Note: You may need to log out and then log back in again for the changes to take effect.

3 Quick reference *(skip on first reading)*

Note: New users should skip to section 4

3.1 Emoltemplate 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 *SPCEflex*, *Monomer*, and *Butane* molecules, and the *TraPPE* namespace defined in sections 4.1, 6.1, 10.6, & 10.2.1.

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 cumulatively 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.)

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, 10.6, and 10.6.1)
MolType.xform()	Apply the coordinate transform *xform()* to the coordinates of the atoms in all molecules of type *MolType*. (See section 10.
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* instance. (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 Init") are inserted into the relevant section of the ESPResSo TCL 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. (This does not work for atom types. See section 10.5.)

category *\$catname*(i_0, Δ) or **category** *@catname*(i_0, Δ)	Create a new variable category. See section B.2 for details.
create_var { *variable* }	Create a variable specific to this molecule object. (Typically this is used to create molecule-ID numbers, for a molecule built from smaller components. See section 6.1.1.)
replace { *oldvariable newvariable* }	Allow alternate names for the same variable. This replaces all instances of *oldvariable* with *newvariable*. Both variable names must have a "@" prefix. This is typically used to reduce the length of long variables, for example to allow the shorthand "@atom:C2" to refer to "@atom:C2_bC2_aC_dC_iC"
#*commented text*	All text following a "#" character is treated as a comment and ignored.

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:*type*	A number which indicates a *type* of bonded interaction. These numbers can refer to either 2-body, 3-body, or 4-body bonded interactions. These @bond:*type* variables are used to lookup force-field parameters for each type of bond, bond-angle, and dihedral-angle interaction.
The numbers assigned to each variable are saved in the ***output_ttree/ttree_assignments.txt*** *file*	
Advanced variable usage	
\$category:**query**()	Query the current value of the counter in this *\$category* without incrementing it. (The "*\$category*" is usually *\$atom*.) This is useful for counting the number of atoms created so far.
@category:**query**()	Query the current value of the counter in this *@category* without incrementing it. (The "*@category*" is usually *@bond*.) This is useful for counting the total number of bonded interaction types declared so far.)
@**{***category:variable***}** or *\$***{***category:variable***}**	Curly-brackets, **{}**, are used to refer to variables with non-standard delimeters or whitespace characters. (See section 5.5.)
@{category:*type*.rjust(n)} or @{category:*type*.ljust(n)} or \${category:*name*.rjust(n)} or \${category:*name*.ljust(n)}	Print the counter variable in a right-justified or a left-justified text-field of fixed width *n* characters. (This is useful for generating text files which require fixed-width columns.)

See section 5.2 for details.

3.3 Coordinate transformations

(See sections 4.2) and 7.1) for details.)

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.*)
.rotvv$(v_{1x}, v_{1y}, v_{1z}, v_{2x}, v_{2y}, v_{2z})$	Rotate atom coordinates with an angle which rotates the vector $\mathbf{v}_1$ to $\mathbf{v}_2$ (around an axis perpendicular to both $\mathbf{v}_1$ and $\mathbf{v}_2$). If you supply 3 additional numbers x_0, y_0, z_0, the axis of rotation will pass through this location.
.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.)	
push(rot(152.3,0.79,0.43,-0.52)) monomer1 = new Monomer *push*(move(0.01,35.3,-10.1)) monomer2 = new Monomer *pop*() *pop*()	Coordinate transformations introduced using the *push()* command are applied to molecules instantiated later (using the *new*) command, and remain in effect until they are removed using the *pop()* command. (And transformations appearing in arrays accumulate as well, but do not need to be removed with *pop()*.) In this example, the first transformation, "rot()", is applied to both "monomer1" and "monomer2". The last transformation, "move()", is applied after "rot()" and only acts on "monomer2".

3.4 emoltemplate.sh command line arguments:

argument	meaning
-raw coords.raw	Read all of the atomic coordinates from an external RAW file. (RAW files are simple 3-column ASCII files contain X Y Z coordinates for every atom. Numbers are separated by spaces, not commas.)
-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 B.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 B.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 B.1.)
-import-path LOCATION	This allows moltemplate to look for .LT files in other directories when using "import". (Multiple directories must be separated by ':' characters.)

4 Introductory tutorial

Summary

Emoltemplate 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. Emoltemplate is (intentionally) ignorant about ESPResSo and molecular dynamics in general. It is the user's responsibility to understand ESPResSo syntax and write ET files which obey it. For users who are new to ESPResSo, the easiest way to do this is to modify an existing example.

