Introduction to *in silico* synthesis of polymers via PySIMM [Article v1.0]

Alexander G. Demidov[1,2,3*], **B. Lakshitha A. Perera**[1,2,3], **Michael E. Fortunato**[1,2,3], **Sibo Lin**[4], **Coray M. Colina**[1,2,3,5*]

[1]Department of Chemistry, University of Florida, Gainesville, FL 32611, USA; [2]George and Josephine Butler Polymer Research Laboratory, University of Florida, Gainesville, FL 32611, USA; [3]Center for Macromolecular Science and Engineering, University of Florida, Gainesville, FL 32611, USA; [4]Aramco Services Company: Aramco Research Center - Boston, 400 Technology Square, Cambridge, MA, 02139, USA; [5]Department of Materials Science and Engineering, University of Florida, Gainesville, FL 32611

This LiveCoMS document is maintained online on GitHub at https://github.com/AlleksD/pysimm_lcms_guide; to provide feedback, suggestions, or help improve it, please visit the GitHub repository and participate via the issue tracker.

This version dated January 14, 2022

Abstract Pysimm is a framework for molecular simulations of polymers and polymer-based nanostructures, which enables their direct chemical synthesis and preparation. Pysimm facilitates the understanding of novel, amorphous, processable materials for a broad range of applications, including heterogeneous catalysts, adsorbents and gas storage materials, as well as protein-polymer conjugates. This tutorial provides a detailed guide on the construction of atomistic and united-atom models of polymers using Pysimm: an open-source Python Application Programming Interface for atomistic molecular simulations. The API complements and simplifies the work of widely known molecular simulation software, such as LAMMPS, CASSANDRA, NAMD and Amber. Readers should be familiar with the basic concepts of atomistic molecular simulations, as well as the basic knowledge of Python programming language, before attempting to follow this tutorial. This work is separated into 3 main sections. First, the process of building an atomic-level model of a polymer chain from its repetitive units is described. The second section shows how to work with existing forcefields, and how Pysimm can automatically read, recognize, and assign appropriate Force Field parameters to a molecule. The final section discusses how to use Pysimm to construct polymer chains with pre-specified tacticity. The section is also available in the form of an interactive Jupyter notebook tutorial outlining simple guidelines to construct polymer models.

***For correspondence:**
alleks.g.d@ufl.edu (AD); colina@chem.ufl.edu (CMC)

1 Introduction

Molecular dynamics (MD) simulations have traditionally been used to study the dynamics of a broad set of systems, from native protein structures, to polymeric materials, at both the all-atom and coarse-grain level of detail [1, 2]. Recent advances in this area have expanded into studying protein-polymer hybrids in systems where polymers were either covalently or non-covalently attached [3]. Demystifying protein-polymer interactions, dynamic changes over time, and subtle initiator- or polymer-induced changes to protein structure through MD is central to these goals [4, 5]. Therefore, integrated platforms need to be developed that can in part retain the detailed resolution of atomistic simulations while expanding the accessible time and

length scales required for thorough and efficient study of molecular chimeras. Although many force fields (FFs) in the biomolecular and small molecule community are optimized using specific water models to reproduce properties under conditions of interest, some water models clearly perform better in representing the general properties of water than others. Moreover, for polymer-water systems, choosing the correct FF for polymers is often ambiguous, and thorough validation with experimental data is often needed [6–10]. For example, recent work has led to the trustworthiness (through development and validation) of appropriate FFs for PEGylated proteins at the atomistic- and meso-level scales [11, 12]. A major limitation in the development of molecular simulation models for water-soluble polymers and protein-polymer conjugates is the lack of FFs for non-PEG-like water-soluble polymers of interest, as well as the lack of experimental data to validate such FFs. To approach the study of these complex polymeric materials from an atomistic molecular simulation standpoint, it is essential that the FFs used are able to accurately capture the critical polymer-water interactions, as well as the tacticity of such polymer chains. This in turn necessitates a detailed FF analysis, for both polymer and water, to validate that the models implemented are sampling the appropriate conformational space prior to being utilized to aid in the design of new polymeric materials and protein/polymer complex systems.

To enable the simulations, we developed, and made available to the broader community, a **Py**thon based **S**imulation **I**nterface for **M**olecular **M**odeling, PySIMM. The interface is an open source Python package designed to assist in the setup and execution of molecular simulations through a high-level application program interface (API) and abstraction from underlying third-party simulation software. So far it includes the ability to perform molecular dynamic simulations (using the LAMMPS engine[13]), Monte Carlo simulations (using the CASSANDRA software [14]), and a few types of fractional free volume analyses (provided by Poreblazer [15] and/or Zeo++ [16] software). Pysimm has been used for several studies involving simulation of atomic-scale polymer models [17–19] and recently it was expanded [20] to build polymer models convenient for bioconjugates modeling as described in this tutorial.

1.1 Scope

This tutorial is separated into 3 parts and will walk the reader through the entire process of building an atomic-level model of a polymer chain (or a polymer melt) from scratch. The first part is concentrated on the basics of pysimm that allow the reader to build a 'small' molecule as a repetitive unit that can be later used in the construction of a macromolecule. In this, we introduce basic pysimm abstractions and list the ways in which a molecule can be imported to pysimm representation. The second part discusses the work with force field parameters and the extent to which pysimm can automatically read, recognize, and assign appropriate FF parameters to a molecule. We will use an example simulation setup to illustrate how to apply/switch a FF for a molecular system using GAFF, and the CHARMM general force field (CGenFF). Finally, the third part discusses the utilization of the force field assisted random walk application, and its modifications to build a macromolecule. In this part, we specifically focus on the tacticity properties of the constructed polymer chains.

The present work is designed to give the readers a relatively simple, yet versatile instrument to build their own atomic-scale models of polymers with precise control over the chain length (or polydispersity index), monomer sequence in copolymers, and polymer chain tacticity.