4.1 Simulating a box of water using emoltemplate and ESPResSo

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

Here we show an example of a espresso-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, "ET" files can optionally include any other kind of ESPResSo settings including RATTLE constraints, k-space settings, etc. *(Unicode is supported.)*

```
# file "spce_flex.et"
#
#    H1     H2
#      \   /
#        O
#
# This is stiff but flexible version of the "SPC/E"
# water model that uses short-range electrostatics.
# (Not optimized.  Do not use it in a publication.)

SPCEflex {

  write("Data Atoms") {
    part $atom:o  pos  0.00000 0.000000 0.0000 type @atom:O q -0.8476 mass 16.0
    part $atom:h1 pos  0.81649 0.577359 0.0000 type @atom:H q  0.4238 mass  1.0
    part $atom:h2 pos -0.81649 0.577359 0.0000 type @atom:H q  0.4238 mass  1.0
  }

  write("Data Bonds") {
    part $atom:o  bond @bond:b_OH  $atom:h1
    part $atom:o  bond @bond:b_OH  $atom:h2
  }
  # 2-body non-bonded interactions:

  write_once("In Settings") {
    inter @atom:O @atom:O lennard-jones 0.1553 3.166 10.0
    inter @atom:O @atom:H lennard-jones 0.0    3.166 10.0
    inter @atom:H @atom:H lennard-jones 0.0    3.166 10.0
  }

  # 2-body (bonded) interactions:
  #
  #   Ubond(r) = (k/2)*(r-r0)^2
  #
  #           bond_type       bond_Style   k     r0
  write_once("In Settings") {
    inter  @bond:b_OH         harmonic    600.0  1.0
  }


  # 3-body interactions in this example are of this type:
  #
  # Uangle(theta) = (k/2)*(theta-theta0)^2
  #
  #     (k in kcal/mol/rad^2, theta0 in radians)
  #
  # inter  angleType  stylename  k     theta0

  write_once("In Settings") {
    inter @bond:a_HOH   angle   600.0 1.91061193216
  }

  write("Data Angles") {
    part $atom:o  bond  @bond:a_HOH  $atom:h1  $atom:h2
  }

} # SPCEflex
```

Comment: Currently, in an ".ET" file, the TCL commands that describe a molecule must be divided into different sections, such as "Data Atoms", "Data Bonds", and "In Settings" See section 5.4 for details.

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.et" --
import "spce_flex.et"
```

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

You can now use “emoltemplate.sh” to create simulation input files for ESPResSo

```
emoltemplate.sh -pdb coords.pdb system.et
```

This command will create espresso input files for the molecular system described in “system.et”, using the desired atom style (“full” by default). In this example, emoltemplate 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 “.et” 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 emoltemplate (which is determined by the order of the “new” commands in the ET 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 ET file itself (discussed in section 6). For example you can replace the line:

```
wat = new SPCEflex [1000]
```

from the example above with 1000 lines:

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

Specifying geometry this way is tedious. Alternatively, emoltemplate 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 SPCEflex [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, and bondIDs (if present), for any molecule elsewhere in your ET files using this notation: “$atom:wat[2][3][4]/h1” 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:

ESPResSo 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") {
  setmd box 34.5 34.5 34.5
  setmd periodic 1 1 1
}
```

This is usually specified in the outermost ET file (“system.et” in this example).

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

```
emoltemplate.sh system.et
```

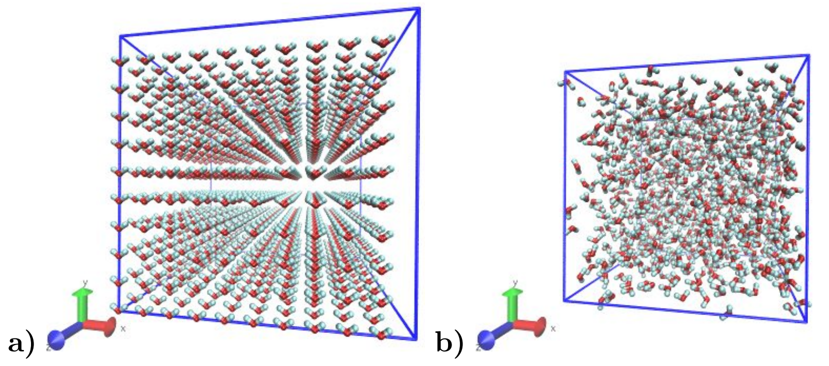

Figure 2: A box of 1000 water molecules (before and after pressure equilibration), generated by emoltemplate and visualized by VMD.

4.3 Running a ESPResSo simulation (after using emoltemplate)

Emoltemplate will create the following file: “system.tcl” This file contains TCL commands for creating atoms, bonds, and other interactions in ESPResSo. They can be run in ESPResSo directly.

To *run* a simulation, you will have to edit this file in order to add additional commands to tell ESPResSo 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).

5 Overview

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

Each ET 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:", "@atom:", "@bond:".) Variables belonging to different categories are counted independently.

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

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 ET 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 *bonded-interaction types.* Later, ESPResSo will use this integer to lookup the bond-length and Hooke's-law elastic constant describing the force between two atoms, and the angle between three atoms, etc...

5.2.2 Instance counters begin with "$"

On the other hand, "$" variables are created whenever a copy of a molecule is created (using the "new" command). (These are usually atom-ID numbers.) If you create 1000 copies of a water molecule using a command like

```
wat = new SPCEflex[10][10][10]
```

then emoltemplate 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:SPCEflex/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:SPCEflex/O" and "@atom:SPCEflex/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 "SPCEflex" 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:SPCEflex/H", and "@atom:Arginine/H").

Sharing atom types or other variables between molecules

There are several ways to share atom types between two molecules. The *recommended way* is to define them in a separate file and refer to them when needed. This approach is demonstrated in section 6.1.

(Alternately, you can define them outside the current molecule definition, and 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". For details, see section 10.4. and appendix E.)

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 emoltemplate. Unfortunately, it is typical for ESPResSo to crash the first time you attempt to run it on a TCL file created by emoltemplate. This often occurs if you failed to spell atom types and other variables consistently. The ESPResSo error message 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. (There is also a comment on each line beginning with a "#" character which indicates the file and line number where this variable is first used.) 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 ET files.

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

5.4 "Data" and "In"

All files whose names begin with "In " or "Data " are special. The emoltemplate.sh script copies the contents of these files into the final TCL file in a different order than the order the commands were issued. Text written to "In Init" and "In Settings" (which usually contain force-field parameters) appears in your TCL file before text written to files with names like "Data Atoms", "Data Bonds", "Data Angles", "Data Dihedrals", "Data Angles By Type", and "Data Dihedrals By Type", for example. (These files contain coordinate data, bonds, angles, dihedrals, as well as rules for generating angles and dihedrals, respectively. Emoltemplate recognizes these files, and treats them differently.) Afterwards these files are moved to the "output_ttree/" directory, in an effort to clean things up and hide them from view. (But theese files are not discarded. If there is an error in your files, the "output_ttree/" directory is a good place to find it.)

More generally, the "write()" and "write_once()" commands can be used to create any other files you may need to run your simulations which refer to the same *@atom* and *@bond* types. (See section 5.5 for an example.)

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

The following excerpt from an ET file creates a file named "table_XCCX.dat", containing 3-column (angle, F, V) data for a tabulated dihedral interaction. It then refers to this file later on.

```
write_once("table_XCCX.dat") {
  # 12
  0.0000000000000000 -0.8000000000000000  0.3000000000000000
  0.5235987755982988  0.14999999999999997 1.0598076211353318
  1.0471975511965976  1.0598076211353316  0.1500000000000000
  1.5707963267948966  0.2999999999999997  -0.800000000000000
  2.0943951023931953 -0.5401923788646684 -0.15000000000000008
  2.6179938779914944  0.1500000000000005  0.5401923788646684
  3.141592653589793   0.8000000000000000 -0.3000000000000000
  3.665191429188092  -0.1499999999999996 -1.0598076211353318
  4.1887902047863905 -1.0598076211353316 -0.15000000000000047
  4.71238898038469   -0.3000000000000004  0.8000000000000000
  5.235987755982989   0.5401923788646685  0.1500000000000000
  5.759586531581287  -0.14999999999999822 -0.5401923788646685
}
write_once("In Settings") {
  inter @bond:xccx tabulated dihedral table_XCCX.dat
}
write_once("Data Dihedrals By Type") {
  @bond:xccx @atom:* @atom:C @atom:C @atom:* @bond:* @bond:* @bond:*
}
```

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

Referencing TCL variables *inside* an .ET file:

The $ character is used to denote ***both*** *TCL variables* ***and*** *emoltemplate variables. Users can include references to TCL variables in their .ET files, but they must precede these them with a "\" character so that emoltemplate does confuse them with its own. (For example, to refer to TCL variable* ***x*** *in a write() statement, you must use* **\$x**, *not* ***$x****)*

Does "@atom:H" conflict with "$atom:H"?

No. It is okay for static(@) and instance($) variables to share the same names. (Moltemplate considers them distinct variables and they will be assigned independently.)

Addional Details

Variable and molecule names can include unicode characters. They can also include some whitespace characters and other special characters by using backslashes and curly-brackets, for example: “@{atom: CA }” and “@atom:\ CA\ ”. Curly-brackets are useful to clarify when a variable name begins and ends, such as in this example: “@{atom:C}-@{atom:H}”. (This prevents the “-” character from being appended to the end of the “C” variable name.)

(Unicode is supported.)

6 Object composition and coordinate generation

Objects can be connected together to form larger molecule objects. These objects can be used to form still larger objects. As an example, we define a small 2-atom molecule named “Monomer”, and use it to construct a short polymer (“Peptide”).

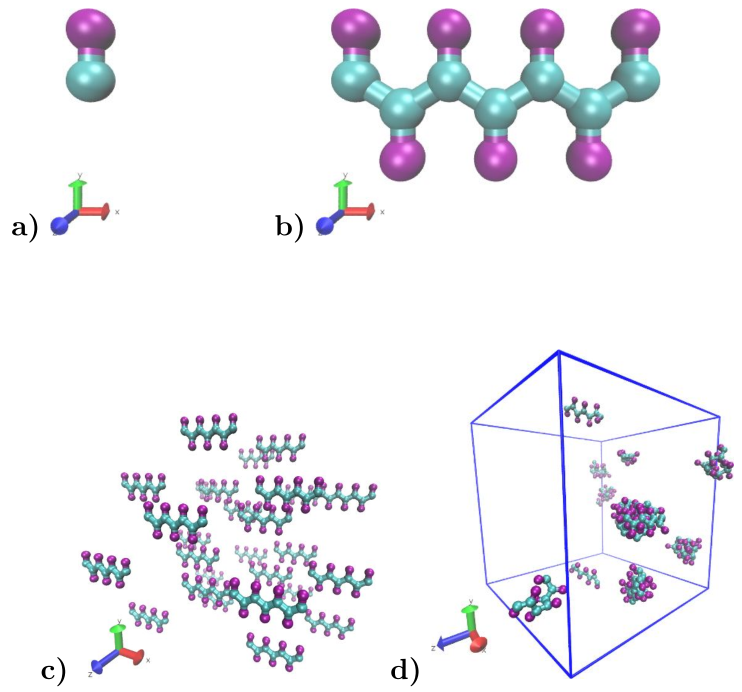

Figure 3: **a)-b)** *Building a complex system from small pieces:* Construction of a polymer (**b**) out of smaller (2-atom) subunits (**a**) using composition and rigid-body transformations. Bonds connecting different residues together (blue) must be declared explicitly, but angle and dihedral interactions 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.