2 Prerequisites

2.1 Background knowledge

The tutorial aims to describe the basic concepts for working with the API for molecular modeling. Correspondingly, readers are expected to be familiar with the basic concepts of atomistic molecular simulations such as (i) functional forms of common class-I FFs and their representation of van der Waals and electrostatic interactions; (ii) basic information about the way the MD equations are integrated, i.e. the most appropriate integration schemes, optimal time step and other possible integration settings; (iii) additional positional constraints that are often imposed on a molecular system (e.g. application of the SHAKE algorithm for the simulation of water models).

2.2 Software/system requirements

As readers will work with a Python-based API, the basic knowledge of Python programming language such as: (i) constructors of basic dynamic data structures (lists, sets, and dictionaries), (ii) functions and their arguments, (iii) strings and basic file input/output is required. The main tool that provides the means for molecular simulations is LAMMPS. Thus the next section briefly discusses how to obtain the pysimm code, and integrate it with LAMMPS (either existing or freshly installed). Please note that all instructions are fit for Linux distributions (Ubuntu, OpenSUSE, and CentOS distributions had been tested) and had not been tested on Windows. However, tools like Cygwin and other Linux environment emulators likely can cover the functionality that is necessary to run pysimm with LAMMPS on Windows.

2.2.1 Docker image with pysimm deployed

One convenient way to use pysimm is the work with the Docker image. The image already contains pysimm and all necessary software dependencies to run this tutorial, so if you choose to work with the image there is no need to install pysimm to your OS, as it is described below in this section.

The *Dockerfile.txt* instructions to build the corresponding image are in the root folder of pysimm distribution. The routine for building and usage the Docker image is standard. To build the Docker image from the file, run the following from the root pysimm directory (optionally changing `my_tag` to any meaningful text tag of your image):

```
$ docker build -t pysimm:my_tag -f Dockerfile
    $PWD
```

This will create a Docker image based on the Debian 10 Linux distribution with pysimm and pre-installed LAMMPS, CASSANDRA, Zeo++ v0.3, and PoreBlazer v4.0. If the build is successful the list of your Docker images will contain freshly built pysimm image. The full list can be seen by running `docker images` command.

After the successful build, run the corresponding pysimm image in bash mode:

```
$ docker run -it pysimm:my_tag bash
```

The pysimm source files are kept in `/usr/local/pysimm` folder. Thus one can quickly test the LAMMPS or CASSANDRA modules by running one of the examples, for instance:

```
$ cd /usr/local/pysimm
$ cd Examples/08_ethanol_acetone_mixture
$ python run.py
```

The successful run of the example will create input text files called 'mixture.*' which contain the simulated equilibrated ethanol/acetone mixture.

2.2.2 Native pysimm installation

This subsection describes how to install pysimm on your operating system directly. To get started, clone the pysimm repository from GitHub:

```
$ git clone
    https://github.com/polysimtools/pysimm
```

If the package manager of your operating system is APT, then the script *complete_install.py* can be used to configure pysimm, install LAMMPS from their git repository, and configure the integration between the two software. For that, navigate into the cloned pysimm directory, and run the *complete_install.py*. The `--pysimm` command line argument passed to the script should be the path to which you cloned the pysimm repository (one directory up). The following example assumes the user cloned the repository directly to their home directory.

```
$ python pysimm/complete_install.py --pysimm
    $PWD
```

Parts of pysimm require the use of the numpy package. To use the `complete_install.py` script to install numpy as well, include the `--apt-install` command line argument.

```
$ python pysimm/complete_install.py --pysimm
    --apt-install $PWD
```

For the manual installation (without using *complete_install.py*) please do the following. After pysimm is cloned from GitHub, add the path to the `pysimm` directory to your `PYTHONPATH`, and the path to `pysimm/bin` to your `PATH` variables.

2.2.3 Notes on LAMMPS installation

If you are using your own build of LAMMPS, be sure that the following packages were included in your installation as some functionalities in the example scripts require a subset of these packages: (1) *molecule*; (2) *kspace*; (3) *user-misc*; (4) *misc*; (5) *qeq*; and (6) *manybody*.

Pysimm can integrate seamlessly with relevant parts of the LAMMPS simulation software package through the `pysimm.lmps` module. To configure the integration, locate your LAMMPS executable. For example, if the path to your LAMMPS executable is `/usr/bin/lmp_mpi`, add this path as an environment variable `LAMMPS_EXEC`:

```
$ export LAMMPS_EXEC=/usr/bin/lmp_mpi
```

The user can consider adding all environmental variables to their `.bashrc` file to have them defined permanently.

2.2.4 Other Python packages required

The latest pysimm distribution includes a "Jupyter notebook" example that covers the last part of this tutorial. The example in the notebook form provides an interactive media to work i.e. with the repetitive units that are provided, or moreover, new repetitive units, hence allowing the users to make their own polymer models. The Jupyter notebook environment can be easily installed with any Python package manager (e.g. pip or anaconda). The visualization in the code part of this tutorial is relying on two (possibly) external Python packages: **Matplotlib** for basic plotting, and **NGLView** for in-code molecular visualization.

3 Content and links

The examples included in the tutorial along with several other examples, and the API code templates are available at https://github.com/polysimtools/pysimm pysimm GitHub repository or can be downloaded from the pysimm web page: https://pysimm.org/download/. Detailed pysimm documentation is available at its ReadTheDocs page

https://pysimm.readthedocs.io. Alternatively, all the materials only referent for this tutorial are available in a separate GitHub repository dedicated to this tutorial article https://github.com/AlleksD/pysimm_lcms_guide.

3.1 Tutorial I: Basic molecule managing and system building in pysimm

First we will create an empty `system.System` object and store this in a variable `s`. This system object will contain and organize all of our molecular data:

```
s = system.System()
```

Figure 1 shows the sketch of the created `system.System` object schematically displaying its structure (fields and their cross-references) that we will setup in this tutorial.