6.1 Building a large molecule from smaller pieces

```
# -- file "monomer.et" --

import "forcefield.et"   # contains force-field parameters

Monomer inherits ForceField {
  write("Data Atoms") {
    part $atom:ca  pos 0.000 1.000 0.000  type @atom:CA  q 0.0  mass 13.0
    part $atom:r   pos 0.000 4.400 0.000  type @atom:R   q 0.0  mass 50.0
  }
  write("Data Bonds") {
    part $atom:ca bond @bond:b_Sidechain $atom:r
  }
}
```

In this example we will define two kinds of molecule objects: “Monomer”, and “Peptide” (*defined later*). It is often convenient to store atom types, masses, and force-field parameters in a separate file so that they can be shared between these different molecules. We do that in the “forcefield.lt” file below:

```
# -- file "forcefield.et" --

ForceField {

  # There are 2 atom types: "CA" and "R"

  # 2-body non-bonded interactions:
  #
  #   Uij(r) = 4*eps_ij * ( (sig_ij/r)^12 - (sig_ij/r)^6 )
  #
  #            i        j          stylename     eps   sig  rcut
  #
  write_once("In Settings") {
    inter   @atom:CA  @atom:CA   lennard-jones  0.10   2.0  9.0
    inter   @atom:R   @atom:R    lennard-jones  0.50   3.6  9.0
    inter   @atom:CA  @atom:R    lennard-jones  0.2236 2.8  9.0
  }

  # 2-body (bonded) interactions:
  #
  #   Ubond(r) = (k/2)*(r-r0)^2
  #
  #          bond_type       bond_Style    k      r0
  write_once("In Settings") {
    inter  @bond:b_Sidechain  harmonic    30.0   3.4
    inter  @bond:b_Backbone   harmonic    30.0   3.7
  }
```

```
# Although the simple "Monomer" object we defined above has only
# two atoms, later on, we will create molecules with many bonds.
# By convention, in this file we keep track of all of the possible
# interactions which could exist between these atoms:

# 3-body interactions in this example are listed by atomType and bondType
# Rules for determining 3-body angle interactions by type
# angle-type     atomType1 atomType2 atomType3  bondType1 bondType2

write_once("Data Angles By Type") {
  @bond:a_Backbone  @atom:CA @atom:CA  @atom:CA     *        *
  @bond:a_Sidechain @atom:CA @atom:CA  @atom:R      *        *
}

# Uangle(theta) = (k/2)*(theta-theta0)^2
#     (k in kcal/mol/rad^2, theta0 in radians)
#
# The corresponding command is:
#
#          angleType        angle    k     theta0

write_once("In Settings") {
  inter  @bond:a_Sidechain  angle   30.0   1.9896753  #114 degrees
  inter  @bond:a_Backbone   angle   30.0   2.3038346  #132 degrees
}

# Rules for determining 4-body dihedral interactions by type
# dihedralType atmType1 atmType2 atmType3 atmType4  bondType1 bnd2 bnd3

write_once("Data Dihedrals By Type") {
  @bond:d_CCCC @atom:CA @atom:CA @atom:CA @atom:CA      *     *    *
  @bond:d_RCCR @atom:R  @atom:CA @atom:CA @atom:R       *     *    *
}

# 4-body interactions in this example are listed by atomType
# The forumula used is:
#
# Udihedral(phi) = K * (1 + cos(n*phi - d))
#
#     The d parameter is in radians, K is in kcal/mol/rad^2.
#
# The corresponding command is
# inter  dihedralType    dihedral       K  n  d

write_once("In Settings") {
  inter  @bond:d_CCCC    dihedral     -0.5 1 3.141592653589793
  inter  @bond:d_RCCR    dihedral     -1.5 1 3.141592653589793
  }

} # ForceField
```

6.1.1 Building a simple polymer

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

```
# -- file "peptide.et" --

import "monomer.et"

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

  # Now, link the residues together this way:
  write("Data Bonds") {
    part  $atom:res1/ca  bond @bond:b_Backbone  $atom:res2/ca
    part  $atom:res2/ca  bond @bond:b_Backbone  $atom:res3/ca
    part  $atom:res3/ca  bond @bond:b_Backbone  $atom:res4/ca
    part  $atom:res4/ca  bond @bond:b_Backbone  $atom:res5/ca
    part  $atom:res5/ca  bond @bond:b_Backbone  $atom:res6/ca
    part  $atom:res6/ca  bond @bond:b_Backbone  $atom:res7/ca
  }
}
```

The position and orientation of each copy of "Monomer" 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 "Monomer" 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.