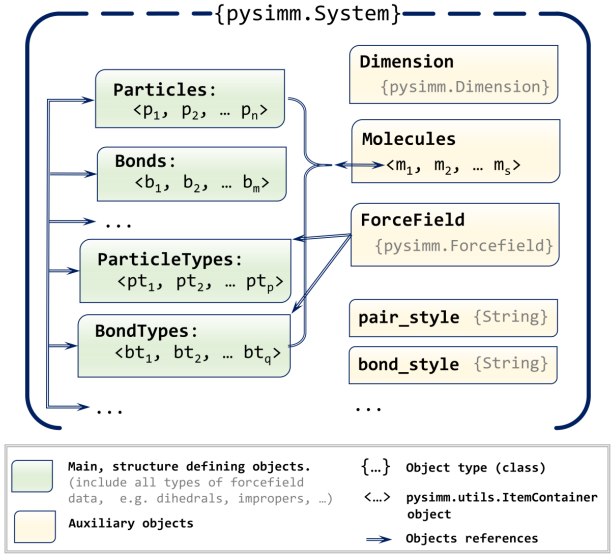

Figure 1. Schematic representation of the `system.System` object that marks major fields of the object and their cross-references.

Next, we need to add a molecule to our system. By default, our system `s` has a container object `s.molecules` that we need to add our new molecule to. We create a new molecule object, `system.Molecule()`, and pass this to the molecule container class method `s.molecules.add()`. This function returns the newly added object to the container, and we store this in variable `m`.

```
m = s.molecules.add(system.Molecule())
```

Now that we have a place to contain all of our molecular data, we need to obtain data describing interactions between atoms from a force field. A `forcefield.Forcefield` object contains the parameters necessary to calculate the energy of your system as well as the logical typing rules for assigning the atom types to the particles of our system. In this example, we will use the GAFF2 force field and instantiate a `Gaff2` force field object that will be stored in variable `f`:

```
f = forcefield.Gaff2()
```

This example assumes the user's knowledge on the GAFF2 atom types that are required for a methane molecule: i.e. "c3" and "hc". Let's get the `system.ParticleType` objects from our `forcefield.Gaff2` object that represent these atom types. Our `forcefield.Gaff2` object has a container object `f.particle_types` that organizes `system.ParticleType` objects and provides a method `f.particle_types.get()` to retrieve specific atom types based on names. The function returns a list of `system.ParticleType` objects, but in this example only one object is returned so we access the first element of the list. Instead of adding the `system.ParticleType` from f, we create a copy, and pass this newly created `system.ParticleType` object to a method that adds it to the particle type container object in our system `s.particle_types`. The `s.particle_types.add()` method returns the newly added object, which we store as the GAFF2 atom type object representation.

```
gaff_c3 = s.particle_types.add(
    f.particle_types.get('c3')[0].copy())
gaff_hc = s.particle_types.add(
    f.particle_types.get('hc')[0].copy())
```

Now we have a System object, `s`, a Molecule object that is stored in our system, `m`, and two GAFF2 atom type objects, `gaff_c3` and `gaff_hc`. Let's start adding atoms.

First, we create the carbon atom (particle) at the origin. Initially, we will add the particle with zero charge, and will derive partial charges later. We instantiate a `system.Particle` object using keyword arguments to set the atom's coordinates x, y, z, the charge, the molecule this particle is a part of, and most importantly, the particle type, `gaff_c3`. Notice that this is a reference to our `system.ParticleType` object that is in our `system.System` object, and not the `forcefield.Gaff2` object. Again we use a container add method to add our new particle to the particles container, and store the newly added particle in variable `c1`.

```
c1 = s.particles.add(system.Particle(
    type=gaff_c3, x=0, y=0, z=0, charge=0,
    molecule=m))
```

The method `add_particle_bonded_to` of a `system.System` object allows the users to add a new particle that should be bonded to an existing particle in the system. If a force field object is also passed into this method, the new bonds, angles, and dihedrals will be created as necessary. The location of the new atom is selected randomly within a carbon-carbon bond length radius around the "parent" particle, and we will later use LAMMPS to perform structural optimization. For each hydrogen atom we want to add, we

create a new `system.Particle` object, with zero charge and the `gaff_hc` type. We also need to identify this as a part of the molecule that already exists, m. This system.Particle object is passed to `s.add_particle_bonded_to()` as the first argument. The second argument is the particle that already exists in our system, in this case c1. The third argument is a reference to our `forcefield.Gaff2` object. The `s.add_particle_bonded_to()` method returns the newly added object, which we store as `h1`, `h2`, `h3` and `h4`.

```
h1 = s.add_particle_bonded_to(
    system.Particle(type=gaff_hc, charge=0,
    molecule=m), c1, f)
h2 = s.add_particle_bonded_to(
    system.Particle(type=gaff_hc, charge=0,
    molecule=m), c1, f)
h3 = s.add_particle_bonded_to(
    system.Particle(type=gaff_hc, charge=0,
    molecule=m), c1, f)
h4 = s.add_particle_bonded_to(
    system.Particle(type=gaff_hc, charge=0,
    molecule=m), c1, f)
```

Now we have added the particles to our system, but there is currently no simulation box. The `system.System` object has a method to construct a simulation box surrounding the atoms it contains, and optionally we can add padding around our molecule. Here we opt to use padding of 10 angstroms.

```
s.set_box(padding=10)
```

Before we optimize our structure, LAMMPS will need to know what type of pair, bond, and angle interactions we are using. These can be defined as string attributes of our `system.System` object.

```
s.pair_style='lj'
s.bond_style='harmonic'
s.angle_style='harmonic'
```

We will use the fire minimization algorithm implemented in LAMMPS. The `lmps` module in pysimm contains a series of shortcut methods to run some types of molecular mechanics simulation. In this case, we will use `lmps.quick_min()` to simultaneously setup and run the energy optimization of the system. The method requires the system object `s`, and the minimization style `min_style` we want to use. Also, we will give each simulation a name so that all log files are kept in separate files.

```
lmps.quick_min(s, min_style='fire',
    name='fire_min')
```

More detailed discussion of the molecular mechanics simulation is in the next tutorial. The `system.System` class also has various methods to format our system data into different file types. Here we will output a PDB (protein data bank), and LAMMPS data file formats:

```
s.write_lammps('methane.lmps')
s.write_pdb('methane.pdb')
```

3.2 Tutorial II: Automatic force field assignment and MD simulations setup

Construction of the whole system from scratch is clearly not the only option for creating the molecular mechanics simulation setup in pysimm. A much more common scenario is loading data from a text file to create the initial system (or several systems) which are then updated, modified, and merged using the API. Table 1 lists text file formats common in computational chemistry that can be used to construct the `system.System` objects in pysimm. The files are read by the collection of static methods of `system` module which have name templates `system.read_***`.

Another utility that pysimm provides for convenient construction of small molecules is the interface to a PubChem database of compounds accessible through a RESTful API [21]. The interface allows users to create `system.System` objects from a PubChem SMILES query. In this example, we will use the SMILES string "CCO" to generate an ethanol molecule using the `system.read_puchem_smiles()` method. The function makes an HTTP request to the PubChem server, which returns an ASCII response formatted as a *.mol* file. The text is interpreted as a `system.System` object that we store in variable ethanol. This system now contains elemental composition bond connectivity and bond orders.

```
ethanol = system.read_pubchem_smiles('CCO')
```

For this example to construct the model of the acetone molecule (the second specie of the mixture) let's use `system.read_pdb()` method to read the coordinates and topology information from the pdb-formatted text file and store the resulting `system.System` object in the variable with the corresponding name.

```
acetone = system.read_pdb('acetone.pdb')
```

The next step is the application of the force field. We will need to use a FF object more than once, so we create a reference to it by instantiating the object and storing a reference to it in variable f.

```
f = forcefield.Gaff2()
```

Our `system.System` objects `ethanol` and `acetone` contain a class method `apply_forcefield()` that accepts a `forcefield.Forcefield` object and *automatically* assigns force field parameters to systems. In this example we use our previously created `forcefield.Gaff2` object and pass it to the `apply_forcefield()` function, as well as specify we would like to derive Gasteiger charges [22].

Table 1. File formats supported in pysimm to import data into `system.System` object

Extension	File format	Method name	Notes
json	ChemDoodle JSON	`read_chemdoodle_json`	
xyz	General XYZ record	`read_xyz`	
yaml	YAML serialisation of `pysimm.system` object	`read_yaml`	
lmps	LAMMPS input	`read_lammps`	Most developed and elaborate implementation. Allows among others to store FF information
cml	Chemical Markup Language file	`read_cml`	
mol	MDL chemical table file	`read_mol`	Both 'V2000' and 'V3000' formats are supported
pdb	Protein data bank file	`read_pdb`	Only particle position ('ATOM', 'HETATM') and bond ('CONECT') information is interpreted

```
ethanol.apply_forcefield(f,
    charges='gasteiger')
acetone.apply_forcefield(f,
    charges='gasteiger')
```

Pysimm has a convenient method that inserts N_k molecules of k different types in a simulation box. The method is called `system.replicate`, it requires a list containing the molecules and a list specifying the number of molecules of each type to insert. Additionally, it also takes density input in g/mL to set up the simulation box. Let's use it to make an initial box with 1-to-1 ethanol/acetone mixture.

```
s = system.replicate([ethanol, acetone],
    [200, 200], density=0.3)
```

Now that we have the initial system with molecules packed in a box, let's equilibrate it in 4 short steps. First, we carry out the energy minimization. Then we run 3 separate instances of molecular dynamics using the default 1 fs integration step for each: NVT MD for 10 ps at low density, and two NPT MD simulations for 100 ps each. At the beginning of the NVT simulation, we will generate new velocities at 100 K, and heat our system from 100 K to 300 K. At the beginning of the first NPT simulation, we will re-generate new initial velocities and start with a positive compressive pressure of 1000 atm which will be decreased to 1 atm during the run. Finally, the second NPT simulation will be held at 1 atm and 300 K to generate some sampling data. The settings for each of those simulation steps are configured as a simple Python dictionary where keys closely follow the names of corresponding commands in LAMMPS. The settings use LAMMPS "real" system of units (time in fs, distance in , energy in Kcal/mole, see the LAMMPS documentation [23]), and have default values defined for each keyword (see the pysimm documentation).

```
min_settings = {'name': 'fire_min',
    'min_style': 'fire'}
```

```
nvt_settings = {'name': 'nvt_md',
    'ensemble': 'nvt',
   't_start': 100, 't_stop': 300, 'new_v':
       True,
    'length': 10000, 'timestep': 1}
```

```
npt1_settings = {'name': 'npt_md',
    'ensemble': 'npt',
   'temp': 300, 'new_v': True,
   'p_start': 1000, 'p_stop': 1,
   'length': 100000, 'timestep': 1}
```

```
npt2_settings = npt1_settings.copy()
npt2_settings['p_start'] = 1
```

Let's instantiate the LAMMPS simulation object that governs the simulation process and will contain other setting objects responsible for various aspects of the simulation. Right away let's add to the simulation container the `OutputSettings` object which will define the style and print frequency of LAMMPS simulation output.

```
sim = lmps.Simulation(s)
sim.add(lmps.OutputSettings( thermo={'freq':
    500, 'style': 'custom',
'args': ['step', 'temp', 'etotal', 'press',
   'density']}))
```

Next, we add an optimization object and a molecular dynamics object using corresponding methods of the `lmps.simulation` – they will be added to the LAMMPS script (and thus executed) in the order they appear in the code. Finally, we send the `run` command that starts the whole simulation sequence.

```
sim.add_min(**min_settings)
sim.add_md(**nvt_settings)
sim.add_md(**npt1_settings)
sim.add_md(**npt2_settings)
sim.run()
```

In the present setup it is easy to replace the GAFF-2 force field with a different one (one of the supported by the API). To use a different FF one changes the reference to the FF object: write `f = forcefield.Charmm()` instead of `f = forcefield.Gaff2()`, and re-run the above workflow.