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")" command) Instead we provided emoltemplate.sh with instructions to help it figure out which atoms participate in 3-body and 4-body bonded interactions. Emoltemplate 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 "forcefield.et" using the "write_once("Data Angles By Type")" and "write_once("Data Dihedrals By Type")" commands. *(More general interactions are possible. See appendix D.2.)*

7 Arrays and coordinate transformations

Emoltemplate 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 **??**. You can make 7 copies of the *Monomer* molecule this way:

```
res = new Monomer[7]
```

This creates 7 new *Monomer* 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 **??** using this command:

```
res = new Monomer [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 Monomer.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 *Monomer* 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 Monomer [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

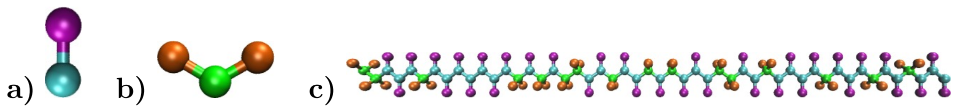

Figure 4: A random heteropolymer (**c**), composed of of *2bead* and *Monomer3* monomers (**a** and **b**) in a 3:2 ratio.

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 *Monomer3* monomers. (See figure 4.)

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

  monomers = new random([Monomer, Monomer3], [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") {
    part  $atom:monomers[0]/ca  bond @bond:b_Backbone  $atom:monomers[1]/ca
    part  $atom:monomers[1]/ca  bond @bond:b_Backbone  $atom:monomers[2]/ca
    part  $atom:monomers[2]/ca  bond @bond:b_Backbone  $atom:monomers[3]/ca
    part  $atom:monomers[3]/ca  bond @bond:b_Backbone  $atom:monomers[4]/ca
      ⋮
    part  $atom:monomers[48]/ca  bond @bond:b_Backbone  $atom:monomers[49]/ca
  }
  #(Note: Both the "Monomer" and "Monomer3" molecules contain atoms
  #       named "$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 (*Monomer* and *Monomer3* 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 emoltemplate.) 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([Monomer.move(0,0.01,0),
                         Monomer3.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 ET file.

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

```
Dimer inherits ForceField {
  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") {
  part $atom:dimer/peptide[0]/res[2]/ca  pos 6.4 8.2 0.6
}
```

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

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

```
delete dimer/peptide[0]/res[3]/ca #<-deletes $atom:ca in dimer/peptide[0]/res[3]
delete dimer/peptide[1]/res[4]/r  #<-deletes $atom:r  in dimer/peptide[1]/res[4]
```

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.)*

Multiple molecules or atoms 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
```

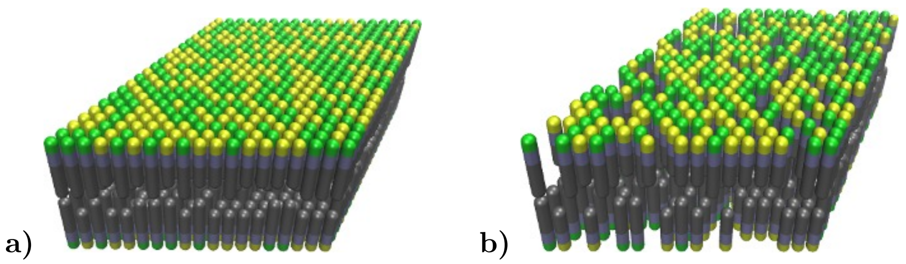

Figure 5: A lipid bilayer membrane composed of a random equal mixture of two different lipid types in a 1:1 ratio. (See section 9.2.) In **b)** one of the molecule types was left blank leaving vacancies behind. (See section 9.3.)

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 entries using [*] or [i-j] notation should be 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.)*

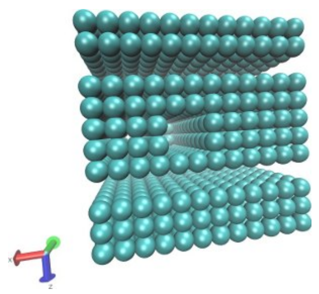

Figure 6: Rectangular holes can be carved out of an array of molecules (represented here by blue spheres) using the "delete" command. Three delete commands were used to remove the two planar regions and the rectangular hole in the center.

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 inherits ForceField {
  write("Data Atoms") {
    part $atom:t1  pos  23.0  0.0   0.0  type @atom:CA  q 0.0  mass 13.0
    part $atom:t2  pos  24.7  4.0   0.0  type @atom:CA  q 0.0  mass 13.0
    part $atom:t3  pos  24.7  8.4   0.0  type @atom:CA  q 0.0  mass 13.0
    part $atom:t4  pos  23.0  12.4  0.0  type @atom:CA  q 0.0  mass 13.0
  }
  write("Data Bonds") {
    part $atom:peptides[0]/res7/CA  bond @bond:b_Backbone  $atom:t1
    part $atom:t1      bond @bond:b_Backbone  $atom:t2
    part $atom:t2      bond @bond:b_Backbone  $atom:t3
    part $atom:t3      bond @bond:b_Backbone  $atom:t4
    part $atom:t4      bond @bond:b_Backbone  $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).

Note1: Coordinate transformations applied to entire molecule types are an experimental feature as of 2012-10-18. This feature has not been rigorously tested.

*Note2: These coordinate transformations will be applied **after** the molecule is completely constructed, (If you add atoms to the molecule, these will be added before the coordinate transformations are applied, even if you issue the command later.) 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 ESPResSo 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.

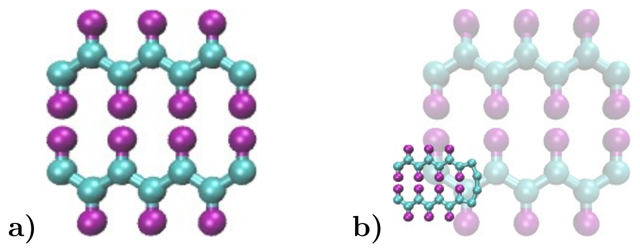

Figure 7: **a)** The "Dimer" molecule. This is a contrived example consisting of two "Peptides". See section 6.1.1 **b)** A customized version of the "Dimer" molecule. (The original "Dimer" is shown faded in the background for comparison.)