In the final part of this tutorial let's see how the same equilibration simulations will differ if molecules are typed with a different FF. As an example let's use CHARMM general force field (CGenFF). Similar to GAFF, pysimm can automatically recognize and type most common CGenFF atom types for H, C, N, O, and S chemical elements.

After the simulations are finished we can parse the output log file and draw the density of the system during the overall 200 ps of both NPT runs. Figure 2 shows the density of acetone-ethanol 1-to-1 mixture as read from the LAMMPS logs of 2 similar NPT simulations that use GAFF2 and CGenFF force fields. The results are close to each other and in this case the difference is likely caused by the difference in compatibility of corresponding FF with Gasteiger partial charges.

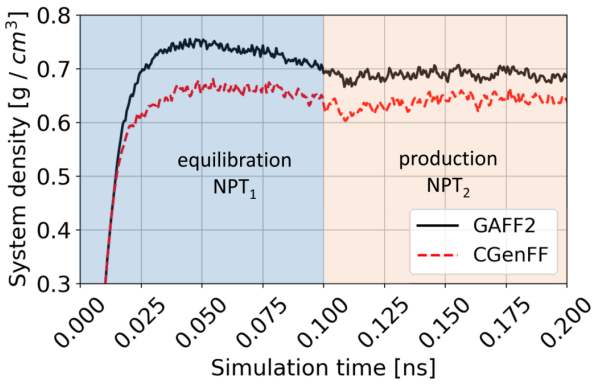

Figure 2. Comparison of ethanol/acetone 1-to-1 mixture density simulated with the usage of GAFF2 and CGenFF forcefields. The LAMMPS simulation setup and automatic FF typing are done in pysimm

3.3 Tutorial III: Simulated random walk polymerization with pysimm

3.3.1 Preparing the repetitive unit

Let's load the repetitive unit of the polymer creating `system.System` object from a .pdb file that has `'CONECT'` information (pysimm can interpret this data to make bonds between the atoms of the system). The `system.read_pdb` method supports additional string parameter that points to the CHARMM stream (.str) file which can be used to update the charges of the particles in the system.

```
sst = system.read_pdb(
    '../data/cbma_monomer.pdb',
    str_file='../data/cbma_monomer.str')
```

To display the repetitive unit on the Jupyter canvas, let's use nglview package. It also can show additional label information for each atom; e.g. indexes of all atoms of the system. Alternatively, if nglview is not available, we encourage the user to open the imported structure in any other standalone molecular viewer (like VMD [24] or Avogadro [25]).

```
view = nglview.show_structure_file(
    '../data/cbma_monomer.pdb')
view.add_label(color='black', scale=1.3,
    labelType='text', zOffset=2.0,
    attachment='middle_center', labelText =
    [str(pt.tag) for pt in sst.particles])
```

Figure 3 shows an example of the definition of the head and tail atoms in the carboxybetaine methacrylate [CBMA] repetitive unit. As shown above, the undercoordinated carbon atoms (carbons with incomplete valency) have indices 1 and 2. They will be the head (the atom with which the current repeating unit connects to the previous repeating unit) and the tail (the atom to which the next repeating unit connects) during the pysimm polymerization process. Let's mark those atoms with the corresponding text labels in our system

```
lnkr_atoms = {'head': 1, 'tail': 2}
for k, v in lnkr_atoms.items():
    sst.particles[v].linker = k
```

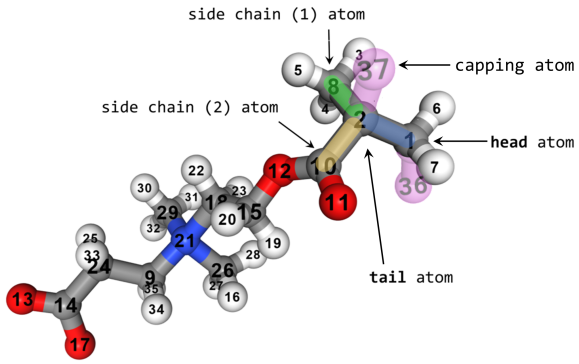

Figure 3. Detailed description of the atoms in the CBMA repetitive unit, that is important for the polymerization and the tacticity analyzer methods of `pysimm.random_walk`

Let's type the repetitive unit with the basic pysimm CGenFF automatic typing tool. The charges of all particles were read from the .str file, so there is no need to reassign them (if not, pysimm can do it using e.g. the Gasteiger method)

```
ff = forcefield.Charmm()
sst.apply_forcefield(ff, charges=None)
```

3.3.2 Making the polymer and checking its tacticity

Once the FF types and charges are defined, one can use the force field assisted random walk method. The method

requires the `system.System` object representing a monomer, and the integer number of monomers to be concatenated. The other optional (keyword) arguments presented in the example define the density of the final output system, flag to indicate to output the construction trajectory (in .xyz format), and a flag to indicate whether the final chain will be unwrapped. Let's build a short (15 repetitive units) chain.

```
sngl_chain = random_walk(sst, 15,
    forcefield=ff, density=0.01, traj=False,
    unwrap=True)
```

Next, let's check the tacticity of the created oligomer. The `check_tacticity()` method of the `random_walk` application analyzes the local geometry of atoms for polymers that can have tacticity. The method returns the distribution of meso-to-racemo diads along the backbone of the macromolecule (as shown in Figure 4).

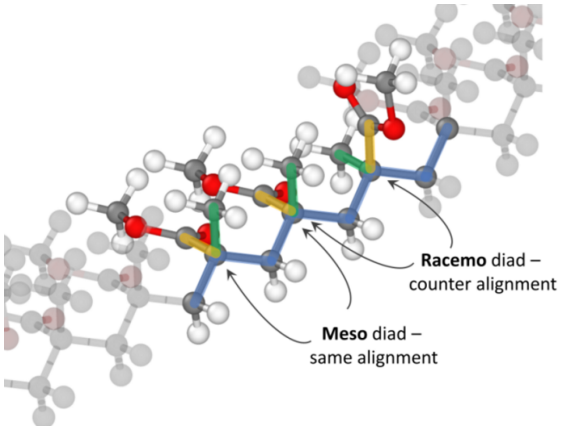

Figure 4. Schematic representation of a meso and a racemo diad along the polymer backbone. In the case of a meso diad both the blue-green-yellow vector triples of the two consecutive monomers have the same alignment, whereas in a racemo diad, one of the blue-green-yellow vector triples is right-handed while the other is left-handed.

The input parameters for the method are:

- A pysimm system that represents a macromolecule.
- A list with 4 integers that defines the indices of the node atoms in a repetitive unit of the macromolecule. The indices in their order represent: (1) the first atom of the backbone; (2) the second atom of the backbone; (3) the first atom of the first side chain (or methyl, or hydrogen); (4) the first atom of the second side chain. Note that the colors of the atom indices match the colors of the vectors on the figure 4.
- Number of atoms (particles) in the repetitive unit of the macromolecule

The second variable of the `check_tacticity` output is the list that shows whether each two consecutive repetitive units in the chain form either a meso (True) or a racemo (False) diad. Let's examine the obtained chain and print the result in the form of a simple 2-column histogram that shows the number of meso and racemo diads in the chain. The indices for the analyzed repetitive unit are highlighted in figure 3, and they are 1, 2, 8, and 10.

```
tacticity_stat = check_tacticity(sngl_chain,
    [1, 2, 8, 10], len(sst.particles))
```

In this case, among 15 monomers (thus 14 diads), we see that most diads have a meso configuration (figure 5-(a)), meaning that the majority of the monomer pairs in the chain have the same orientation. Only a very few monomer pairs (10%-30%, depending on the run) will form racemo diads. In this implementation of the random walk there is no explicit control of the following monomer orientation, hence all monomers attached initially had to form meso diads. However, due to the geometry optimization and short MD simulations that are implemented in the random walk method, the orientation of neighboring monomers occasionally can switch, thus giving some number of racemo diads. Therefore, depending on the strength of the energy barrier for rotation, during longer MD simulations the polymer can relax to an atactic state. However, pysimm can also allow gaining more control over the polymer tacticity.

3.3.3 Polymerization with controlled tacticity

While polymers synthesized through conventional radical polymerization are usually atactic, newer polymerization catalysts are able to offer high purity syndiotactic or isotactic polymers, often with markedly different physical properties. In this part, let's concentrate on building models of syndiotactic or isotactic polymers. We will use another method which is called `random_walk_tacticity`, which allows the user to define the orientation of the next attached monomer during the polymer building phase. For that, some additional modifications should be done to the repetitive unit that was previously used. The `random_walk_tacticity` method requires a capped monomer, therefore let's add capping carbon atoms to the linker atoms of our repetitive unit (atoms with tags 1 and 2, see figure 3). The capping atoms (ones with tags 36 and 37 on figure 3) are dummy atoms, so we can assign them any types, for convenience, let's reuse a type that is already present in our system: like `'CG321'` an sp^3 hybridized carbon with 2 hydrogen atoms attached. Moreover, the capping atoms will be undercoordinated, but it is not important in this case, as they are dummy atoms and will be removed during the simulated polymerization.

Both capping atoms should be decorated with an additional field named `rnd_wlk_tag` that contains a string with either `'head_cap'` or `'tail_cap'` value, respectively. Finally, another label `linker='mirror'` should be assigned to an atom of

the system; the atom with this label together with 2 labeled atoms of the backbone will be used to build a mirror plane for construction of the reflected repetitive unit for syndiotactic insertion. In our case, the most convenient `'mirror'` atom is the freshly added `'head_cap'` atom (it will be connected to the atom #1 to the vacant position of the tetrahedron formed by atoms with tags 2, 6, and 7; see figure 3).

To shorten the tutorial text we do not list the whole code that performs the aforementioned labeling, but only the first and last lines of it (please refer to the code in *.py* or *.ipynb* files for the implementation with detailed inline commentaries).

```
new_sst = sst.copy()
captype = new_sst.particle_types.get(
    'CG321')[0]
...
new_sst.objectify()
```

Note that preparations written above are not needed if capping atoms are already present in the initial structure of the repetitive unit. Please, refer to example 12 of pysimm distribution, where the tacticity-controlled polystyrene chain is built (*Examples/12_tacticities/polystyrene_nosim.py*).

To control the tacticity of the chain, the method has `tacticity` keyword argument that accepts a real number $n \in [0, 1]$, which defines the relative number of isotactic insertions (note that generally for any value $n \in (0, 1)$ the polymer is considered atactic). Thus, $n = 1$ will be a fully isotactic chain, $n = 0$ will be syndiotactic chain, whereas $n = 0.5$ will be a chain with equal number of isotactic and syndiotactic insertions. This method also accepts keyword strings as values of tacticity key. One can use either $n = 1$ or 'isotactic', $n = 0$ or 'syndiotactic', and $n = 0.5$ or 'atactic'. For more details and options of `random_walk_tacticity` method please refer the pysimm documentation.

First, let's run the `random_walk_tacticity` in "no simulation" mode. In that regime next monomer will be put to an approximately correct, geometrically calculated position without the FF optimization and NVE molecular dynamics simulations.

```
polymer_nosim =
    random_walk_tacticity(new_sst, 15,
    forcefield=ff, sim='no',
    tacticity='syndiotactic', density=0.01)
```

The result is a syndiotactic chain, and all diads in this case are clearly racemo- diads (figure 5-(b)). Now let's do the same but with FF optimization of the growing polymer chain between every monomer insertion, and see how many diads will be reconfigured from racemo- to meso- geometry.

```
polymer = random_walk_tacticity(new_sst, 15,
    forcefield=ff, tacticity=0.0)
```

The figure 5-(c) confirms that the optimization can change the initial distribution of the monomer orientations. However, the chain obtained with `random_walk_tacticity` has more racemo than meso diads (as the example was based on creating a syndiotactic polymer), compared to the chain obtained by the original `random_walk` method (10%-30%).