(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 emoltemplate and are discussed in section 10.6.

(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 10.6 mentions one way to do this and section 10.6.3 discusses alternate approaches.

Advanced emoltemplate usage

10.1 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). ET 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 ET files. Suppose two of the ET files both happen to contain definitions for "Water". Emoltemplate 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 10.3 for an example.

10.2 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.et”.

```
import "trappe1998.et"

cyclopentane {

  write("Data Atoms") {
    part $atom:c1 pos 0 0.00000000 1.0000000 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c2 pos 0 0.95105652 0.3090170 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c3 pos 0 0.58778525 -0.809017 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c4 pos 0 -0.5877853 -0.809017 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c5 pos 0 -0.9510565 0.3090170 type @atom:TraPPE/CH2 q 0.0 mass 14
  }

  write("Data Bonds") {
    part $atom:c1  bond @bond:TraPPE/CC  $atom:c2
    part $atom:c2  bond @bond:TraPPE/CC  $atom:c3
    part $atom:c3  bond @bond:TraPPE/CC  $atom:c4
    part $atom:c4  bond @bond:TraPPE/CC  $atom:c5
    part $atom:c5  bond @bond:TraPPE/CC  $atom:c1
  }
}
```

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

```
# 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 SPCEflex 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.et” 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 10.3.) The important parts of the “trappe1998.et” file are shown below:

10.2.1 Namespace example

```
# -- file "trappe1998.et" --

TraPPE {
  write_once("In Settings") {
    inter @bond:CC          harmonic      120.0   1.54
    inter @bond:CCC         harmonic      62.0022 114
    inter @bond:CCCC        opls          1.411036 -0.271016 3.145034 0.0
    inter @atom:CH2 @atom:CH2 lennard-jones 0.091411522 3.95 10.0
    inter @atom:CH3 @atom:CH3 lennard-jones 0.194746286 3.75 10.0
    # (Interactions between different atom types use mixing rules.)
  }
  write_once("Data Angles By Type") {
  @bonid:CCC @atom:C* @atom:C* @atom:C* @bond:CC @bond:CC
  }
  write_once("Data Dihedrals By Type") {
@bond: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.

WARNING: BROKEN EXAMPLE. This example was converted from another format into ET format. The example above uses "opls" dihedral style which does not exist in ESPResSo. *The "opls" force-field allows contains a 4-body dihedral interaction potential which is the sum of two sinusoidal functions. When I learn ESPResSo well enough, I will replace the command above with something more relevant. (Tabulated potentials?) I'll worry about this later. -Andrew 2012-10-18*

Bonded interactions *by type*

Again, the "Data Angles By Type" and "Data Dihedrals By Type" sections tell emoltemplate.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 "SPCEflex" 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 10.3.)

10.3 Nested molecules

Earlier in section 10.2.1, 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.et" --

import "trappe1998.et"

Butane {
  write("Data Atoms"){
    part $atom:c1 pos 0.41937 0.00 -1.937329 type @atom:TraPPE/CH3 q 0.0 mass 15
    part $atom:c2 pos -0.41937 0.0 -0.645776 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c3 pos 0.41937 0.00  0.645776 type @atom:TraPPE/CH2 q 0.0 mass 14
    part $atom:c4 pos -0.41937 0.00 1.937329 type @atom:TraPPE/CH3 q 0.0 mass 15
  }
  write("Data Bonds"){
    part $atom:c1  bond @bond:TraPPE/CC  $atom:c2
    part $atom:c2  bond @bond:TraPPE/CC  $atom:c3
    part $atom:c3  bond @bond:TraPPE/CC  $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.et" --

import "trappe1998.et"

TraPPE {
  Butane {
    write("Data Atoms"){
      part $atom:c1 pos 0.41937 0.00 -1.937329 type @atom:../CH3 q 0.0 mass 15
      part $atom:c2 pos -0.41937 0.0 -0.645776 type @atom:../CH2 q 0.0 mass 14
      part $atom:c3 pos 0.41937 0.00  0.645776 type @atom:../CH2 q 0.0 mass 14
      part $atom:c4 pos -0.41937 0.00 1.937329 type @atom:../CH3 q 0.0 mass 15
    }
    write("Data Bonds"){
      part $atom:c1  bond @bond:../CC  $atom:c2
      part $atom:c2  bond @bond:../CC  $atom:c3
      part $atom:c3  bond @bond:../CC  $atom:c4
    }
  }
}
```

Note: Wrapping Butane within "TraPPE{ }" clause merely appends additional content to be added to the "TraPPE" object defined in the "trappe1998.et" file (which was included earlier). It does not overwrite it. Again "../" tells emoltemplate use the "CH2" atom defined in the context of the TraPPE environment (IE. one level up). This insures that emoltemplate 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.et"
new butane = TraPPE/Butane
```

(You don't need to import "trappe1998.et" in this example because it was imported within "trappe_butane.et".) The "TraPPE/" prefix before "Butane" lets emoltemplate/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 10.6 and 10.6.1.*

10.4 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).