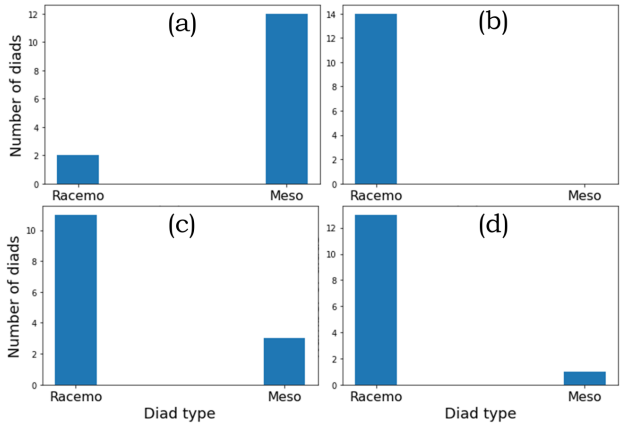

Figure 5. Meso-to-racemo diad distribution in a single pCBMA chain containing 15 repetitive units made with different methods of `pysimm.random_walk` application. (a) Standard `random_walk` method. (b) `random_walk_tacticity` method with no simulations applied, (c) `random_walk_tacticity` method with simulations on, (d) `random_walk_tacticity` run with a post-optimization.

3.3.4 A setup to construct a polymer chain with exact tacticity

In the previous section it was shown that in pysimm, one can easily construct a "no-simulation" polymer chain. The simple chain (but illustrative for tacticity explanations) is constructed by connecting the repetitive units to each other without force field simulations. The tacticity of that chain is easy to set to be exact, unlike the tacticity of a chain built with the FF simulations enabled.

The robustness of the FF functional form allows one to run the simulations using the polymer chain built by `random_walk_tacticity` with `no_sim` flag on as an initial structure. Long enough MD simulation with the fixed (via SHAKE) angles between the side chains will result in a relaxed polymer chain with exactly the same tacticity as it was constructed at the beginning. Please note, that depending on the geometry of your repetitive unit, the initial structure of the chain of concatenated repetitive units might be defined approximately. If possible we recommend putting the linker atoms (head and tail atoms) into anti-periplanar positions (see the CBMA repetitive unit in this tutorial, figure 3).

Below is a small code sample that sets up (in pysimm) the MD simulations using the 'no simulation' configuration and results in a relaxed polymer chain. First let's invoke the `Simulation` object of the `pysimm.lmps` module using the copy of the initial "no simulation" polymer chain.

```
pmer_shake = polymer_nosim.copy()
```

```
sim = lmps.Simulation(pmer_shake,
    name='shake_relax', log='shake.log')
```

Next, let's add a SHAKE directive to the simulation. The aim is to keep the angle between the side chains of each repetitive unit fixed (figure 3). To apply shake to an angle in LAMMPS, the user is required to know the identifiers of bond types for the corresponding angle and the identifier of the angle type. Presume here, that we don't know the indexes of corresponding bonds and angle, but we know the types of particles that form this angle, which will give us the required indexes.

```
bnd1 = polymer.bond_types.get(
    'CG2O2,CG301')[0].tag
bnd2 = polymer.bond_types.get(
    'CG301,CG331')[0].tag
angl = polymer.angle_types.get(
    'CG331,CG301,CG2O2')[0].tag
sim.add_custom('fix shck_fix all shake 0.001
    40 0 b {} {} a {}'.format(bnd1, bnd2,
    angl)
```

Finally, let's run a short (30 ps) NVE MD simulation with a single built chain to give it enough time to relax all non-fixed bonds, angles and dihedrals.

```
sim.add_md(ensemble='nve', limit=0.1,
    length=30000)
sim.run()
pmer_shake.unwrap()
```

The tacticity of the resulting chain remains almost perfectly syndiotactic, as it was before the relaxing MD simulation (figure 5 (b) and (d)). The exception is the geometry of the terminal repetitive unit that eventually can switch during the MD run, giving one meso diad.

4 Author Contributions

AGD, BLAP, MEF and CMC conceived and wrote the online tutorials 1 and 2. All authors conceived and wrote tutorial 3 and contributed to manuscript writing. AGD maintains the source repository of the project at GitHub.

For a more detailed description of author contributions, see the GitHub issue tracking and changelog at https://github.com/AlleksD/pysimm_lcms_guide.

5 Other Contributions

The authors would like to acknowledge other pysimm developers and testers: Ping Lin, Dylan Anstine, Grit Kupgan, Shalini Jayaraman Rukmani, and Aravinda Munasinghe.

For a more detailed description of contributions from the community and others, see the GitHub issue tracking and changelog at https://github.com/AlleksD/pysimm_lcms_guide.

6 Potentially Conflicting Interests

The authors declare no potential conflict of interests.

7 Funding Information

The authors would like to acknowledge U.S. Department of Energy for funding (grant DE-FG02-17ER16362).

Author Information

ORCID:

B. Lakshitha A. Perera: 0000-0003-3015-3529

Coray M. Colina: 0000-0003-2367-1352

References

[1] **Frenkel D**, Smit B. Understanding Molecular Simulation: From Algorithms to Applications. New York: Academic Press; 1996.

[2] **Leach AR**. Molecular Modelling: Principles and Applications. Prentice Hall; 2001.

[3] **Russell AJ**, Baker SL, Colina CM, Figg CA, Kaar JL, Matyjaszewski K, Simakova A, Sumerlin BS. Next Generation Protein-Polymer Conjugates. AIChE J. 2018; 64(9):3230–3245. https://doi.org/https://doi.org/10.1002/aic.16338.

[4] **Lin P**, Colina CM. Molecular Simulation of Protein–Polymer Conjugates. Curr Opin Chem Eng. 2019; 23:44–50. https://doi.org/10.1016/j.coche.2019.02.006.

[5] **Perera BLA**, Colina CM. Cluster Formation of Initiators as a Tool to Impose Conformational Stability to Unstructured Regions of a Protein. Mol Phys. 2021; 119(19-20):e1963000. https://doi.org/10.1080/00268976.2021.1963000.

[6] **Kubota K**, Fujishige S, Ando I. Solution Properties of Poly (N-Isopropylacrylamide) in Water. Polym J. 1990; 22(1):15–20. https://doi.org/10.1295/polymj.22.15.

[7] **Kubota K**, Hamano K, Kuwahara N, Fujishige S, Ando I. Characterization of Poly (N-Isopropylmethacrylamide) in Water. Polym J. 1990; 22(12):1051–1057. https://doi.org/10.1295/polymj.22.1051.

[8] **Tacx JCJF**, Schoffeleers HM, Brands AGM, Teuwen L. Dissolution Behavior and Solution Properties of Polyvinylalcohol as Determined by Viscometry and Light Scattering in DMSO, Ethyleneglycol and Water. Polymer. 2000; 41(3):947–957. https://doi.org/10.1016/S0032-3861(99)00220-7.

[9] **Bucholz EW**, Haskins JB, Monk JD, Bauschlicher Jr CW, Lawson JW. Phenolic Polymer Solvation in Water and Ethylene Glycol, I: Molecular Dynamics Simulations. J Phys Chem B. 2017; 121(13):2839–2851. https://doi.org/10.1021/acs.jpcb.7b00326.

[10] **Rukmani SJ**, Kupgan G, Anstine DM, Colina CM. A Molecular Dynamics Study of Water-Soluble Polymers: Analysis of Force Fields From Atomistic Simulations. Mol Simul. 2019; 7022:310–321. https://doi.org/10.1080/08927022.2018.1531401.

[11] **Ramezanghorbani F**, Lin P, Colina CM. Optimizing Protein–Polymer Interactions in a Poly(Ethylene Glycol) Coarse-Grained Model. J Phys Chem B. 2018; 122(33):7997–8005. https://doi.org/10.1021/acs.jpcb.8b05359.

[12] **Munasinghe A**, Mathavan A, Mathavan A, Lin P, Colina CM. Molecular Insight into the Protein–Polymer Interactions in N-Terminal PEGylated Bovine Serum Albumin. J Phys Chem B. 2019; 123(25):5196–5205. https://doi.org/10.1021/acs.jpcb.8b12268.

[13] **Plimpton S**. Fast Parallel Algorithms for Short-Range Molecular Dynamics. J Comput Phys. 1995; 117(1):1–19. https://doi.org/10.1006/JCPH.1995.1039.

[14] **Shah JK**, Marin-Rimoldi E, Mullen RG, Keene BP, Khan S, Paluch AS, Rai N, Romanielo LL, Rosch TW, Yoo B, Maginn EJ. Cassandra: An Open Source Monte Carlo Package for Molecular Simulation. J Comput Chem. 2017; p. 1727–1739. https://doi.org/10.1002/jcc.24807.

[15] **Sarkisov L**, Harrison A. Computational Structure Characterisation Tools in Application to Ordered and Disordered Porous Materials. Mol Simul. 2011; 37(15):1248–1257. https://doi.org/10.1080/08927022.2011.592832.

[16] **Willems TF**, Rycroft CH, Kazi M, Meza JC, Haranczyk M. Algorithms and Tools for High-Throughput Geometry-Based Analysis of Crystalline Porous Materials. Microporous and Mesoporous Mater. 2012; 149(1):134–141. https://doi.org/10.1016/j.micromeso.2011.08.020.

[17] **Rukmani SJ**, Lin P, Andrew JS, Colina CM. Molecular Modeling of Complex Cross-Linked Networks of PEGDA Nanogels. J Phys Chem B. 2019; 123(18):4129–4138. https://doi.org/10.1021/acs.jpcb.9b01622.

[18] **Song C**, Hu F, Meng Z, Li S, Shao W, Zhang T, Liu S, Jian X. Atomistic Structure Generation of Covalent Triazine-Based Polymers by Molecular Simulation. RSC Adv. 2020; 10(8):4258–4263. https://doi.org/10.1039/c9ra11035f.

[19] **Anstine DM**, Strachan A, Colina CM. Effects of an Atomistic Modeling Approach on Predicted Mechanical Properties of Glassy Polymers via Molecular Dynamics. Modell Simul Mater Sci Eng. 2020; 28(2):025006. https://doi.org/10.1088/1361-651X/ab615c.

[20] **Demidov AG**, Perera BLA, Fortunato ME, Lin S, Colina CM. Update 1.1 to "Pysimm: A Python Package for Simulation of Molecular Systems". SoftwareX. 2021; 15:100749. https://doi.org/10.1016/j.softx.2021.100749.

[21] **Kim S**, Thiessen PA, Cheng T, Yu B, Bolton EE. An Update on PUG-REST: RESTful Interface for Programmatic Access to PubChem. Nucleic Acids Res. 2018; 46(W1):W563–W570. https://doi.org/10.1093/nar/gky294.

[22] **Gasteiger J**, Marsili M. Iterative Partial Equalization of Orbital Electronegativity—a Rapid Access to Atomic Charges. Tetrahedron. 1980; 36:3219–3288. https://doi.org/10.1016/0040-4020(80)80168-2.

[23] units command – LAMMPS documentation;. https://docs.lammps.org/units.html.

[24] **Humphrey W**, Dalke A, Schulten K. VMD: Visual Molecular Dynamics. J Mol Graphics. 1996; 7855(December 1995):33–38. http://www.ks.uiuc.edu/Research/vmd.

[25] **Hanwell MD**, Curtis DE, Lonie DC, Vandermeersch T, Zurek E, Hutchison GR. Avogadro: An Advanced Semantic Chemical Editor, Visualization, and Analysis Platform. J Cheminf. 2012; 4(1):1–17. http://avogadro.cc.