10.5 *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.et"

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.et"

using namespace TraPPE

butane = new Butane
```

This only works for molecule types, not atom types

Unfortunately, you still *must* always **refer to** atom types, bonded-interaction 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.et"
using namespace TraPPE
butane = new Butane
write("Data Atoms") {
  part $atom:c1 pos  0.41937 0.00 1.937329  type @atom:TraPPE/CH3 q 0.0 mass 15
  part $atom:c2 pos -0.41937 0.00 -0.645776 type @atom:CH2        q 0.0 mass 14
}
```

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 10.6.)

10.6 Inheritance

We could have defined *Butane* this way:

```
import "trappe1998.et"

Butane inherits TraPPE {
  write("Data Atoms"){
    part $atom:c1 pos 0.41937 0.00 -1.937329 type @atom:CH3 q 0.0 mass 15
    part $atom:c2 pos -0.41937 0.0 -0.645776 type @atom:CH2 q 0.0 mass 14
    part $atom:c3 pos 0.41937 0.00  0.645776 type @atom:CH2 q 0.0 mass 14
    part $atom:c4 pos -0.41937 0.00 1.937329 type @atom:CH3 q 0.0 mass 15
  }
  write("Data Bonds"){
    part $atom:c1  bond @bond:CC  $atom:c2
    part $atom:c2  bond @bond:CC  $atom:c3
    part $atom:c3  bond @bond:CC  $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.)

10.6.1 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.et"
import "dreiding1990.et"

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

Details:Emoltemplate 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 2012-10-18.)

10.6.2 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.)

10.6.3 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.et"  # <-- defines the molecule type "B"

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

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

11 Known bugs and limitations

Please report any bugs you find by email to **jewett.aij@gmail.com**

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 2012-12-04). (Memory usage appears to scale linearly with system size.) Python programs can require more than 20 times as much memory as similar programs written in C/C++. *(I wish I had known this earlier.)* There are several simple tricks available to reduce memory usage in python. *I hope to try these eventually if I have time.*

Meanwhile this problem might 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`.) Alternately, it may be necessary to split a large system into pieces, run moltemplate on each piece, and combine the resulting data files into one large data file later. A strategy for combining data files together is discussed in appendix B.3.

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

As of 2012-12-01, only point-like particles have been tested. Non-point like particles like dipoles and ellipsoids are probably not rotated correctly.

3) Triclinic boundary conditions have not been tested:

As of 2012-12-04, support for PDB files with triclinic cells is experimental. Please let me know if it is not working.

4) When placed at the end of a line, TCL interprets the "
" character as a request to merge two lines together. *It is usually safe to use this character inside emoltemplate write() or write_once() commands.* However in some rare cases, joining two lines together using the "
" character can confuse emoltemplate.

Appendices

A Bonded interactions "By Type"

Interactions between atoms in ESPResSo are normally specified *by atom type*, unless they are directly bonded together. However, as of 2012-10-18, 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 ESPResSo TCL file are used to keep track of them.)

This has changed in emoltemplate.sh. emoltemplate.sh contains a utility which can generate angles, dihedrals, and impropers automatically by atom and bond *type*. (This utility is described in section D.) emoltemplate.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 "Data Angles By Type", "Data Dihedrals By Type", or "Data Impropers By Type" 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:SAT @bond:SAT
}
```

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

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 2012-10-18.*

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

B Advanced emoltemplate.sh Usage

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

```
Usage:

emoltemplate.sh [-pdb/-xyz coord_file] \
                [-a assignments.txt] file.et

Optional arguments:

-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 be created in the same order in the .ET 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 ESPResSo 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 ESPResSo data file.
                (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 emoltemplate.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 emoltemplate.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.
                    Note: this argument must appear first in the list, for example:
                    emoltemplate -nocheck -pdb f.pdb -a "$atom:res1/ca 1" system.et
```

B.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_flex.et" 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:

```
emoltemplate.sh -a "@atom:SPCEflex/O 2" system.et
```

This assigns the oxygen atom type to "2". Note that quotes are necessary around the '@atom:SPCEflex/O 2' string, which is a single argument. (Also note that it is necessary to include SPCEflex/ before the O, because in that example, this atom appeared (and was thus defined) inside the SPCEflex 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:

```
emoltemplate.sh -a '@atom:SPCEflex/O "a string"' system.et
```

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

```
emoltemplate.sh -a '@atom:SPCEflex/O 2' -a '@atom:SPCEflex/H 1' system.et
```

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 emoltemplate this way:

```
emoltemplate.sh -a new_assignments.txt system.et
```

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

```
@atom:SPCEflex/O  2
@atom:SPCEflex/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 ".et" file). For example:

```
emoltemplate.sh -pdb file.pdb -a '@atom:SPCEflex/O 2' system.et
```

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:SPCEflex/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:

```
emoltemplate.sh -b '@atom:SPCEflex/O 2' system.et
```

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

The "ttree_assignments.txt" file

Generally, after running emoltemplate.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 "$" characters, as in "@/atom:SPCEflex/O". (This slash is optional and indicates the environment in which the counter is defined. The "@atom" counter is defined globally. The "$resid" counter example described in section B.2 is not.)

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

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

B.2 Customizing the counting method using *category*

Variables in ".et" 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 ET file:

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

(This command can also be used with traditional counter categories like *$atom* and *bond*).

B.3 Combining files together

This is useful if you are combining data files from two systems together. For example if a previous system contains 317982 atoms, then the next time you run emoltemplate, you would insert the following text at the beginning of your ET file (system.et)

```
category $atom(317983, 1)
```

This will avoid overwriting the settings for these atoms in the previous system. If you need help to combine a large number of systems together, contact **jewett.aij@gmail.com** and we can work on an automated solution. I would like to eventually see emoltemplate be used for large systems.)

B.4 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") {
    part $atom:ca pos 0.0   0.0 0.0 0.0 type @atom:C  q 0.0 mass 13
    part $atom:cb pos 0.0  1.53 0.0 0.0 type @atom:C  q 0.0 mass 14
  }
  write("AuxiliaryFile") {
    atom# $atom:ca belongs to residue# $resid:. of protein# $mol:...
    atom# $atom:cb belongs to residue# $resid:. of protein# $mol:...
  }
}

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

  create_var { $mol:. } # <- creates a $mol counter variable for this protein
                        #    "$mol:..." above will refer to this counter
}
```

```
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.

B.5 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 ET 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

ET files are parsed by emoltemplate.sh/ettree.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, ettree.py begins the instantiation phase.

The *instantiation* phase

During this phase, ettree.py makes copies of (instantiates) classes which were requested by the user using the "new" command. During this stage, ettree.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.

C Using *ettree.py* or *ttree.py* directly

(bypassing emoltemplate.sh)

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

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

C.1 First run ttree.py

The syntax for running "ttree.py" is identical to the syntax for running emoltemplate.sh. The emoltemplate.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 ESPResSo input files manually. This is explained below.

Example: Go to the "waterSPCE+Na+Cl" directory (in the examples) and run:

ttree.py system.et

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

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

This data can be easily combined into a single ESPResSo TCL file 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 emoltemplate 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).

C.2 Then create a ESPResSo TCL file

Create a new file (“system.tcl” in this example), and append the following files to it.

```
echo "" >> system.data
echo "# -- Init --" >> system.data
echo "" >> system.data
cat "In Init" >> system.data
echo "" >> system.data

echo "# -- Bond force-field parameters --" >> system.data
echo "" >> system.data
cat "Data Bond Coeffs" >> system.data
echo "" >> system.data


echo "# -- Angle force-field parameters --" >> system.data
echo "" >> system.data
cat "Data Angle Coeffs" >> system.data
echo "" >> system.data


echo "# -- Dihedral force-field parameters --" >> system.data
echo "" >> system.data
cat "Data Dihedral Coeffs" >> system.data
echo "" >> system.data
echo "# -- Impropre force-field parameters --" >> system.data
echo "" >> system.data
cat "Data Improper Coeffs" >> system.data
echo "" >> system.data
echo "" >> system.data
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 "# -- End of system definition --" >> 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.

C.3 Extract coordinates

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

C.4 Convert the coordinate file to ESPResSo 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
```

D Using the *nbody_by_type.py* utility

(bypassing emoltemplate.sh)

emoltemplate.sh uses the “nbody_by_type.py” utility to generate many-body interactions between bonded atoms by atom type. In the event that emoltemplate.sh crashes or is not up-to-date with ESPResSo, 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 ESPResSo 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 ESPResSo data file. (The ttree_render.py program may be useful for this. Open the emoltemplate.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 “ettree.py” which contain atom and bond data in ESPResSo 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 “ettree.py” on an ET 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
```

D.1 Usage

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

```
    nbody_by_type.py reads a ESPResSo data file (or an excerpt of a ESPResSo)
    data file containing bonded many-body interactions by atom type
    (and bond type), and generates a list of additional interactions
    in ESPResSo 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 ESPResSo 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
    ESPResSo 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.
```

D.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 ESPResSo, 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.

E Variable syntax details

Counter variables have names like:

$***cpath*/*catname*:*lpath***
or
@***cpath*/*catname*:*lpath***
(Note: All of the variable examples in this appendix can refer to either static @ variables or instance $ variables. Both variable types obey the same syntax rules. For brevity, only the instance $ variables are shown.)
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, in which case it means that the category has global scope. *(This is true for all of the standard counter variable types: "@atom", "$atom", "$mol", "@bond", "$bondid", "@angle", "@dihedral", and "@improper".)* However the ***cpath*** can be specified explicitly, as in this example: "$/atom:" ("/" denotes explicitly that the counter has global scope). Another example with an explicit ***cpath*** is the custom local counter variable named "$/proteins[5]/resid:." (See section B.4.) In this example, the ***cpath*** is "$/proteins[5]", the ***catname*** is "resid", and the ***lpath*** is ".". (In section B.4, we never explicitly specified the ***cpath***. This is a source of confusion. When ***cpath*** is omitted, then the program searches up the tree for an ancestor node containing a category with a matching ***catname***. Consequently the ***cpath*** rarely ever needs to be stated explicitly. See section E.2 for more details.)

E.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 usage example, ***lpath*** identifies the location of the variable's corresponding "leaf" object relative to the category scope object (***cpath***). On the other hand, if the the category's scope (***cpath***) was not explicitly stated by the user (which is typical), then the ***lpath*** identifies the location of the leaf object relative to the object in which the variable was referenced (the current-context ".").

E.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 ET 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.

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 ".ET" 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] 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/`.

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

[3] John Stone, Justin Gullingsrud, Paul Grayson, and Klaus Schulten. A system for interactive molecular dynamics simulation. In John F. Hughes and Carlo H. Séquin, editors, *2001 ACM Symposium on Interactive 3D Graphics*, pages 191–194, New York, 2001. ACM SIGGRAPH.

[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.